14:332:452 SOFTWARE ENGINEERING

System Design

Parking Garage Project

R. Romanowski, R. Roman, J. Jacob, B. Goodacre, M. Rodriguez

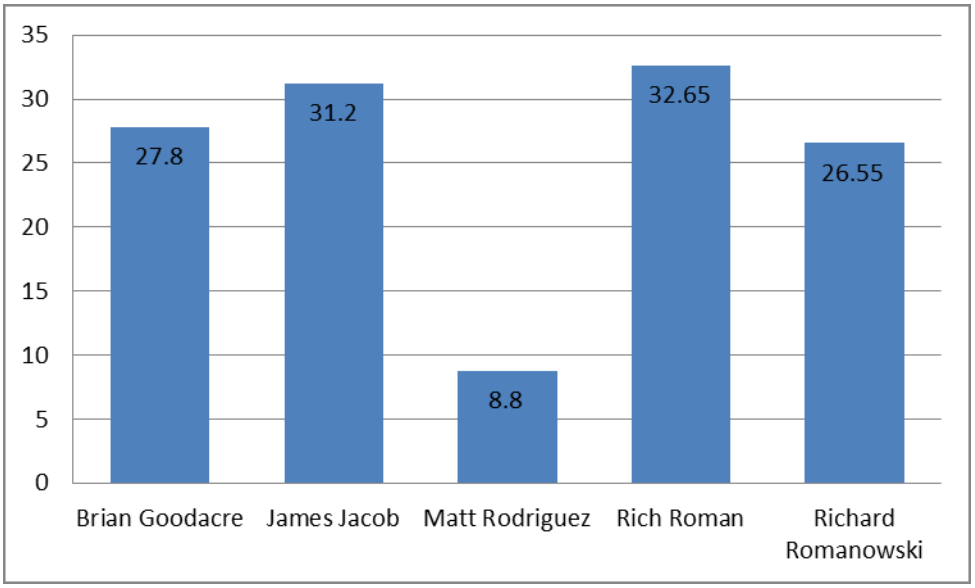
3/11/2011

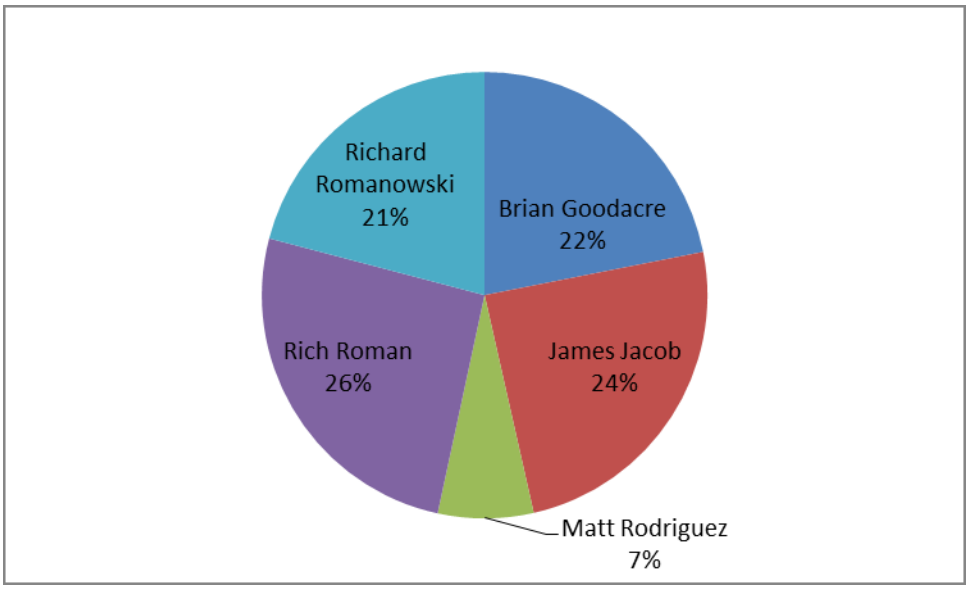
This is the second report of Software Engineering class. It details the design of the project. The team has met multiple times and worked together and independently to put together this report that will keep up on track to completing this large project.

Table of Contents

Individual Contributions Breakdown

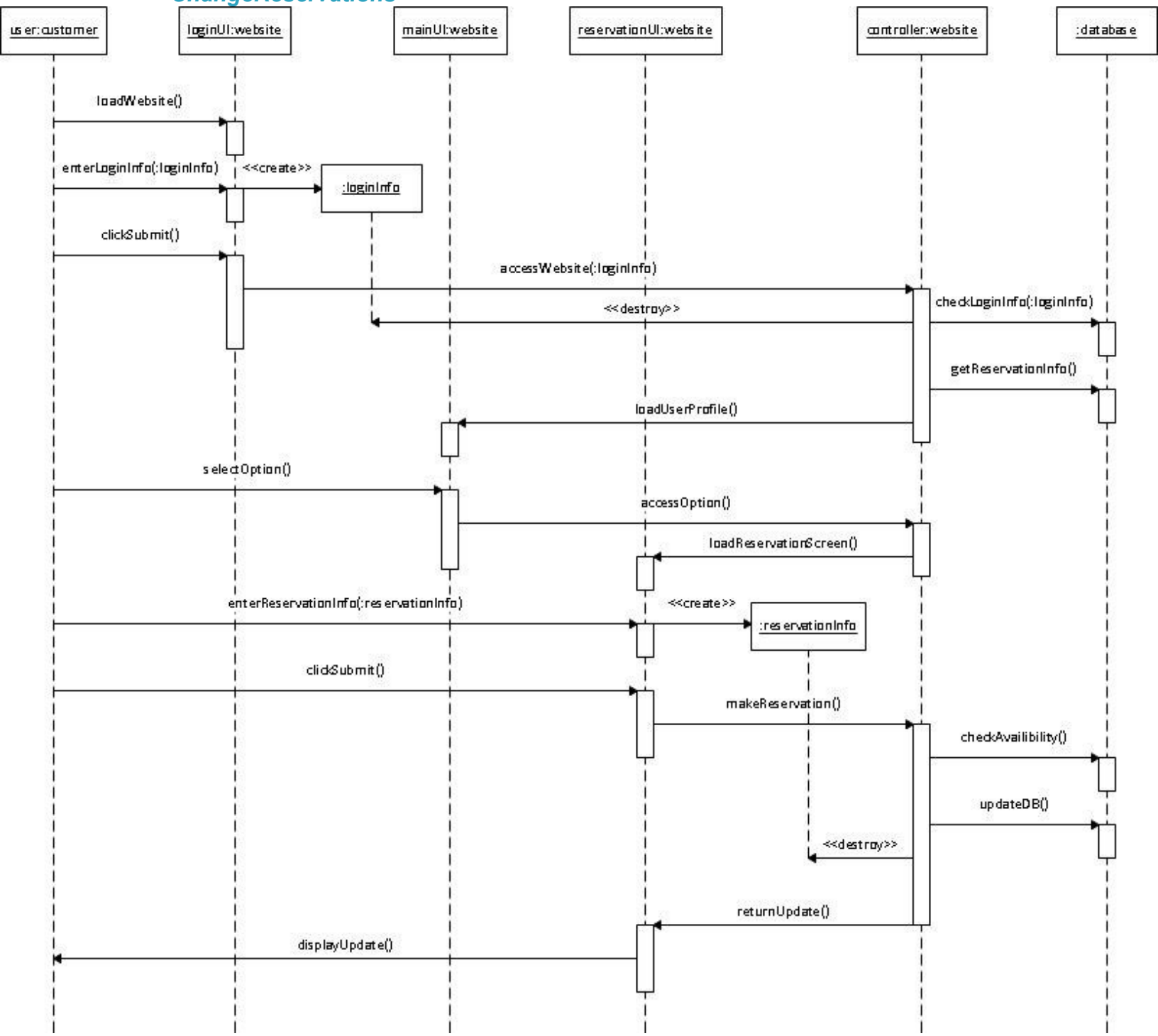
Responsibility	Effort	Brian Goodacre	James Jacob	Matt Rodriguez	Rich Roman	Richard Romanowski
Project Management	3		100%			
Team Effort Breakdown	1	100%				
Compiling Report	4	100%				
Sec 3. Interaction Diagrams						
Interaction Diagrams-Diagram	7		90%			10%
Interaction Diagrams-Write up	6		100%			
Sec 4. Class Diagram and Interface Specification						
Class Diagram-Diagram	11				90%	10%
Class Diagram-Write up	13				90%	10%
Data Type-Diagram	7				90%	10%
Sec 5. System Architecture and System Design						
Architectural Styles-Write up	5		5%		95%	
Identifying Subsystems-UML	7	90%	10%			
Mapping Subsystems to Harddrive-Write up	5	95%	5%			
Persistent Data Storage-Write up	6	95%	5%			
Network Protocol-Write up	4	10%	80%			10%
Global Control Flow-Write up	4		5%			95%
Hardware Requirements-Write up	4		5%	95%		
Sec 6. Algorithms and Data Structures						
Algorithms -Write up	6		90%			10%
Data Structures-Write up	5		5%			95%
Sec 7. User Interface Design and Implementation						
Pencil Drawing	4					100%
Photoshop	5			100%		
Write up	5					100%
Sec 8. Progress Report and Plan of Work						
Progress Report-Write up	4		5%			95%
Plan of Work-Diagram	5	100%				
Breakdown of Responsibilities-Write up	5	5%	95%			
Sec 9. References	1	40%	20%			40%


35
30
25
20
15
10
5
0
27.8
31.2
8.8
32.65
26.55
Brian Goodacre
James Jacob
Matt Rodriguez
Rich Roman
Richard Romanowski


Richard Romanowski 21%
Brian Goodacre 22%
Rich Roman 26%
James Jacob 24%
Matt Rodriguez 7%

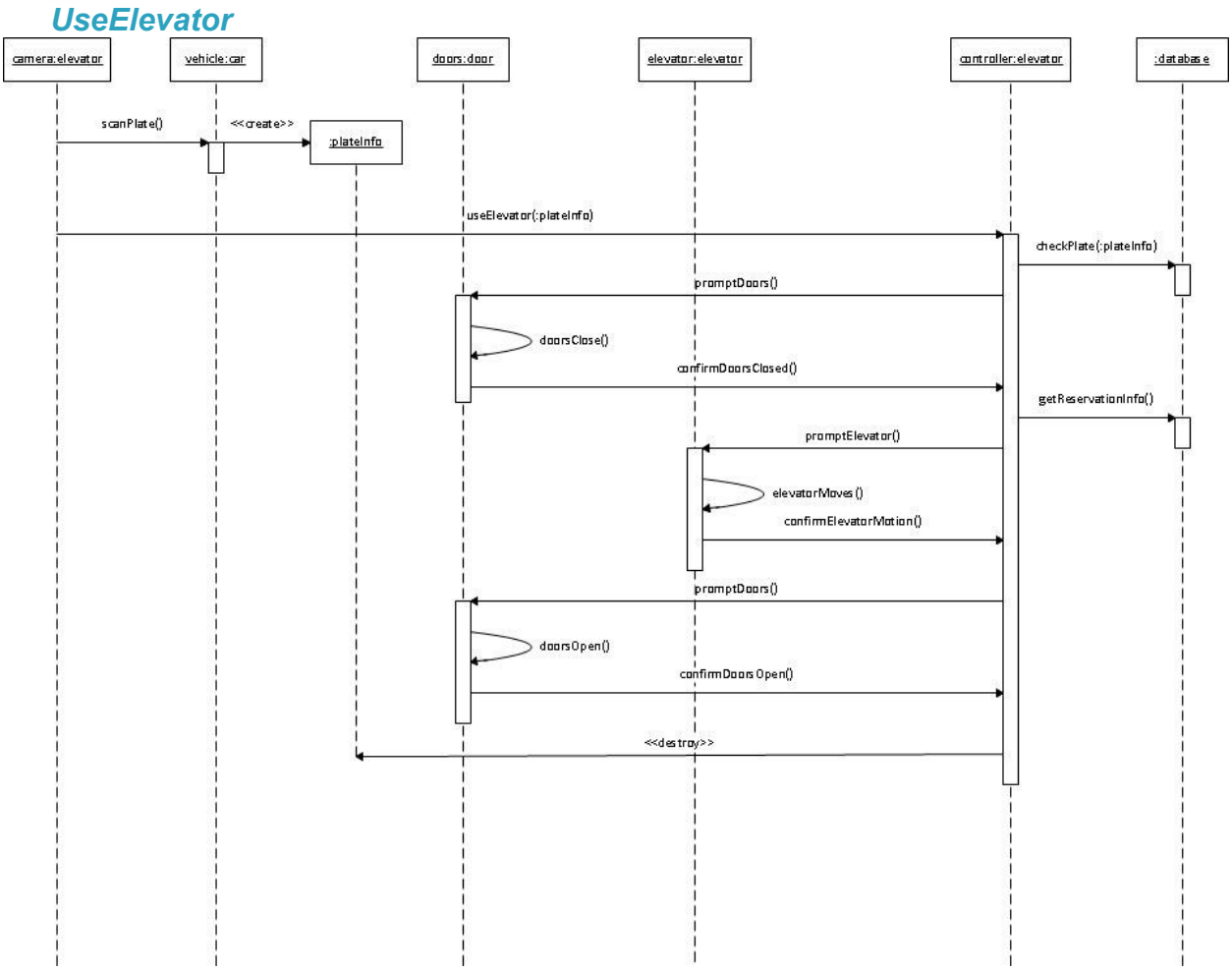
Interaction Diagrams

ChangeReservations

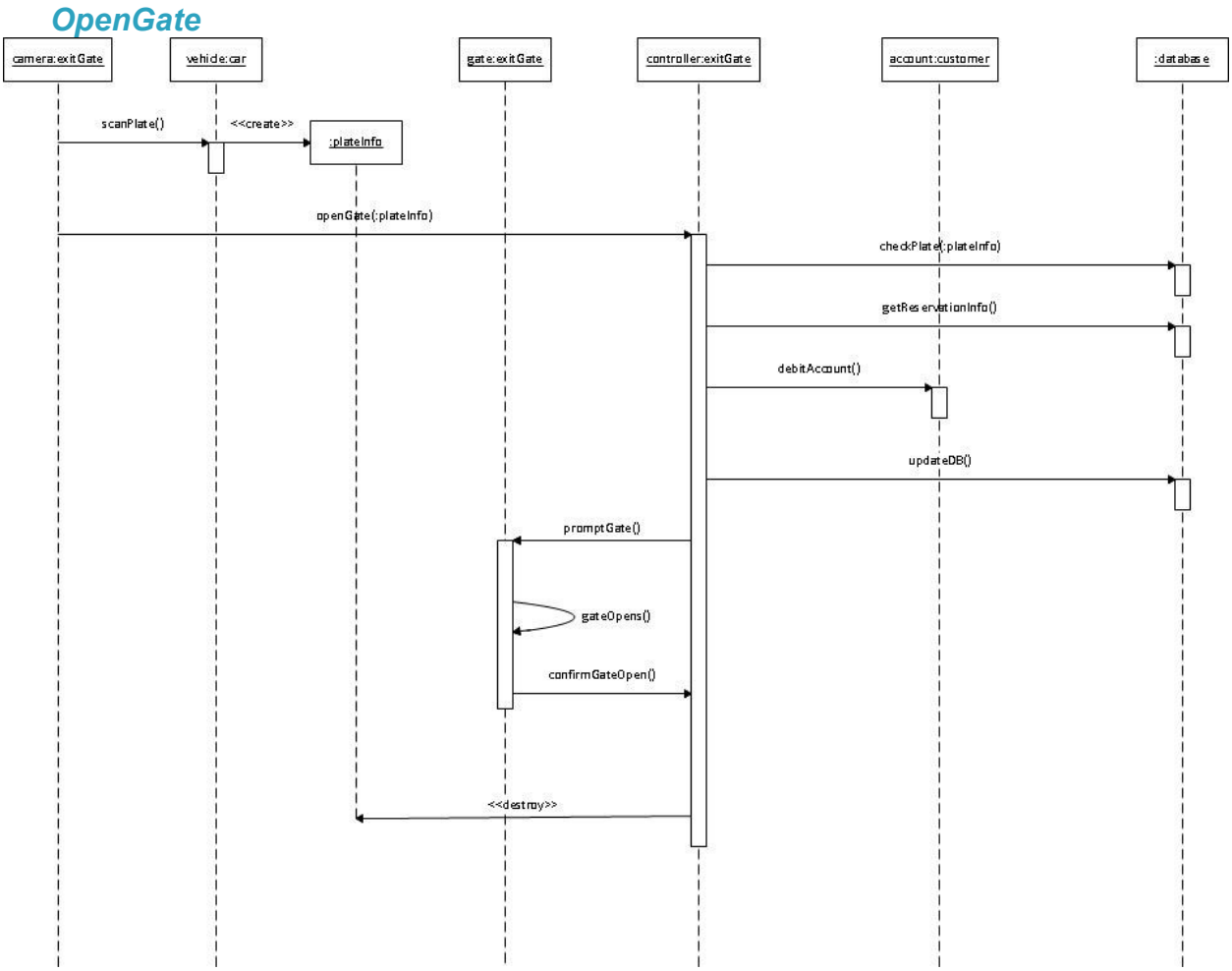


The High Cohesion principle was employed to assign responsibilities to objects within this use case. All objects, such as the user interfaces and the database report to the controller object, which then send messages in order to invoke the methods required to complete a task.

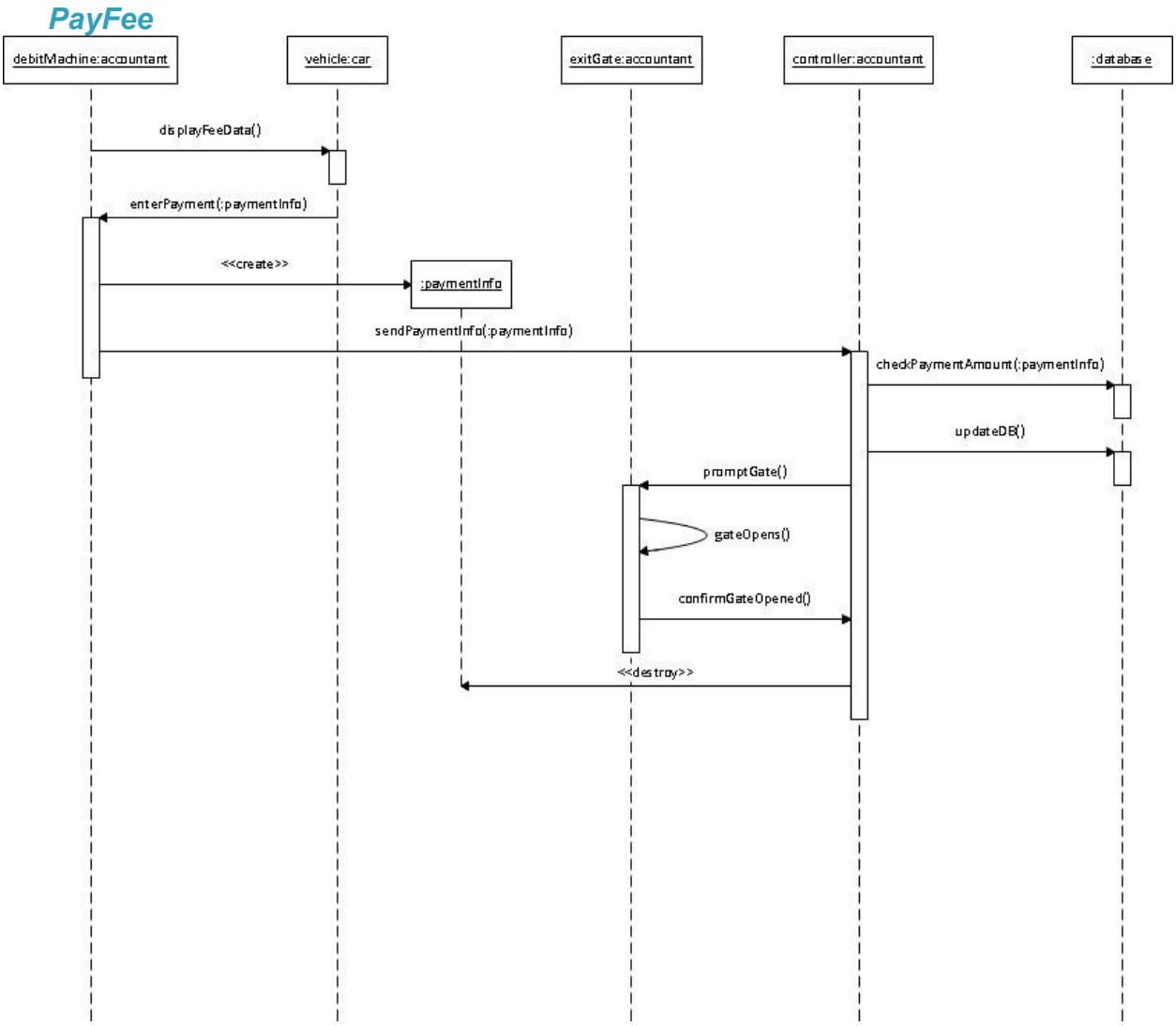
Thus, the controller takes on the responsibility of maintaining communication between the different objects within this use case.



This use case utilized the High Cohesion property to assign responsibilities to internal objects. The controller object took on a bulk of the responsibilities in terms of sending messages between objects. Besides the controller, each object was responsible for only its own tasks, thereby avoiding having too many extraneous chores for these objects.

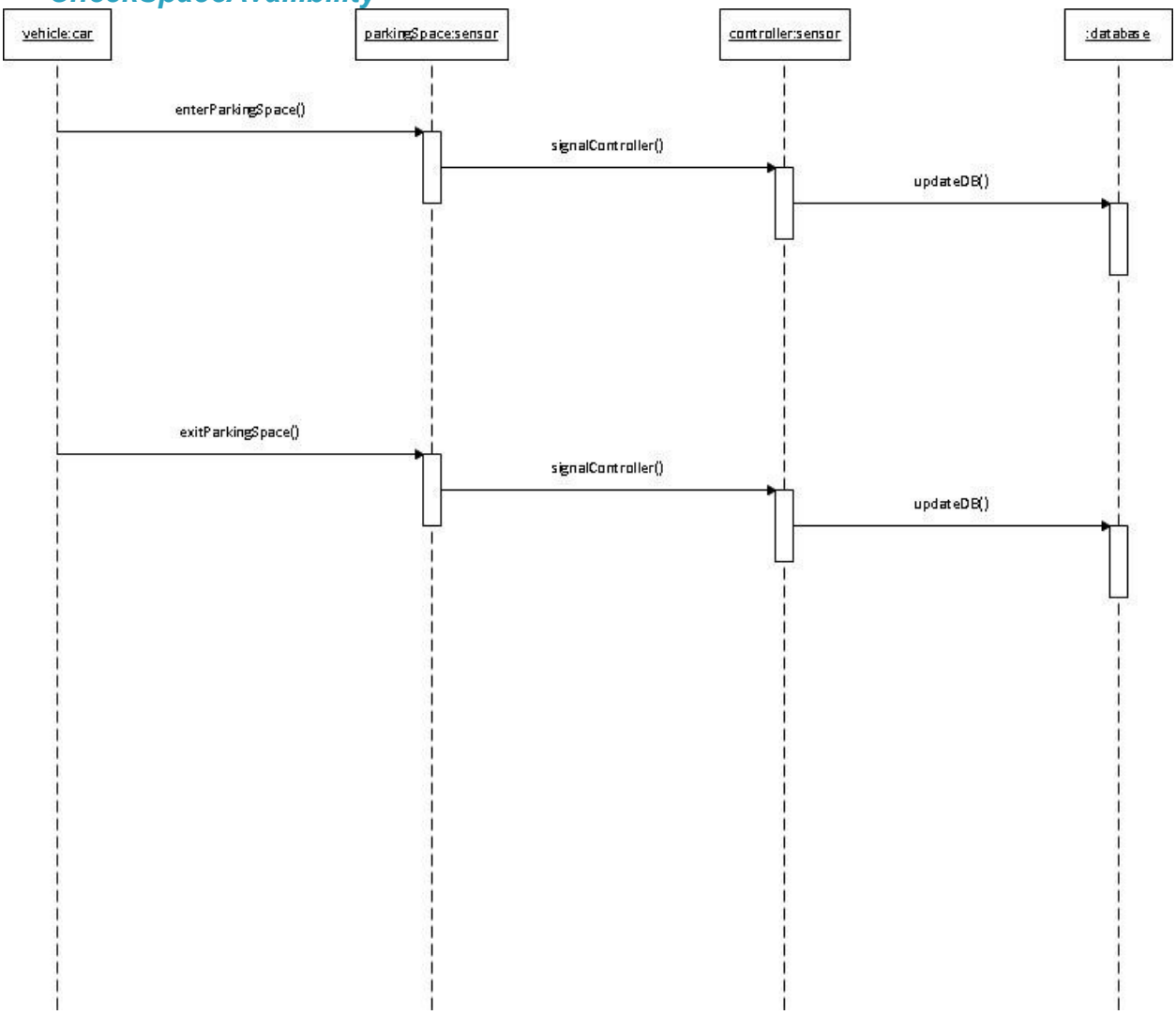


Within this use case, the High Cohesion principle was used to avoid objects having to take on tasks outside their expertise. Objects would simply focus on their designated duties and all message sending and external tasks are shifted over to the controller. This does not conform with the Low Coupling principle since the controller now has many different associations with all the objects within the use case.



The High Cohesion property was implemented within this use case, thereby shifting most of the computation related tasks towards the controller. All of the other objects within the use case simply need to work on their own tasks. In the situation where an object needs to invoke a response from another object, it will send a message to the controller and the controller will then invoke the needed method within the second object.

CheckSpaceAvailibility



Both the High Cohesion and Low Coupling principles were utilized within this use case. The sensor object is only tasked with sensing the arrive or departure of a vehicle. When such an event occurs, the sensor informs the controller, which then updates the database. Thus, each object is only involved with its own tasks. However, since this use case was fairly small, each object only had a few responsibilities. The number of associations between objects was greatly reduced.

Class Diagram and Interface Specification

Class Diagram

Accountant

- Dependencies
  - Admin -> Accountant; Accountant -> Accounts_Database;
  - Accountant -> Charge

- The Accountant class charges the credit cards the appropriate amount. Generally it takes the card number from the Accounts_Database, and charges for the duration recorded in Parking_Database. However, for unregistered walk-ins, Accountant has to take the card number from the Charge hardware module.

❖ Variables
- ➢ float rate; private
  - The usual rate for hourly parking.
- ➢ float overRate; private
  - The hourly rate for overstays.
- ➢ float specialRate; private
  - The hourly discount rate for special, low volume times.

❖ Functions
- ➢ float getRate(int rateType); package
  - Get function for the different rates.
- ➢ bool chargeAccount(long cardNum, float time, float overtime); private
  - Charges account of exiting car, either with the card number obtained from Charge for unregistered walk-ins, and from Accounts_Database for every other customer.
- ➢ void setRate(int rateType, float newRate); package
  - Set function for the different rates.

Accounts_Database

❖ Dependencies:
- ➢ Customer -> Accounts_Database; Reservation ->Accounts_Database;
- ➢ Hardware_Control -> Accounts Database;
- ➢ Admin -> Accounts_Database; Accountant -> Accounts_Database;
- ➢ Accounts_Database -> Car
  - Accounts_Database is the main log for all information about the accounts, including information and history. It is also the class that allows for updating of accounts.

❖ Variables
- ➢ int accountNum; package
  - Holds the value of the Account number for searching, login, and updating purposes. Used in Reservation, and Customer for customer indexing.
- ➢ string plateNum; private
  - Holds the value of the license plate; used for updating the account and updating the car list.
- ➢ long cardNum; package
  - Holds the value of the credit card number. Is used for account updating, and also for payment by the Accountant class.
- ➢ string name; package
  - Holds the value of the name of the customer. Used for account updating purposes, and also the Hardware Control class for display purposes.

❖ Functions
- ➢ bool login(int accountID); private
  - Logs into the account information
- ➢ void editAccount(int accountID); package
  - Edits account info. Specific values to edit are in function. Because this is webpage compatible, is a package function.

- addAccount2Car(string plateNum, int accountNum); package
  - Adds account number to car file. Includes car class, is therefore a package function.
- void write2file(); private
  - Writes info to file.
- void findAccount(int accountID); package
  - Locates and loads account info into variables. Used for account updates.

Admin

- Dependencies
  - Website -> Admin; Admin -> Accounts_Database; Admin ->Prediction;
  - Admin -> Accountant
    - Admin has only a few functions because most of the things it does resides in other classes that it accesses. Admin uses Accounts_Database the same way that Customer does with some more abilities. It also can adjust the Prediction variables in the Prediction class, and can adjust rates in the Accountant class.
- Variables
  - int adminOp; private
    - This variable expresses which operation is selected by the admin. Each value corresponds to a different operation.
- Functions
  - void detAdminOp(); private
  - Set function of adminOp.
  - getAdminOp(); package
  - Get function for adminOp

Camera

- Dependencies
  - Hardware_Controller -> Camera
    - This class manages the cameras. There is one camera in the elevator that reads the license plate of the entering vehicle. There is one camera at the exit that determines if the car exiting must pay at the Charge module (unregistered walk-ins) or if the car can exit (any registered vehicle).
- Variables
  - string plateNum; private
    - This variable is the license plate that the camera reads.
  - bool cameraNum; private
    - This variable determines what camera is being referenced. It is a boolean variable because there are only two cameras.
- Functions
  - string getPlateNum(bool camNum); package
    - Get function to return license plate number read by one of the cameras.
  - void setPlateNum(string newPlateNum, bool camNum); package
    - Set function for license plate number on one of the cameras.

Car

- Dependencies
  - Accounts -> Car

- Variables
  - string accountNums[]; package
    - Holds the accounts associated with each license plate. Used in Accounts_Database for updating.
- Functions
  - n/a

Charge

- Dependencies
  - Hardware_Controller -> Charge; Accountant -> Charge
    - This class manages the credit card machine that reads the credit card number at the exit for the unregistered walk-ins.
- Variables
  - long cardNum; private
  - This stores the credit card number.
- Functions
  - long getCardNum(); package
    - Get function for cardNum
  - void setCardNum(long newCardNum); package
    - Set function for cardNum

Customer

- Dependencies
  - Website -> Customer; Customer -> Reservation, Customer ->Account_Database
    - The customer passes values onto Reservation in the event of making, deleting, or editing a reservation. The customer passes values onto Account_Database when editing Account information. This is a basic class to provide a middle step before choosing an option on the website, an intermediary variable holder.
- Variables
  - int custOp; private
    - This holds the value of the operation that the customer is trying to achieve.
  - int accountNum; package
    - This holds the value of the account number inputted into the website. It is accessed by both Account_Database, and Reservation.
  - string plateNum; package
    - This holds the value of the license plate number, inputted into the website for reservation or account updating purposes. It is accessed by both Account_Database, and Reservation.
  - int start; package
    - Start time of the reservation; this is used by Reservation.
  - int end; package
    - End time of the reservation; this is used by Reservation.
- Functions
  - void detCustOp(); private
  - Sets custOp. Only used within Customer
  - string parseinfo(); package

- Parses the information inputted into the Website interface. May be used by website in addition to customer.

➢ getCustOp(); package
- Returns the value of custOp. Determines what class to go to next, Reservation or Accounts_Database

Door

❖ Dependencies
➢ Hardware_Controller -> Door
- This class manages the two doors on the elevator and the gate at the exit.

❖ Variables
➢ bool front; private; defaults to
- This is the door that allows the car into the elevator upon entrance.

➢ bool back; private
- This is the door that allows the car to pull onto the levels of the parking garage.

➢ bool gate; private
- This is the gate at the exit.

❖ Functions
➢ bool getDoorStatus(int doorNum); package
- Get function for the doors.

➢ void setDoorStatus(int doorNum, bool doorStatus); package
- Set function for the doors.

Elevator

❖ Dependencies:
➢ Hardware_Controller -> Elevator
- This class manages the elevator.

❖ Variables
➢ int atfloor; private
- Expresses the current floor that the elevator is on

❖ Functions
➢ void setFloor(int floorNum); package
- Set function for atfloor
- int getFloor(); package
  - Get function for atfloor

Front_Display

❖ Dependencies
➢ Hardware_Controller -> Front_Display
- This class manages the display that is in the entrance of the garage before the elevator that expresses if there is space for walk-ins.

❖ Variables
➢ bool walkinStatus; private
- This is the variable that says if there is space for walk-ins or not. It is a boolean variable because there are only two options.

❖ Functions
➢ bool getWalkInStatus(); package

- Get function for walkinStatus
- void setWalkInStatus(bool newStatus); package
  - Set function for walkinStatus
- void dispStatus(bool walkinStatus); package
  - Display function for the screen.

Garage_UI

- Dependencies
  - Hardware_Controller -> Garage_UI
    - This class manages the user interface inside the elevator.
- Variables
  - n/a
- Functions
  - void navigateUI(); package
    - This function navigates through the user interface inside the elevator, including the display and the keypad.

Hardware_Controller

- Dependencies
  - Hardware_Controller -> Accounts_Database;
  - Hardware_Controller -> Parking Database;
  - Hardware_Controller -> Camera;
  - Hardware_Controller -> Door; Hardware_Controller ->Elevator;
  - Hardware_Controller -> Garage_UI;
  - Hardware_Controller -> Front_Display; Hardware_Controller -> Sensor;
  - Hardware_Controller -> Charge
    - The Hardware_Controller coordinates all of the hardware components, and manages logistics for garage procedures.
- Variables
  - n/a
- Functions
  - void entrance(); private
    - This function coordinates all of the hardware from the moment that the car drives into the elevator all the way through until the elevator comes back down to the ground floor after transporting the car to the correct floor.
  - void exit(); private
    - This function coordinates all of the hardware when the car is exiting.
  - void monitor(); private
    - This function checks the hardware that reports the states of the garage, specially the spot sensors. This is the function that signals for overstaying, and signals to release spots that are free sooner than expected.

Parking_Database

- Dependencies
  - Reservation -> Parking_Database; Admin -> Parking_Database;
  - Accountant -> Parking_Database; Prediction -> Parking_Database;
  - Hardware_Controller -> Parking_Database

- Variables
  - string otPlate[]; private
    - A list of license plates that are overtime in their spots; private because all of the overtime computations are done in the Parking_Database class.
  - int otTime[]; private
    - A list of times that the corresponding license plates started in overtime.
- Functions
  - int findRes(string plateNum, int accountNum, int start); package
    - The main purpose of this function is to recognize whether the cars driving into the garage have a reservation. Uses values from Hardware_Controller.
  - void assignSpot(int accountNum, string plateNum, int start, int end); private
    - Manages spot scheduling by assigning spots to reservations and walk-ins. This is an important function and is for that reason private.
  - void freeSpot(int spotNum, int time); private
    - Frees up spots when a car leaves early, if there is a no-show, or if a reservation is cancelled or changed. This is also a very important function so it is private.
  - void adjustSpot(int spotNum, int time); private
    - This function is a utility function for assignSpot. It helps to reschedule a number of spots at one time for optimal scheduling, and for conflicts. It will keep a record of spots that are moved and optimally insert them where it is best.
  - void addOT(int spotNum, int otstart); package
    - This is the first step for overtime recognition. It lists the offending vehicles in a queue, and the corresponding times that overtime started in another queue. Uses information from Hardware_Controller.
  - void concludeOT(int spotNum, int endot); package
    - This is called when the offending vehicle is exiting. It takes the time of exit, from Hardware_Controller, and calculates how much over the vehicle was.
  - void viewStats(); package
    - This allows the Admin to see parking statistics for the garage.

Prediction

- Dependencies
  - Admin -> Prediction; Prediction -> Parking Database
    - The Prediction class is the module that accounts for the predictions that drive the overbooking in the garage. It allows admin to set variables, and computes them and passes them to the Parking_Database.
- Variables
  - int predictVar[]; private
    - This is an array of the different prediction variables. Because we have get and set functions for this array, the array is private.
- Functions
  - int getPredictVar(int varNum); package
    - Get function for predictVar[].
  - void setPredictVar(int varNum, int varVal); package
    - Set function for predictVar[].
  - int calcPredict(); package

- This function calculates the overbooking variable using the prediction variables. It forwards this value to Parking_Database, to allow it to schedule effectively.

Reservations

- Dependencies
  - Customer -> Reservation; Reservation -> Parking_Database;
  - Reservation -> Accounts_Database;
    - Reservation takes the information inputted into the website and forwards it to the scheduling database (Parking_Database), and the log database (Accounts_Database).
- Variables
  - n/a
- Functions
  - void makeRes(int accountNum, string plateNum, int start, int end); private
    - compiles information to input to the Parking_Database class to check for availability. Only used in Reservation class.
  - void delRes(int accountNum, int start); private
    - This inputs the needed information to delete a reservation as requested by a user. This passes information along to the Parking_Database class, and adjusts the account info in Accounts_Database. This is only used in Reservation.
  - void editRes(int accountNum, int start); private
    - Allows the user to edit the reservation; first passes the info on to Parking_Database to check for availability, then updates Accounts_Database with the new reservation information.

Sensor

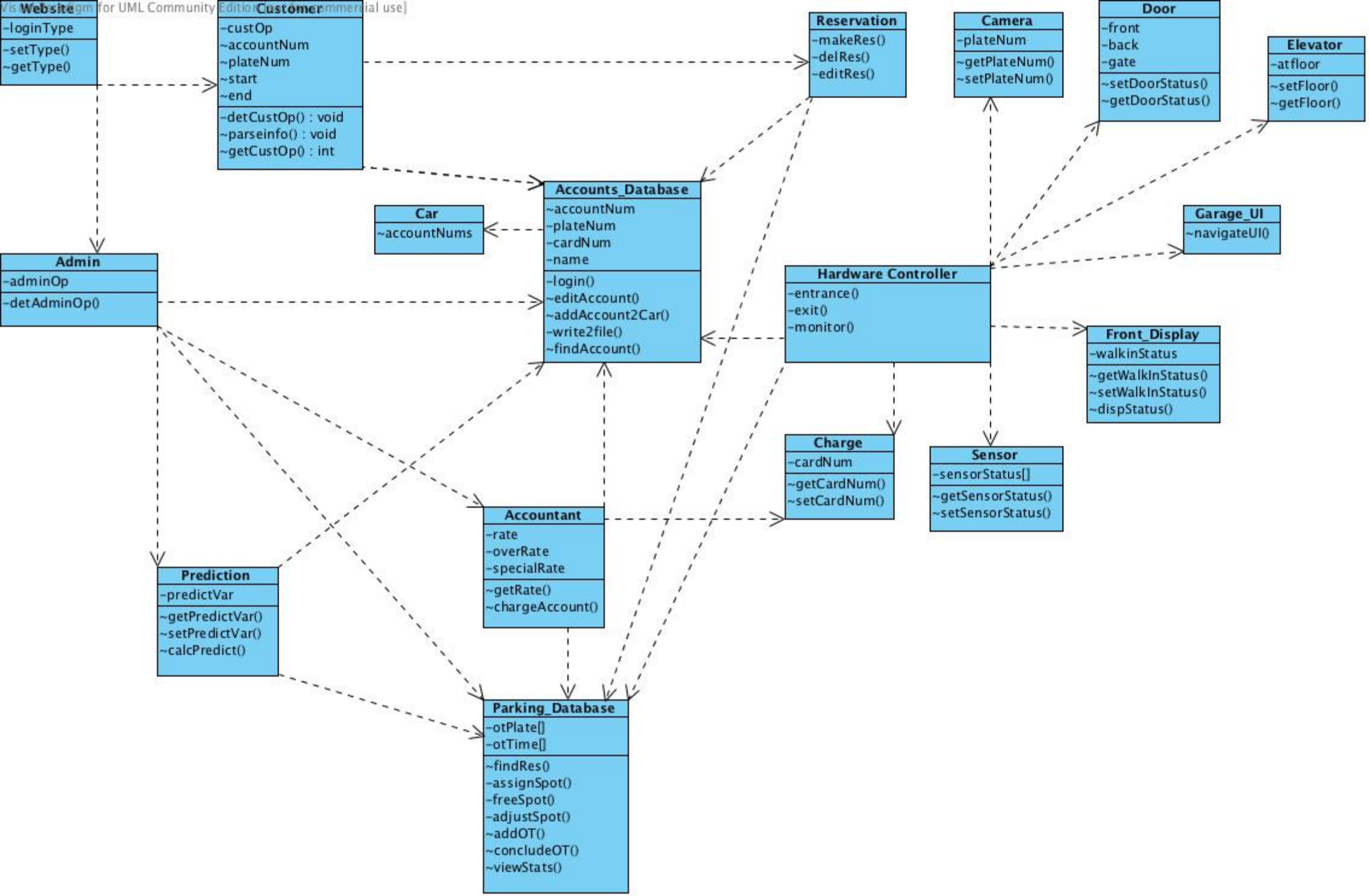
- Dependencies
  - Hardware_Controller -> Sensor
    - This class manages the sensors that are in each parking spot that report if the spot is vacant or occupied.
- Variables
  - bool sensorStatus[]; private
    - This array tells the state of each sensor. Each spot has a corresponding spot in the array.
- Functions
  - bool getSensorStatus(int sensornum); package
    - Get function for the sensors.
  - void setSensorStatus(int sensornum, bool sensorStatus); package
    - Set function for the sensors.

Website

- Dependencies
  - Website -> Customer; Website -> Admin
    - The website class passes values onto both the Customer and Admin classes. It is the initial User Interface for both the customer and administrators.
- Variables
  - bool loginType = 0; private

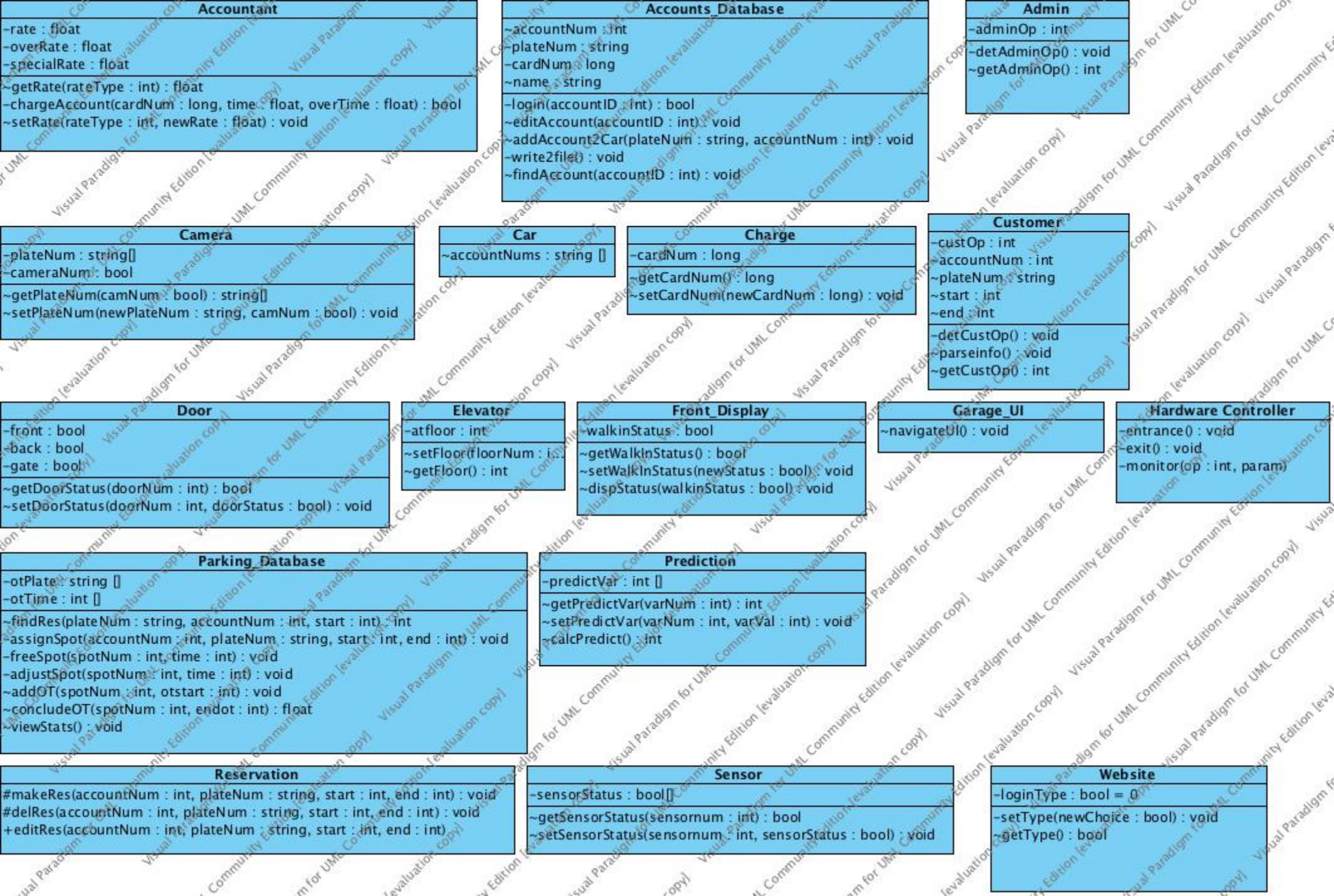
- This variable is to determine what kind of login the user will be attempting, administrative or customer. The variable is boolean because there are only two types of login types, so this saves space. The default setting is 0, or customer because that will be the most used login of the two. This variable will only be changed and viewed with functions within the class. Therefore it is private.

❖ Functions
- ➢ void setType(bool newChoice); private
  - This function sets the variable loginType. This function will only be used within the class so it is set as private. The argument into the function will provide the new value for loginType.
- ➢ bool getType(); package
  - This returns the value of loginType. This is visible to the package because other classes such as Admin and Customer will use it.



Data Types and Operation Signatures

On the next page is the diagram showing the Data Types and Operation Signatures. Please excuse the watermark.


Accountant
-rate : float
-overRate : float
-specialRate : float
~getRate(rateType : int) : float
-chargeAccount(cardNum : long, time : float, overTime : float) : bool
~setRate(rateType : int, newRate : float) : void
Accounts_Database
~accountNum : int
~plateNum : string
~cardNum : long
~name : string
-login(accountID : int) : bool
~editAccount(accountID : int) : void
~addAccount2Car(plateNum : string, accountNum : int) : void
-write2file() : void
~findAccount(accountID : int) : void
Admin
-adminOp : int
-detAdminOp() : void
~getAdminOp() : int
Camera
-plateNum : string[]
-cameraNum : bool
~getPlateNum(camNum : bool) : string[]
~setPlateNum(newPlateNum : string, camNum : bool) : void
Car
~accountNums : string []
Charge
-cardNum : long
~getCardNum() : long
~setCardNum(newCardNum : long) : void
Customer
-custOp : int
~accountNum : int
~plateNum : string
~start : int
~end : int
-detCustOp() : void
~parseinfo() : void
~getCustOp() : int
Door
-front : bool
-back : bool
-gate : bool
~getDoorStatus(doorNum : int) : bool
~setDoorStatus(doorNum : int, doorStatus : bool) : void
Elevator
-atfloor : int
~setFloor(floorNum : i...
~getFloor() : int
Front_Display
-walkinStatus : bool
~getWalkInStatus() : bool
~setWalkInStatus(newStatus : bool) : void
~dispStatus(walkinStatus : bool) : void
Garage_UI
~navigateUI() : void
Hardware Controller
-entrance() : void
-exit() : void
-monitor(op : int, param)
Parking_Database
-otPlate : string []
-otTime : int []
~findRes(plateNum : string, accountNum : int, start : int) : int
~assignSpot(accountNum : int, plateNum : string, start : int, end : int) : void
-freeSpot(spotNum : int, time : int) : void
-adjustSpot(spotNum : int, time : int) : void
~addOT(spotNum : int, otstart : int) : void
~concludeOT(spotNum : int, endot : int) : float
~viewStats() : void
Prediction
-predictVar : int []
~getPredictVar(varNum : int) : int
~setPredictVar(varNum : int, varVal : int) : void
~calcPredict() : int
Reservation
#makeRes(accountNum : int, plateNum : string, start : int, end : int) : void
#delRes(accountNum : int, plateNum : string, start : int, end : int) : void
+editRes(accountNum : int, plateNum : string, start : int, end : int)
Sensor
-sensorStatus : bool[]
~getSensorStatus(sensornum : int) : bool
~setSensorStatus(sensornum : int, sensorStatus : bool) : void
Website
-loginType : bool = 0
~setType(newChoice : bool) : void
~getType() : bool

System Architecture and System Design

Architectural Styles

Event-Driven Architecture

When looking at the system as a whole, it is easy to see that everything is based on events. Everything from making a reservation, and updating an account, to the entire process of parking and exiting the garage, to the utility functions that provide the framework and information needed to make the system run smoothly are all driven by events. The event emitters are customers and to a smaller extent administration. Everything that occurs in the system occurs because a customer takes some action, be it making a reservation online, or leaving the garage after having been parked. The nature of the project and the way in which it is meant to work guaranteed that our system would follow this style the closest.

Front End – Back End

Another style that permeates through the structure of this program is Front End – Back End. Clearly the databases are the back end of the program. They store all the information and make the decisions that drive the program. But the rest of the program is put into place to transfer information presented to the system (website, hardware) into language that the databases can understand and use.

Database-Centric Architecture

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

Pipeline

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

Client-Server Model

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

Identifying Subsystems

The three main packages are Administration, Users, and Simulation. This is meant to go along with the Use Cases so that the reason of using a package is nearly inherent to the package itself.

The User has limited access to other packages, but these packages will allow the user to accomplish his tasks.

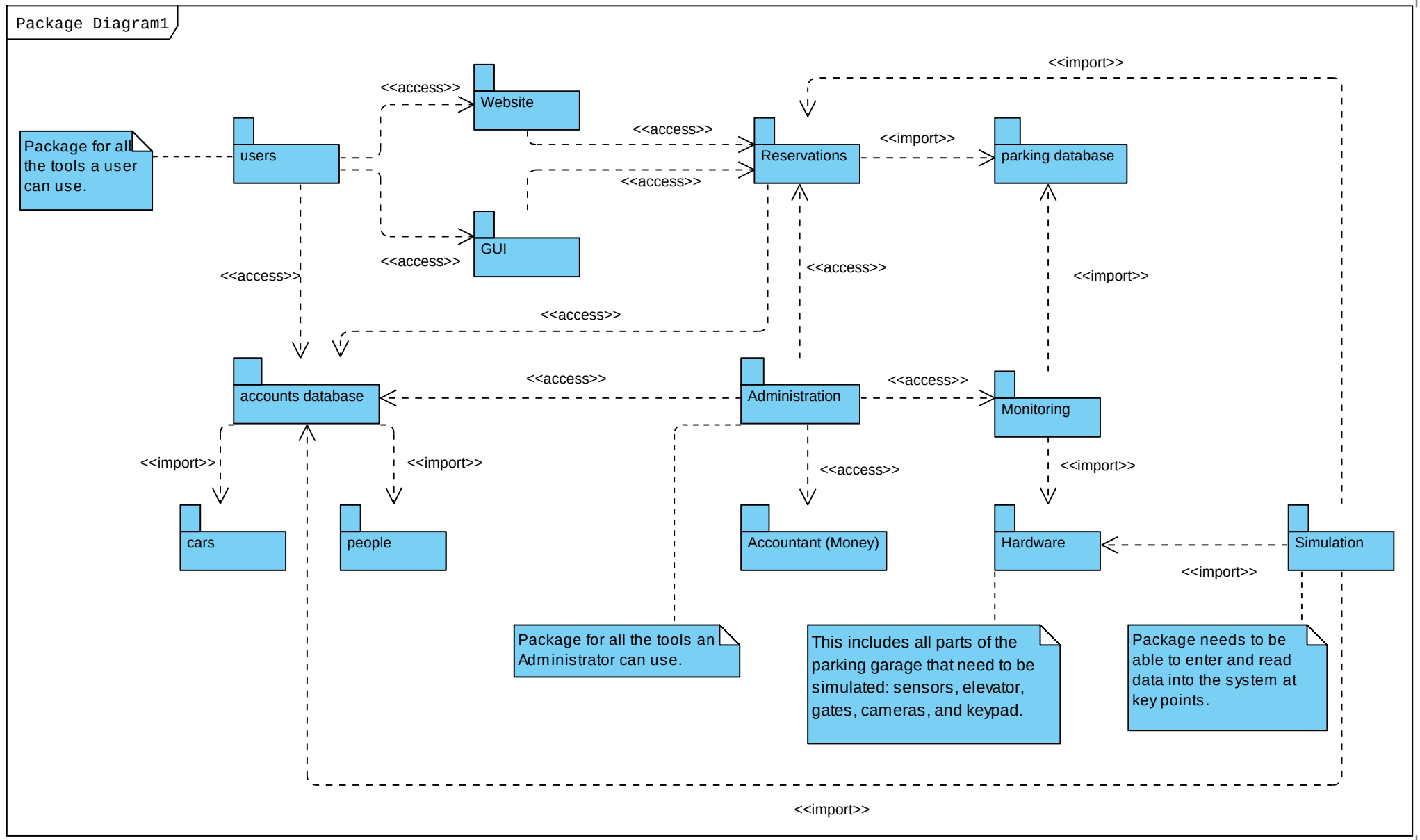
The Administration can view the important packages related to the parking garage. The Administrator has limited access to the data storage files but can view the required information for operation of the parking garage.

The final main package is the Simulation package. This package does not have any packages unique to itself and it should not. Its purpose is to simulate cars and users using the parking garage. Due to its function, the design choice of the Simulation package is to be able to <<import>> information from important databases and allow it to edit these values to make it seem like the system is being simulated. For a real parking garage, the Simulation Package would not be necessary.

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

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

UML Package Diagram



Brian Goodacre

Mapping Subsystems to Hardware

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

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

Persistent Data Storage

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

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

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

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

The first maintains reservation and account information per customer.

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

Customer and Cars Database Columns:

- Customer ID
- Customer Name
- License Plates of Cars
- Current Balance
- Upcoming Reservation 1 Arrival Time
- Upcoming Reservation 1 Departure Time
- Upcoming Reservation 2 Arrival Time
- Upcoming Reservation 2 Departure Time
- Upcoming Reservation 3 Arrival Time
- Upcoming Reservation 3 Departure Time
- Permanent Reservation
- Last X Number of Reservation (Where X is TBD)

Parking Space Database Columns

- Parking Space ID
- Parking Space Name
- Current Occupant of Car ID Occupant
- Current Occupant of Customer ID
- Customer ID at March 15, 2011 1:00pm
- Customer ID at March 15, 2011 1:15pm
- Customer ID at March 15, 2011 1:30pm
- Customer ID at March 15, 2011 1:45pm
- Etc…

Network Protocol

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

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

Global Control Flow

Execution Orderness:

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

Time Dependency

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

Concurrency

For the web database, if two users want to access the same information at the same time, we would need multiple threads for that. (i.e. a husband and wife, both logon on to edit a license plate, on two different accounts.) For this we just need a simple lock around sections where data is being edited and then read, so wrong values are not used. Otherwise, only one customer can park in any spot at one time, so each sensor is single-threaded. Only one customer can be in the elevator and use the keypad, so that can be single-threaded. Also, only one customer can leave at a time, so that camera is single-threaded. Finally, the camera that reads license plates is also single-threaded since it only reads one at a time. The controller itself must multi-threaded since it needs to send requests from multiple sensors and cameras to the main system. For this, the requests will need to be put a queue, since the controller can only deal with one at a time. We need to make sure there is not starvation for one of the input devices.

Hardware Requirements

- Touch screen display
  - minimum 800x600 pixels. The touch screen display will be the main way of the customer communicating with the system and entering information into the database.
- Hard drive
  - At least 10 GB of space. This will be the local hard drive for the garage where the local, cached information is stored. It'll hold the information about the parking spots and the customers.
- Network Connection

- A 1Mbps connection to support website and database transfers. A fast connection will allow the online customer database to interact with the system at the garage to keep track of reservations and which customers showed up.

- RAM
  - Minimum 2 GB of RAM to process the information from the touch screen. Without a good amount of RAM, the touch screen system UI will be slow and customers will be stuck in the UI menus waiting for them to load
- CPU
  - 2GHz processor to process the times and customer information efficiently. A slow processor will again make menu navigation a headache. Fast speed and efficiency will keep the customers happy and make things move faster.
- Sensors
  - IR sensors to detect whether or not a car is in a parking spot.
- Camera
  - 5 Megapixel Resolution for optimal reading of license plates
- Elevator buttons
  - To dictate which floor you are taking your car to.

Algorithms and Data Structures

Algorithms

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

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

Where:

P_T = Total # of parking spaces within garage

R_C = Confirmed reservations

R_G = Guaranteed reservations

N = No-show factor for reservations

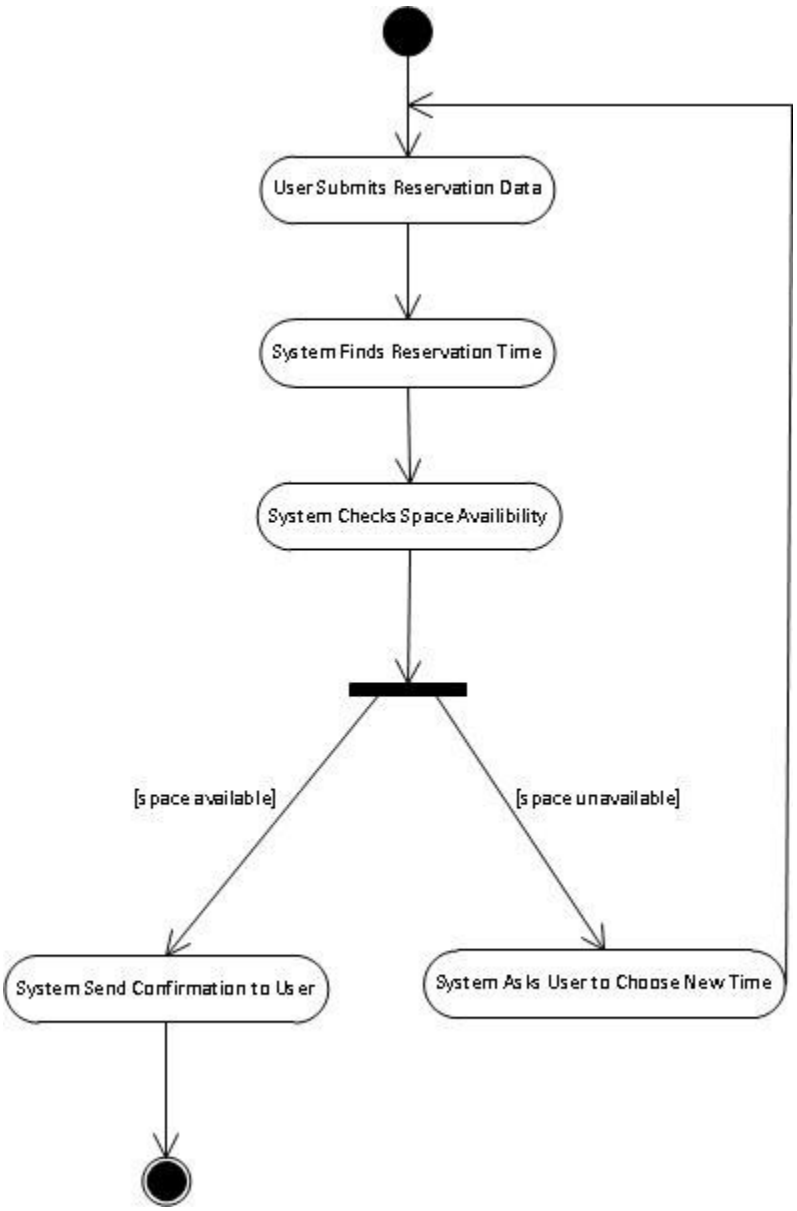
O_P = predicted overstays

U_P = predicted understays

W_P = predicted walk-ins

P_F = # of parking spaces that are currently free

The confirmed reservations account for one time reservations made by registered customers and the guaranteed reservations account for contract reservations made by registered customers. The no-show factor is based on historical data gathered during parking garage operation and it represented the percentage of customers that actually show up to claim their reserved parking space. The overstay count is predicted based on the percentage of cars within the garage that overstayed their parking duration in the past and the understay count is predicted based on the percentage of cars that understayed their parking duration in the past. The walk-in count is predicted based on the percentage of cars within the garage which are simply walk-in customers who have not made a reservation. Thus, this algorithm relies heavily on the ability to collect, store, and analyze the parking garage data.



Data Structures

Customer – The Customer class has the following data associated with it:

- Primary
  - int: Registration ID. This int will be the unique identifier when talking about customers.
- Secondary
  - String: First Name
  - String: Last Name
  - String: Street Address
  - String: City
  - String: State
  - int: Zip code
  - int: Credit Card number
  - int: Phone Number
  - String: email address
  - String[]: License plates. This needs to be dynamically allocated that automatically doubles it size when required. Since most users should have anywhere between 1 and 4 license plate, it will at most waste one String of allocated memory.

Summary of Customer Data

Most of the data here is self-explanatory. We chose to do an array for the multiple cars that will be used since most customers will only have one car associated with them. Also, it is easier to implement and faster to search the array then a linked list, especially if you know the index number. Many of the integer values can probably be casted as smaller sized variable types once we decide on final details. (i.e. length of ID number)

Car – The Car class had the following data associated with it:

- Primary
  - String: License plate. This will be the unique identifier of a car.
- Secondary
  - int[]: Registration IDs. A list of all the customers by registration ID associated with this Car. We will also use a dynamically allocated array here, since cars will only have more than 1 to 4 users if it is a company car, and if this is a popular option, we can add a company ID number to customer classes, so cars can belong to many people.

Summary of Car Data

The license plate can be any alphanumeric character, so a String is the easiest implementation. We chose to implement arrays here for multiple customers, since most cars will be associated with one or two customer, so a dynamically doubling array will waste no space in these two scenarios. It is also faster and easier to search than a LinkedList. Once again, int may change depending on length of registration ID required.

Storing Mass Quantities of Data in Memory

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

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

Other Data

All other permanent storage of data will be done with a database, which will be read when required.

User Interface Design and Implementation

The following description is for the User Interface navigation for the keypad on the elevator:

As a customer drives up to the elevator, a splash screen is displayed (Figure 1) as a place holder until the license plate is read.	 Figure 1
[**OPTIONAL**] After this a confirmation screen appears with the image of the license plate (Figure 2) that the camera captured. Above this image is the license plate string that the system recognized from the image. The customer can confirm this by clicking yes, or clicking no. If the customer clicks yes they will be brought to one of two screens.	 Figure 2

If only one customer is associated with the registered car, then they are automatically welcomed (Figure 3), and their next reservation is listed on the screen if there are any. It also gives the customer the option of editing this reservation. (i.e. to extend it). The customer will then confirm it is he or she.

Figure 3

The second option is that multiple customers are associated with the car, if that is the case then a list of all the customers associated with the car is displayed. (Figure 4) The customer can then select their name from the list, or click the button to say they are not listed. If they select one of the names, they are brought to a similar screen as Figure 3 with their current reservation, and the option to edit this reservation, or to continue parking.

Figure 4

… If the user clicks no in Figure 2, or the license plate is not recognized, or if the customer clicks the "I'm not here" button in Figure 4, the customer is brought to the screen where they can type in their registration identification number (RIN). (Figure 5) If they type in their correct registration number, they will be brought to their own welcome screen where their current reservation is listed. If they type it incorrectly, they will be given two more tries to get it correct before they are asked to leave. They may choose to skip this, if they do not have one. They are now considered a walk-in customer.

Figure 5

If the customer is a walk-in or a registered customer who wants to edit their reservation, they are brought to Figure 6. They can choose the time they want to now stay using the drop-down menus provided. Once they hit ok, they are brought to the thank you splash screen (Note this Figure is just a placeholder)

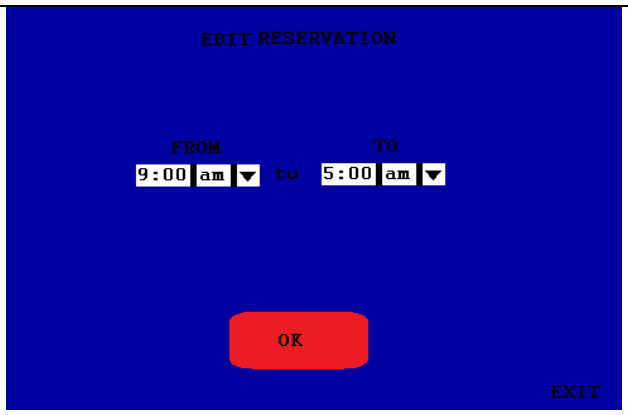


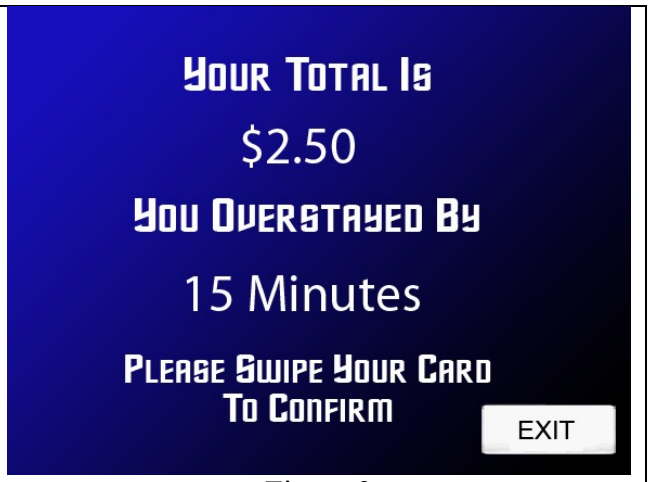
Figure 6

Once the customer confirms their final reservation, they are greeted with a final splash screen (Figure 7) telling them to drive forward.

Figure 7

This is the splash screen (Figure 8) that the customers will see upon exit. Once the camera picks up on the license plate of the leaving car, it displays the next screen. (Figure 9)

Figure 8

The screen displays how much the customer will be charged. It also displayed the amount they overstayed, so any extra charged is explained. If they are a registered customer in the system, they will be automatically charged, and they won't need to slide their credit card. However, if it is an unregistered customer leaving from a walk-in, they will need to slide their credit card. After the customer pays they are shown the splash screen again, thanking them for parking.	 Figure 9

Progress Report and Plan of Work

Progress Report

We have a very basic database currently working. The java program currently has two files associated with it: customers.txt and cars.txt. The first file list all the customers on each line individually, with the registration number first. After that follows all the user information, contact information, and then all the cars associated with the customer by license plate, all separated by a delimiter. The second file lists the cars on each line individually, with the license plate first, and then all the customers associated with the car, listed by registration number, separated by a delimiter.

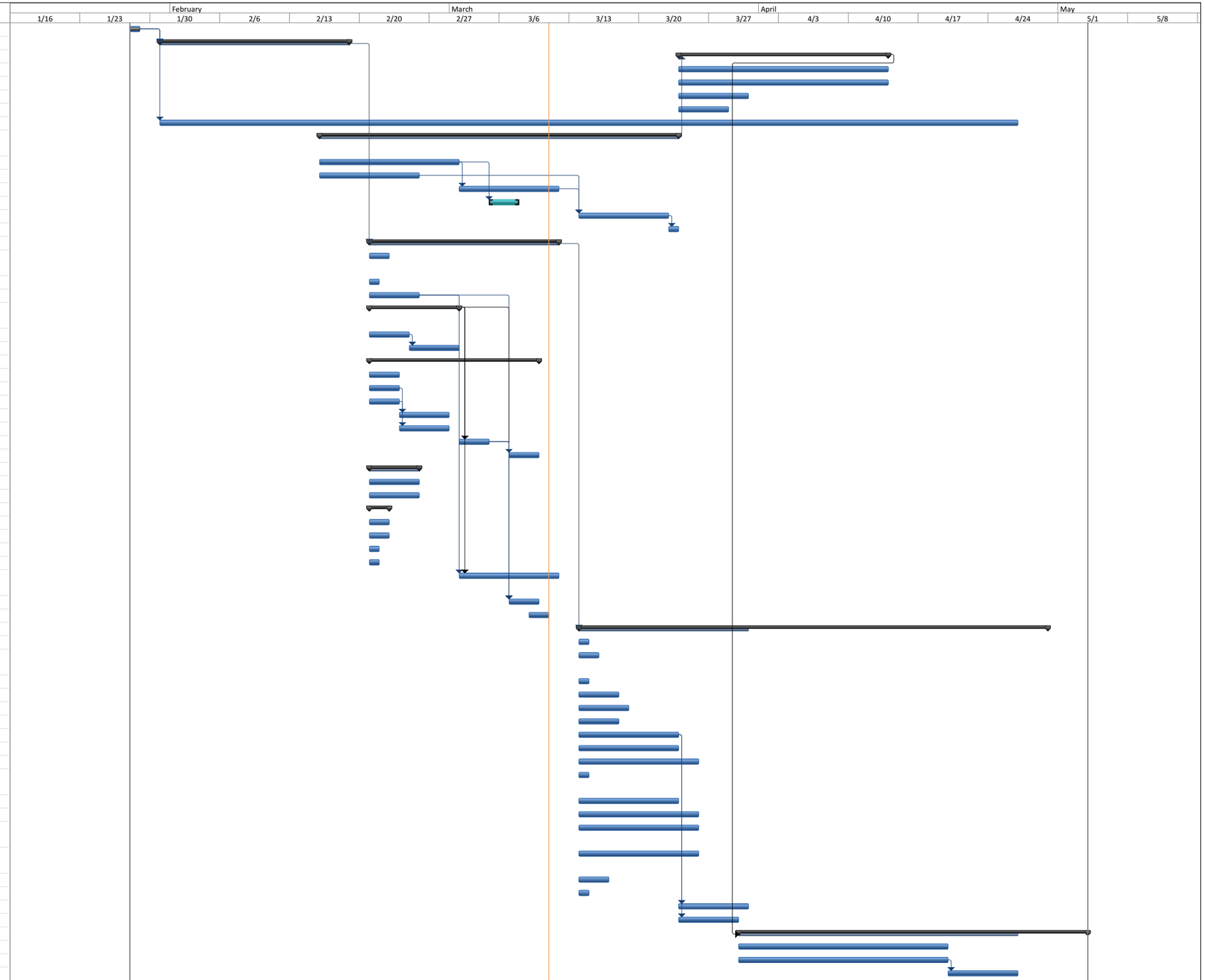
The program when it starts up parses all this information and loads it into memory. Right now this is small scale, less than ten users, but we want to see if it's realistic to load a large number of information at startup, and use this information in memory, so the program runs fast, and allows for ease of programming. All the customers and cars are put in their own Linked List, that can be searched.

After all variables are initialized, the user has several options, they can view their information, by looking at the license plate. This models the camera sending a license plate to the database, and fetching the user name. If that doesn't work, aka the user wants to skip it, the user can enter their registration number, and that will fetch their information via that variable. If the user does not have a registration number, they can choose to skip this step, and they will be asked to input their first name and last name, as well as the license plate of the primary car they want to be associated with. They will then be assigned a registration number, and their information will be displayed for them. This is analogous to registering via the website. All new information is written to text files for permanent storage.

The next task to be tackled is to implement a reservation module. We need to allow user to make a reservation, view their reservation, and edit their reservation. We also need to figure out how to represent this data in the database. Along with making reservations, we need to make parking spot classes, that can get take reservations, and a system that oversees these operations. Another system being worked on is the elevator user interface that the customer sees when they drive up to the elevator.

Plan of Work

ID		Task Mode	Task Name	Duration	Start	Finish	Predecessors
1			Proposal	1 day	Fri 1/28/11	Fri 1/28/11	
2			**First Report**	**15 days**	**Mon 1/31/11**	**Fri 2/18/11**	**1**
59			**First Demo**	**15 days**	**Thu 3/24/11**	**Wed 4/13/11**	**26**
60			Product Brochure	15 days	Thu 3/24/11	Wed 4/13/11	
61			Demo Format	15 days	Thu 3/24/11	Wed 4/13/11	
63			Demo Schedule	5 days	Thu 3/24/11	Wed 3/30/11	
62			Create Database	3 days	Thu 3/24/11	Mon 3/28/11	
87			Electronic Project Archive	62 days?	Mon 1/31/11	Tue 4/26/11	1
26			**First Level Programming of the Assignment**	**26 days**	**Wed 2/16/11**	**Wed 3/23/11**	
28			Determine how to begin coding	10 days	Wed 2/16/11	Tue 3/1/11	
31			Work with partners	8 days	Wed 2/16/11	Fri 2/25/11	
29			Code first parts	8 days	Wed 3/2/11	Fri 3/11/11	28
27			Research Data Structures	2 days	Sat 3/5/11	Mon 3/7/11	28
32			Debug other's code	7 days	Mon 3/14/11	Tue 3/22/11	29,31
30			Start working on next part	1 day	Wed 3/23/11	Wed 3/23/11	32
33			**Second Report - Design**	**15 days**	**Mon 2/21/11**	**Fri 3/11/11**	**2**
34			Cover Page and Individual Contributions Breakdown	2 days	Mon 2/21/11	Tue 2/22/11	
35			Table of Contents	1 day	Mon 2/21/11	Mon 2/21/11	
36			Interaction Diagrams	5 days	Mon 2/21/11	Fri 2/25/11	
37			**Class Diagram and Interface Specification**	**7 days**	**Mon 2/21/11**	**Tue 3/1/11**	
38			Class Diagram	4 days	Mon 2/21/11	Thu 2/24/11	
39			Data Types and Operation Signatures	3 days	Fri 2/25/11	Tue 3/1/11	38
40			**System Architecture and System Design**	**13 days**	**Mon 2/21/11**	**Wed 3/9/11**	
41			Architectural Styles	3 days	Mon 2/21/11	Wed 2/23/11	
44			Persisten Data Storage	3 days	Mon 2/21/11	Wed 2/23/11	
45			Network Protocol	3 days	Mon 2/21/11	Wed 2/23/11	
46			Global Control Flow	3 days	Thu 2/24/11	Mon 2/28/11	45
47			Hardware Requirements	3 days	Thu 2/24/11	Mon 2/28/11	44
42			Identifying Subsystems	3 days	Wed 3/2/11	Fri 3/4/11	37
43			Mapping Subsystems to Hardware	3 days	Mon 3/7/11	Wed 3/9/11	42
48			**Algorithms and Data Structures**	**5 days**	**Mon 2/21/11**	**Fri 2/25/11**	
49			Algorithms	5 days	Mon 2/21/11	Fri 2/25/11	
50			Data Structures	5 days	Mon 2/21/11	Fri 2/25/11	
52			**Progress Report and Plan of Work**	**2 days**	**Mon 2/21/11**	**Tue 2/22/11**	
53			Progress Report	2 days	Mon 2/21/11	Tue 2/22/11	
54			Plan of Work	2 days	Mon 2/21/11	Tue 2/22/11	
55			Breakdown of Responsibilities	1 day	Mon 2/21/11	Mon 2/21/11	
56			References	1 day	Mon 2/21/11	Mon 2/21/11	
51			User Interface Design and Implementation	8 days	Wed 3/2/11	Fri 3/11/11	37,36
57			review uml diagrams	3 days	Mon 3/7/11	Wed 3/9/11	42,37,36
58			review written works	2 days	Wed 3/9/11	Thu 3/10/11	
64			**Third Report**	**35 days**	**Mon 3/14/11**	**Fri 4/29/11**	**33**
65			split up parts	1 day	Mon 3/14/11	Mon 3/14/11	
66			Cover Page and Individual Contributions Breakdown	2 days	Mon 3/14/11	Tue 3/15/11	
67			Table of Contents	1 day	Mon 3/14/11	Mon 3/14/11	
68			Summary of Changes	4 days	Mon 3/14/11	Thu 3/17/11	
69			Customer Statement of Requirements	5 days	Mon 3/14/11	Fri 3/18/11	
70			Glossary of Terms	4 days	Mon 3/14/11	Thu 3/17/11	
71			Functional Requirements	8 days	Mon 3/14/11	Wed 3/23/11	
74			Domain Analysis	8 days	Mon 3/14/11	Wed 3/23/11	
75			Interaction Diagrams	10 days	Mon 3/14/11	Fri 3/25/11	
76			Class Diagrams and Interface Specification	1 day?	Mon 3/14/11	Mon 3/14/11	
77			System Architecture and System Design	8 days	Mon 3/14/11	Wed 3/23/11	
78			Algorithms and Data Structures	10 days	Mon 3/14/11	Fri 3/25/11	
79			User Interface Desgin and Implementation	10 days	Mon 3/14/11	Fri 3/25/11	
80			History of Work & Current Status of Implementation	10 days	Mon 3/14/11	Fri 3/25/11	
81			Conclusions and Future Work	3 days	Mon 3/14/11	Wed 3/16/11	
82			References	1 day	Mon 3/14/11	Mon 3/14/11	
72			Nonfunctional Requirements	5 days	Thu 3/24/11	Wed 3/30/11	71
73			Effort Estimation using Use Case Points	4 days	Thu 3/24/11	Tue 3/29/11	71
83			**Second Demo**	**25 days**	**Wed 3/30/11**	**Tue 5/3/11**	**59**
84			Product Brochure	15 days	Wed 3/30/11	Tue 4/19/11	
85			Demo Format	15 days	Wed 3/30/11	Tue 4/19/11	
86			Demo Schedule	5 days	Wed 4/20/11	Tue 4/26/11	85



Project: Plan of Work
Date: Fri 3/11/11

Task | Split | Milestone | Summary | Project Summary | External Tasks | External Milestone | Inactive Task | Inactive Milestone | Inactive Summary | Manual Task | Duration-only | Manual Summary Rollup | Manual Summary | Start-only | Finish-only | Deadline | Progress

Breakdown of Responsibilities

James Jacob

is currently responsible for developing, coding, and testing the following classes:

- The Hardware_Controller class
- The Door class
- The Elevator class
- The Garage_UI class
- The Front_Display class

Brian Goodacre

is currently responsible for developing, coding, and testing the following classes:

- The Prediction class
- The Parking_Database class
- The Accountant class

Richard Romanowski

is currently responsible for developing, coding, and testing the following classes:

- The Customer class
- The Car class
- The Account_Database class

Matthew Rodriguez

is currently responsible for developing, coding, and testing the following classes:

- The Website class
- The Admin class

Richard Roman

is currently responsible for developing, coding, and testing the following classes:

- The Sensor class
- The Camera class
- The Charge class

Groups

Richard Romanowski, Brian Goodacre, and James Jacob will coordinate the integration of all of the classes into the full system. In addition, Brian Goodacre and James Jacob will perform the testing of the integrated system

Bibliography

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

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

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

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

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

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

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