Parking Garage Automation:
Reserve Your Spot!

Software Engineering – 14:332:452

Group #3:
Bartosz Agas, Christoper Tran, Marvin Germar, Michael Genderen, Justin Levatino, Tarun Katikaneni

URL(s):
www.reserve-your-spot.com (Actual Website)
www.sites.google.com/site/ece452parkinggarage (Project Tracking Site)

Submission Date:
March 2, 2012

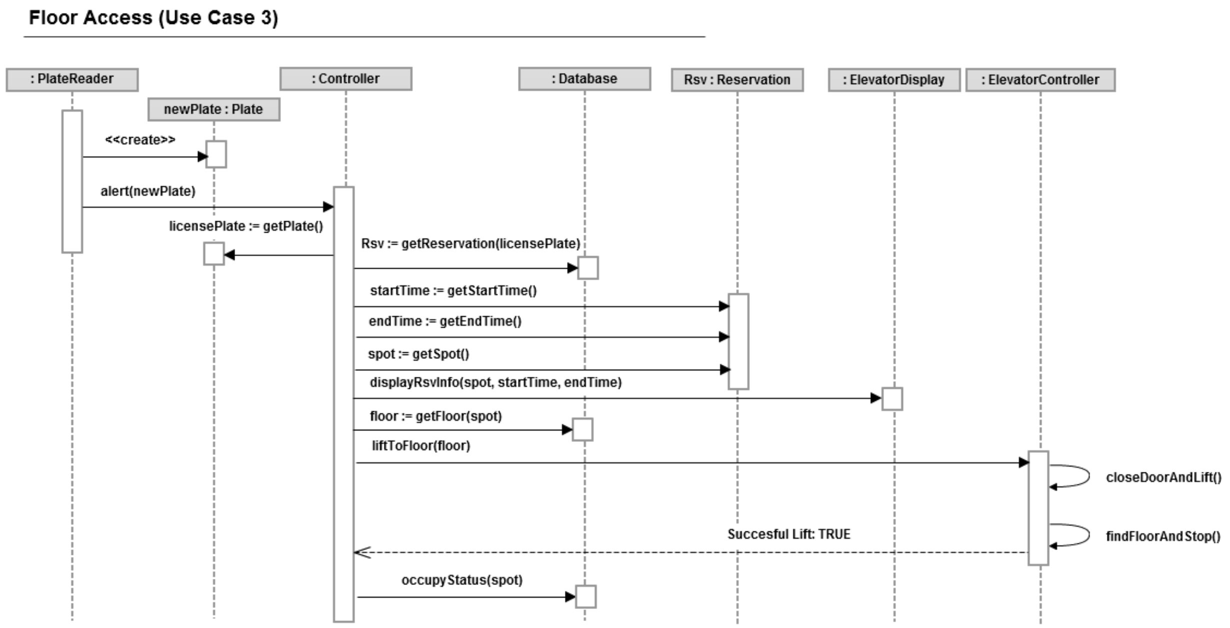
Effort Breakdown

All team members contributed equally

Table of Contents

Interaction Diagrams

Floor Access – Use Case 3



PlateReader

This object is a camera located in the elevator and it is responsible for extracting the information off the actual vehicle. It does so by creating a new plate in the system. The camera shows high cohesion since it is a physical device that handles one task, scanning the license plate off cars that enter the elevator.

Controller

The controller is the most essential portion of this interaction as it interacts with all other objects within it. When a vehicle enters the elevator, the controller will get the license plate number extracted by the camera and query the database for the cars reservation. Once it finds the reservation is gets the start point, end point, and spot location the user requested when they made the reservation. It then displays all this information on the screen in order to remind the user of their time limits and location within the garage. It then proceeds to lift the elevator to the proper level and marking that spot as occupied in the database.

Database

The database will be responsible for maintaining the information about the customer and the reservation. It means the requirements for the high cohesion principle because it takes in a variety of inputs and then processes the data to return the information.

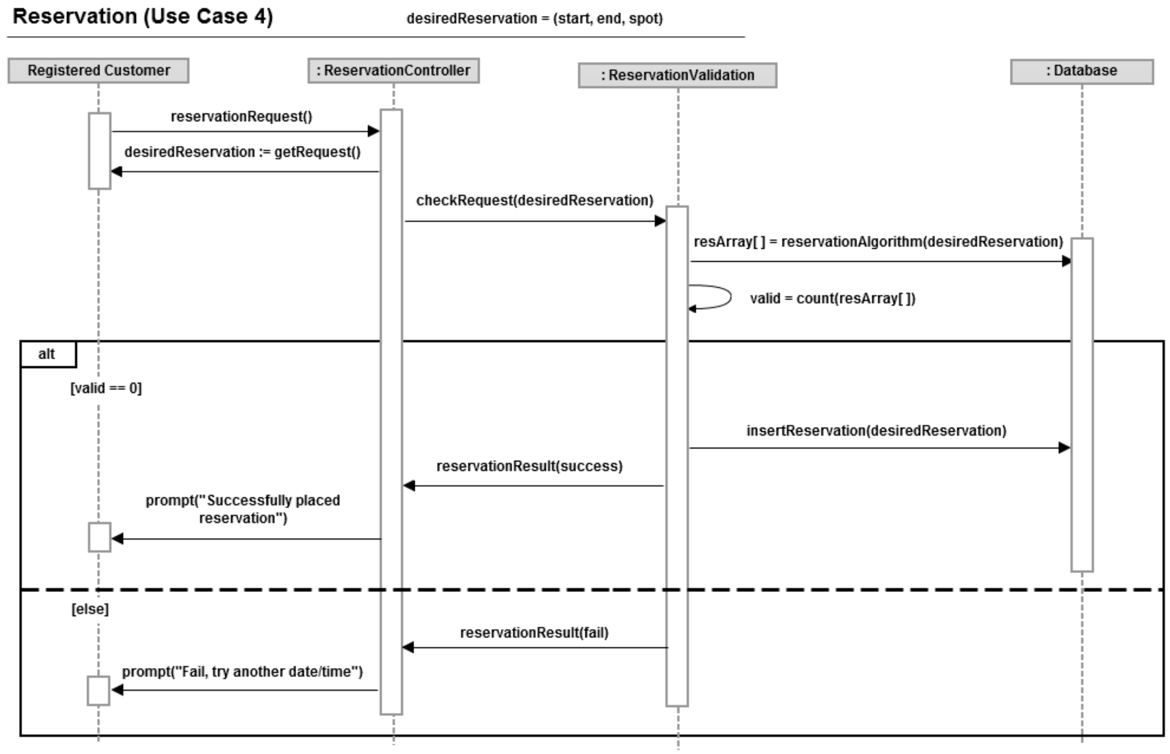
Reservation

A reservation object is created when the controller can successfully pull the reservation from the database. It is used to extract the reservation information in order to display the information to the user as well as send the elevator to the proper floor.

ElevatorDisplay and ElevatorController

The elevator display is an LCD panel located in the elevator that will display important information to the user regarding their reservation. The elevator controller is responsible for taking the elevator to the proper level as instructed by the main controller. Both these objects meet the specifications for low coupling because they don't take on too many tasks at once. Each object simply talks to the next as shown in the diagram above.

Reservation – Use Case 4


Reservation (Use Case 4)
desiredReservation = (start, end, spot)
Registered Customer
: ReservationController
: ReservationValidation
: Database
reservationRequest()
desiredReservation := getRequest()
checkRequest(desiredReservation)
resArray[ ] = reservationAlgorithm(desiredReservation)
valid = count(resArray[ ])
alt
[valid == 0]
insertReservation(desiredReservation)
reservationResult(success)
prompt("Successfully placed reservation")
[else]
reservationResult(fail)
prompt("Fail, try another date/time")

***Note:** When a customer places their reservation, they are asked to enter a start point (time and date), end point (time and date), and a spot within the garage. This is represented as one entry, for simplicity, in the above diagram as *desiredReservation*.

Registered Customer

The registered customer will be the one initiating this interaction by placing a reservation request. After the request is processed, the customer will be informed whether that reservation is valid or not.

ReservationController

When alerted that a request for a reservation has been put in, it will prompt the user for the reservation information. Once this data is received, it will forward the request to see if there are any conflicts with existing reservations. Upon receiving the result, the controller will return to the customer whether this transaction is valid or whether the user should attempt another reservation.

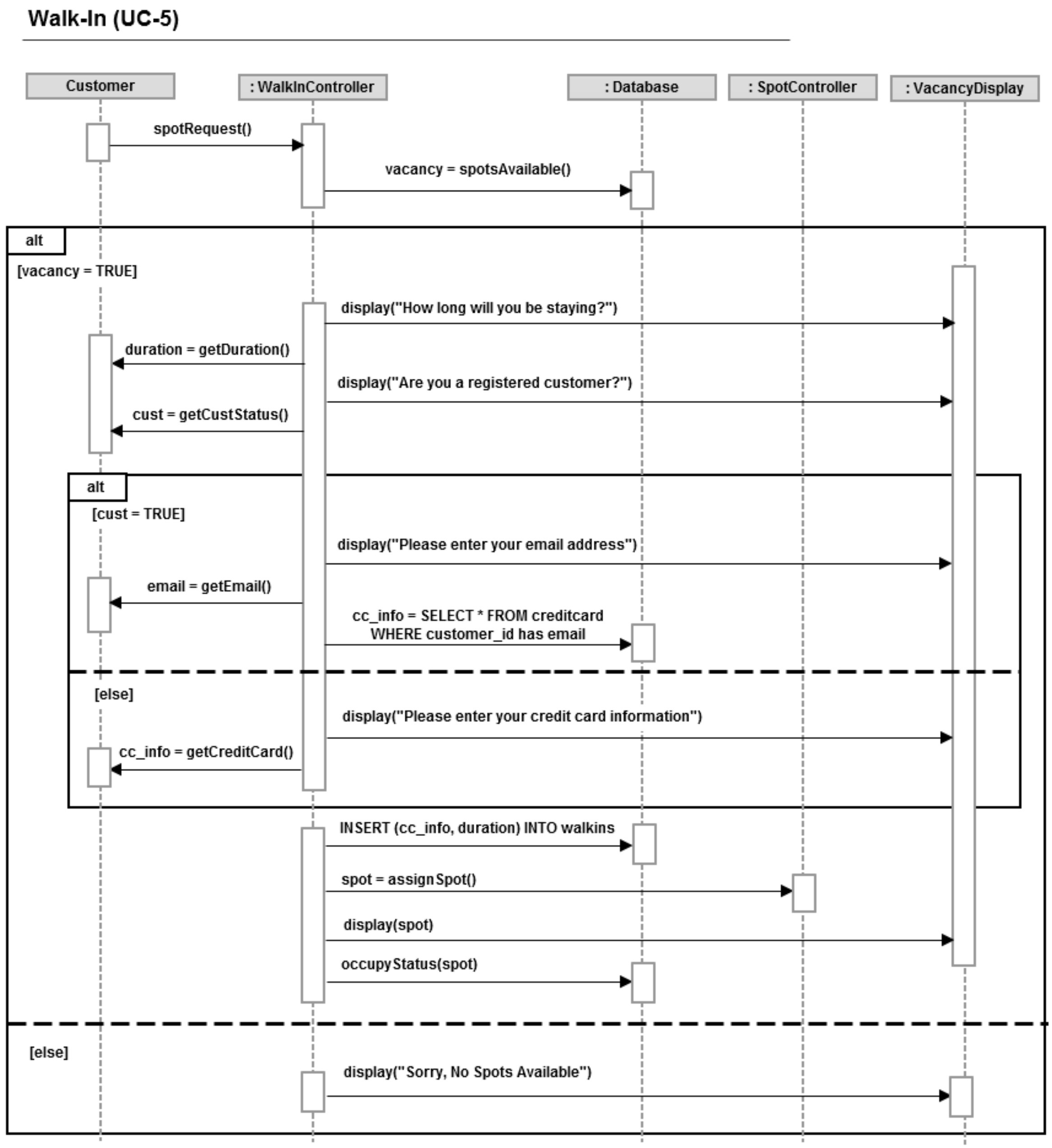
ReservationValidation

This will serve as both a connection to the database, responsible for passing messages to and from the database, and validating a reservation request by running it through an algorithm to validate it.

Database

The database will be responsible for maintaining the information about the customer and the reservation. It means the requirements for the high cohesion principle because it takes in a variety of inputs and then processes the data to return the information.

Walk-In – Use Case 5


Walk-In (UC-5)
Customer
: WalkInController
: Database
: SpotController
: VacancyDisplay
spotRequest()
vacancy = spotsAvailable()
alt
[vacancy = TRUE]
display("How long will you be staying?")
duration = getDuration()
display("Are you a registered customer?")
cust = getCustStatus()
alt
[cust = TRUE]
display("Please enter your email address")
email = getEmail()
cc_info = SELECT * FROM creditcard WHERE customer_id has email
[else]
display("Please enter your credit card information")
cc_info = getCreditCard()
INSERT (cc_info, duration) INTO walkins
spot = assignSpot()
display(spot)
occupyStatus(spot)
[else]
display("Sorry, No Spots Available")

Customer

The customer is the initiator in this transaction and responsible for entering the reservation duration and billing information accurately. If there are no spots available at the time the customer arrives to the parking garage, the interaction will terminate and inform the customer accordingly.

WalkInController

The controller is responsible for interacting with the customer via the vacancy display and the system to validate all requests. It follows the expert doer principle as it knows who and what should perform each task.

Database

The database will be responsible for maintaining the information about the customer and the reservation. It means the requirements for the high cohesion principle because it takes in a variety of inputs and then processes the data to return the information.

SpotController

This will be responsible for assigning the customer with the first available spot located in the ground level parking. If the interaction reaches this point, it has been confirmed that there are spots available and this will pull the first available spot and assign it to this customer.

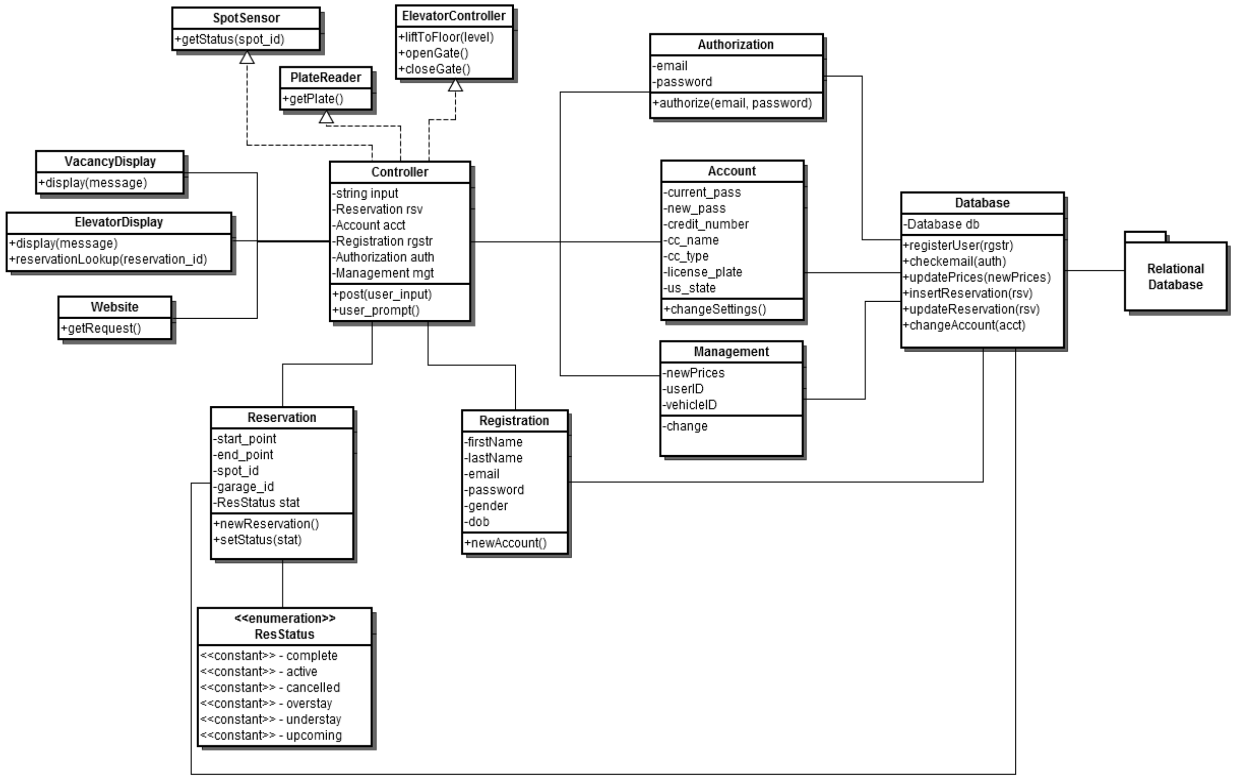
VacancyDisplay

This will be solely responsible for engagin the user to find out how long the user would like to stay as well as how to charge them for the parking. All users interact with this if they want to park on the ground level as a ‘walk-in customer.’

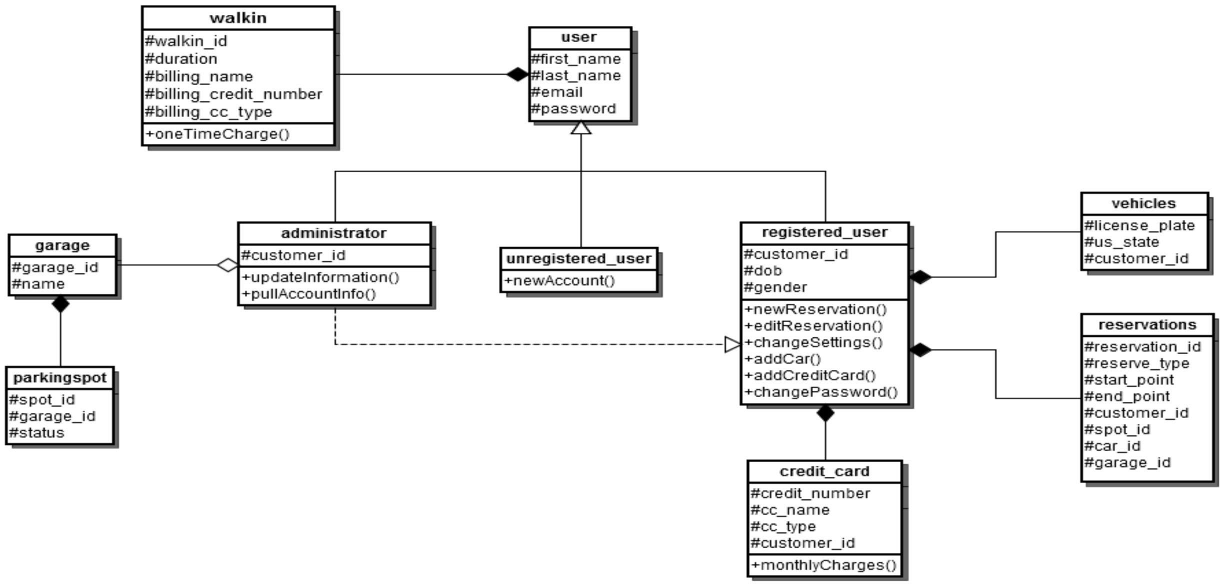
Class Diagram and Interface Specification

Class Diagram

Class Diagram - General



Relational Database



Data Types and Operation Signatures

All the classes above are broken down a little better below. Every class contains the attributes associated with it as well as the operations they carry out.

customer:

- Attributes:
  -int customer_id
  -string first_name
  -string last_name
  -string email
  -string password
  -date dob
  -enum gender(‘Male’, ‘Female’)
- Operation: (all operations attach customer to any created objects)
  +addCar() : creates a car and inserts into database
  +addCreditCard() : creates a credit card and inserts into database
  +newReservation() : creates a reservation and inserts into database

parkingspot:

- Attributes:
  -int spot_id
  -int garage_id
  -enum status(‘vacant’, occupied’, ‘reserved’)
- Operation:
  +updateStatus()

garage:

- Attributes:
  -int garage_id
  -string name
- Operation:
  +createGarage( )
  +insertSpot( )

creditcard:

- Attributes:
  -string credit_number
  -int customer_id

-string cc_name
-string cc_type

- Operation:
  +createCreditCard( )

car:

- Attributes:
  -car_id
  -license_plate
  -us_state
  -customer_id
- Operation:
  +createCar( )
  +removeCar( )

reservations:

- Attributes:
  -int reservation_i
  -time start_time
  -time end_time
  -date reserve_date
  -int customer_id
  -int car_id
  -int garage_id
  -int spot_id
  -enum reserve_type('active', 'canceled', 'overstay', 'understay', 'completed', 'upcoming')
- Operation:
  +createReservation( ) : creates a reservation record
  +cancelReservation( ) : voids the reservation placed
  +editReservation( ) : for upcoming reservations only, changes can be made

Traceability Matrix

All the classes seen above were derived from the traceability matrix. It is here that all the domain concepts were merged with the use cases in order to see how the system will function. Once this was set up, the classes began to form starting with user and from what point of view he/she would see the system. The views were dependent on the user being an administrator, unregistered, or registered user. All the essential equipment used throughout the garage also got involved as these pieces of equipment pulled essential information from the customer and/or garage in order to efficiently maintain the web service.

The remainder of the classes were focused around the database as it does hold the key pieces to upkeep all this information. These classes included creating credit cards, vehicles, and the garages along with their spots as well. They each have their own respective operations within them to insert, remove, and upkeep accurate data. This is reflected upon the traceability matrix as the database is involved with practically all the use cases created.

System Architecture and System Design

Architectural Styles

An architectural style provides a framework for a system which includes software components, its properties, and the relationships among them. The most useful style for the parking garage automation is the event-driven architecture. This software architecture pattern promotes the production, detection, consumption of, and reaction to events. One specific event is reserving a parking spot. It is the system's focus, and thus is the reason why event-driven architecture is most suitable.

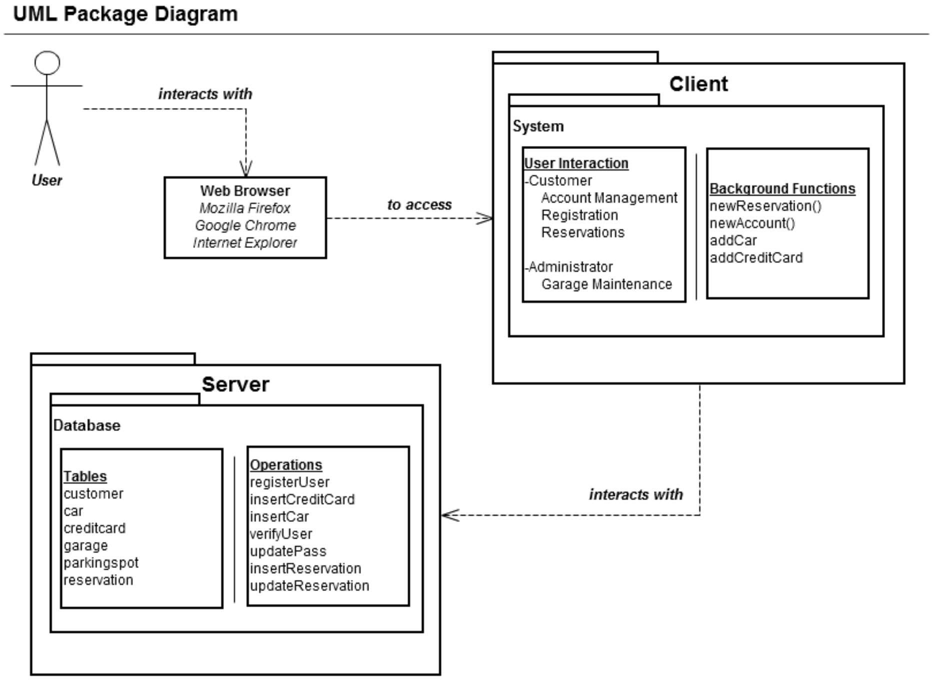
When a parking spot is reserved, it causes software components to change and others to react. The table below illustrates the system's cause and effect to an event.

Event	**Before**	**After**
Reserve	none	reserved
Cancel	reserved	cancel
Extend	reserve, extend	reserve, extend
Overstay	parked	overstay
Understay	parked	understay
Missed	reserved	missed
Completed	reserved, overstay, understay, missed	completed

Event-driven architecture is geared towards unpredictable and asynchronous environments. This is common when a customer interacts with the system. By using event-driven architecture, the parking garage automation sustains a stable and responsive system.

Identifying Subsystems

As this website will be an online service it will contain a client to interact with the user and a server to maintain a record of all the interactions and requests. This is accomplished by the user accessing the web client through a web browser user the http protocol. The client is comprised of HTML, CSS, and PHP and will interact with the server within those limits. User interaction will take place on the client side of the system and make function calls to the server in the background. Once in the server, the client will update the tables stored through the operations location within the database class.



Mapping Subsystems to Hardware

The model of the System calls for it to be ran on more than one computer. These computers can be broken into two categories; Web servers and client computers.

Web Servers

The web server will be used as the main database in which all of the sensors will store the data in. Majority of the source code will be run within the web servers. All client account information will be stored within the web servers.

In addition the web server will have a container that stores empty as well as occupied parking spots. The web server will keep track of all of the reservations and contracts. It will store a history of all of the reservations and overstays so the system can better predict occupancy.

The web server will store all of the prices and fees and issue them accordingly. It will have the ability to contact the client via email to send reminders; such as a reminder that a reservation is coming up and a reminder that a payment is due.

Client Computers

The client computers will access the web servers via a browser website that is running on html code. The client computers include the elevator display, vacancy display and client's personal computers.

The client's personal computers will access the web servers to set up an account, make a payment, make a reservation, create a contract and edit account information. There is an administrator mode that can be accessed when administrator accounts login. From administrator mode the user will be able to set prices and policies as well as manage delinquent accounts.

The elevator display and vacancy display will access the web servers to retrieve system status. This includes account information as well as vacant parking spots. It will access the web servers in order to display reservation information to the client.

Persistent Data Storage

MySQL was chosen as the database to maintain all the information acquired by the system. Seven tables, customer, creditcard, car, garage, parkingspot, walkin and reservation, will be used to hold all the relevant information pertaining to the user and their requests. The tables are seen below along will their respective attributes and detail.

DATABASE TABLES				
Field	**Type**	**Null**	**Extra**	**Links**
customer				
customer_id	int	no	auto increment	
first_name	varchar(45)	no		
last_name	varchar(45)	no		
email	varchar(45)	no	unique	
password	varchar(45)	no		
dob	date	no		
gender	enum('Male', 'Female')	no		
creditcard				
credit_number	varchar(45)	no		
customer_id	int	no		customer->customer_id
cc_name	varchar(45)	no		
cc_type	enum('Visa', 'Mastercard', 'American Express')	no		
car				
car_id	int	no	auto increment	
license_plate	varchar(45)	no		
us_state	varchar(3)	no		
customer_id	int	no		customer->customer_id
garage				
garage_id	int	no		
name	varchar(45)	no		
parkingspot				
spot_id	int	no		
status	enum('vacant', occupied', 'reserved')	no	default: 'Vacant'	
garage_id	int	no		garage->garage_id

walkin				
walkin_id	int	no	Auto increment	
billing_name	varchar(45)	no		
billing_credit_number	varchar(45)	no		
billing_cc_type	enum('Visa', 'Mastercard', 'American Express')	no		
reservation				
reservation_id	int	no	auto increment	
reserve_type	enum('active', 'canceled', 'overstay', 'understay', 'missed', 'completed', 'upcoming')	no		
start_point	datetime	no		
end_end	datetime	no		
spot_id	date	no		parkingspot->spot_id
customer_id	int	no		customer->customer_id
car_id	int	no		car->car_id
garage_id	int	no		garage->garage_id

Network Protocol

Our service will be offered as a website and it will be hosted off of a single server/machine, therefore, there will not be a need to utilize any other communication protocol other than HTTP. The system is still being built but it will be built in such a way that it can be adapted quickly; in the future, if the need for security arises, the system can be modified to accept other communication protocols. Also, as per the problem statement, the system is designed such that it communicates with our database, which stores all client names and information.

Global Control Flow

Execution Orderness:

Our service is two-fold; it is event-driven in that customers are required to enter in information regarding themselves and the vehicle that they will be parking in the garage. It is procedure-driven in that as customers approach the garage, the camera based license-plate recognition software checks each and every license plate for reservations; the floor sensors that act to verify if a parking spot is occupied or vacant is also procedure-driven.

Time Dependency:

Our system does employ timers; they serve an important role in that they measure the duration a customer has utilized the parking garage service. They are also used to measure when a customer has checked in to the garage and they also serve to inform the management of duration of the over-stay, if the customer over-stay's his reservation.

Concurrency:

No.

Hardware Requirements

Hard-drive: A hard-drive is needed for storing customer information. Assuming that customer information per customer costs about a Megabyte of memory and there are about 200 parking spots which would create a maximum of 71,200 customers a year (200 * 356), then the hard-drive should be at a minimum of 100,000 MB per year.

Elevator Display: The display in the elevator should be a minimum of 800x600 resolution since it is a little smaller and closer to the customer.

Vacancy Display: The display in the lobby should be bigger so more people can see it at once. This should be a minimum of 1280x720 resolution.

Network Bandwidth: Since the memory cost of the customer was estimated at 1MB, then a bandwidth of 50 KB per second should be the minimum bandwidth. This would mean that a customer would have to wait at the most 20 seconds for their memory to transfer.

Elevator Keypad: The elevator keypad should consist of only number buttons, an enter button, a backspace button, and a cancel button since it is only meant to put in reservation numbers.

Walk-In/Park Computer: A cheap computer should be available for walk in customers to make their on the spot confirmation.

Sensor: A light sensor or a ultrasonic sensor would both suffice to sense the vacancy of each parking spot while a pressure sensor is needed for the elevator.

Servers: There should be a minimum of one server to handle the garages reservations.

Cameras: The cameras do not have to be too expensive as long as they can differentiate between license plates within a 100 meter distance.

IPhone: For the customers that choose to have the app, they will need an Iphone.

Computer: To access the website, a customer will need a computer to create a reservation.

Central Computer: To run the whole garage in parallel, the garage will need a computer solely to run all of the garage programs.

Algorithms and Data Structures

Algorithms

Scheduling a Reservation

The complex algorithm used to check for possible reservations is more accurately described as a database query. Like an algorithm, the query follows a protocol by communicating with the database in a specific manner. The protocol that is followed consists of steps such as retrieving, inputting, and verifying user inputs with the database. When a user places a reservation via the website, they select when it will begin, when it will end, and which spot they would like to park in. Once this selection is made and submitted, the database query is put together to check to see if this reservation can be made.

This is accomplished by establishing a couple important ground rules with the database itself. For starters, the start and end points for a reservation will be stored into the reservation table with a DATETIME type. These entries are stored in the following format:

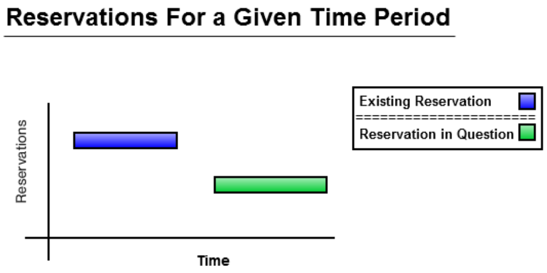
```
'YYYY-MM-DD HH:MM:SS'
```

This simplifies the process because the values in these fields can be compared directly via MySQL and don't need to go through any concatenation. The system begins by pulling all the reservations made for a particular spot, called *requested_spot*, via the following query:

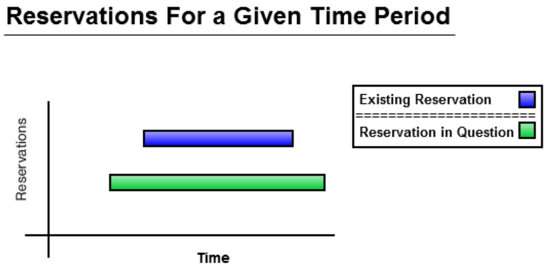
```
SELECT * FROM reservation WHERE spot_id=request_spot
```

With this query, we pull all the reservations in our database related to that one spot. It will serve the first half of our nested, complex query. The next step is to compare the start and end points themselves. For demonstration purposes, let's call the start point *newStart* and the end point *newEnd*. Again, these two points are of the datetime format. All time comparisons fall into one of five total categories:

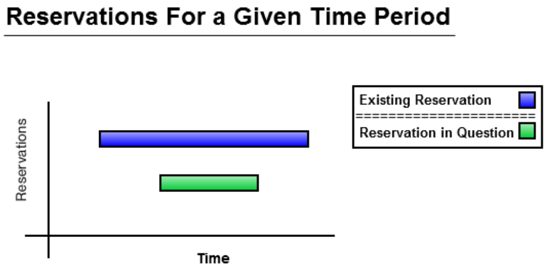
1. The best case scenario, when there is no reservation overlap and the requested reservation can be made.



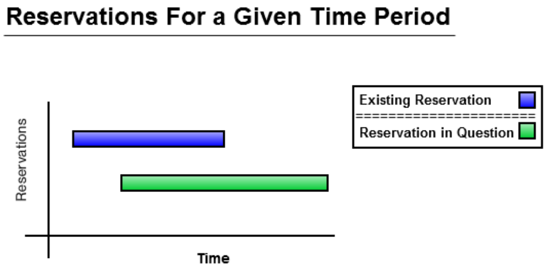
2. This next case occurs when a user wishes to place a reservation that starts before an existing reservation and ends after it. This overlap is an issue and the system will inform the user that it is not possible.



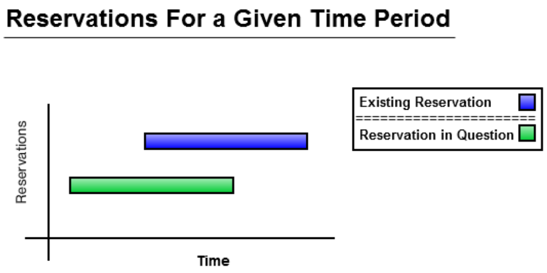
3. The reverse of the previous scenario arises when a user wishes to place a reservation that falls right in the middle of an existing reservation. Just like the previous case, this is an overlap and the system will inform the user it is not possible.



4. Another instance of an overlap will occur when a user attempts to place a reservation that starts right before an existing reservation is about to end. This is overlap is not acceptable and the system will inform the user.



5. Lastly, when a user wishes to place a reservation that will start right before an existing reservation is set to start. Once again, the system will inform the user this is not acceptable.



The following query is a generic example that will pull any of the four possible overlap cases:

```
mysql> select * from reservation R where (R.start>=newStart and
R.end<=newEnd) or (R.start<= newStart and R.end>= newEnd) or
(R.start>= newStart and R.start<= newEnd) or (R.end>= newStart
and R.end<= newEnd);
```

Once all this is put together and the final query is made, if there are any results, there is an overlap and the requested reservation cannot be made.

Data Structures

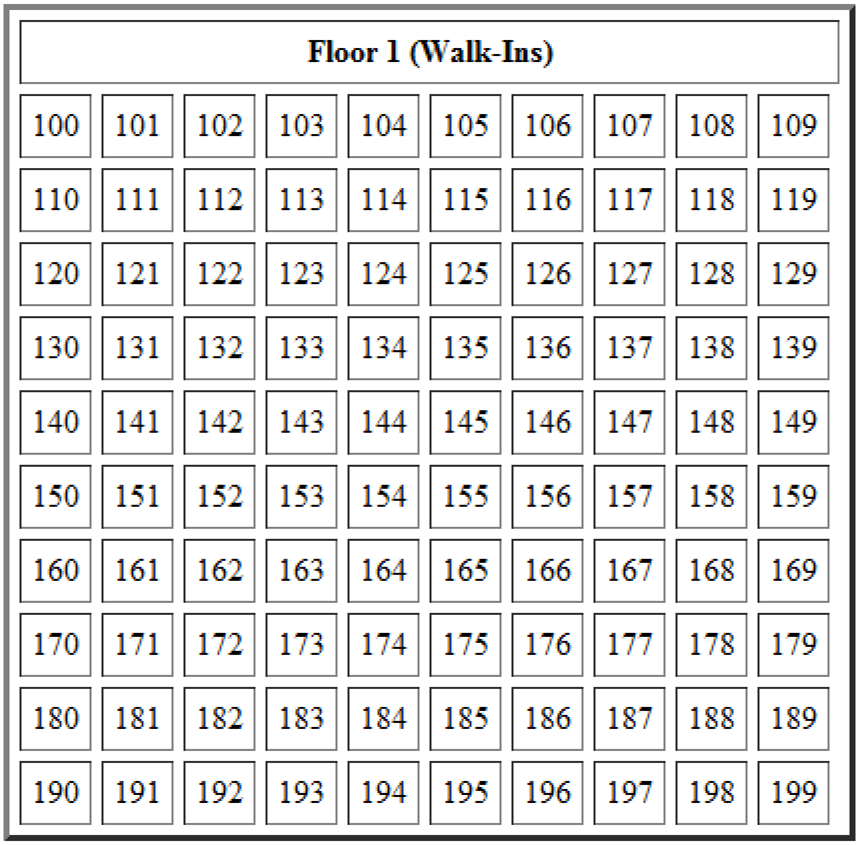
As mentioned earlier, all the information the system obtains will be stored and maintained in a database. MySQL will be used to make the necessary queries and modifications, as well as perform the algorithms described later in this section. It was chosen as it is the world's most used relational database management system.

As some database queries will pull multiple entries at once, an array data structure is used to store all these search query results in one place. It's really the only data structure that was implemented due to its simplicity and efficiency. Future changes to this system will include a map of the garage at the current time to give the user a better understanding of what is available. As this portion hasn't been implemented yet, any additional data structures can't be guaranteed just yet.

User Interface Design and Implementation

The following user interfaces are additions to the ones seen in report one. As of now, there haven't been any modifications submitted in the original report, but the following are to appear on the site in the near future. Each of the following interfaces give a brief idea of what the final product will look like as well as an explanation as to how it work or what it does.

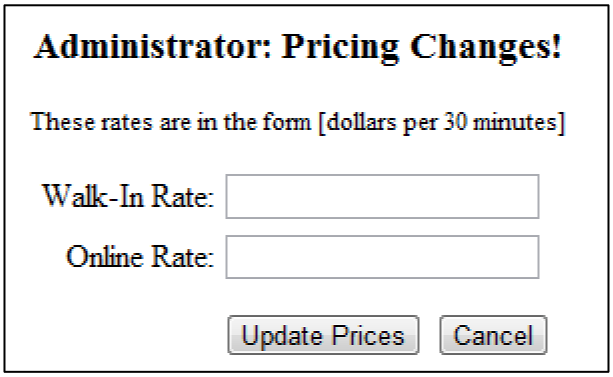
Map of the Garage:



Floor 1 (Walk-Ins)									
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

Figure x-1: This table is a mapping of what spots are currently available or taken. The table is still a work in progress because it needs to be color coded. Available spots will have a green background while taken spots will be red. Users will be able to see all four floors via the online interface. The first floor is for Walk-In customers while floors two, three, and four will be for those who reserved spots in advance.

Administration Menu:



Administrator: Pricing Changes!

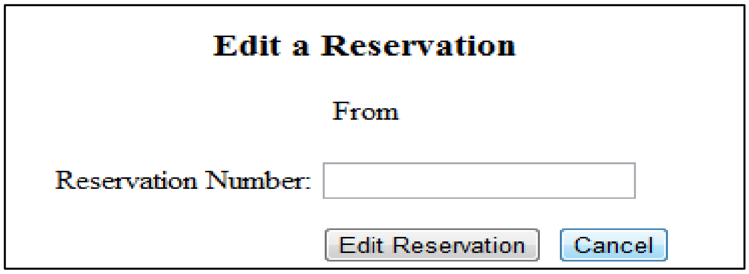
These rates are in the form [dollars per 30 minutes]

Walk-In Rate:

Online Rate:

Figure x-2: This figure shows the current administrator menu. An administrator will be able to access this page and change pricings as needed. More features will be added to this menu so that admins can have more control over pricing. For instance, admins will be able to change prices for holidays, weekends, and different times of the day.

Edit Reservation:

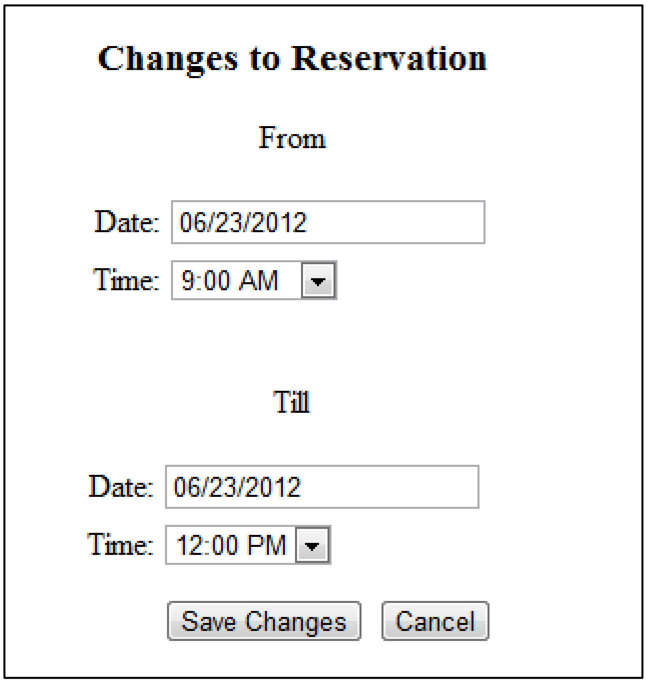


Edit a Reservation

From

Reservation Number:

(a)



Changes to Reservation

From

Date: 06/23/2012

Time: 9:00 AM

Till

Date: 06/23/2012

Time: 12:00 PM

(b)

Figure x-3: This figure shows what users will see when they want to edit an online reservation. (a) They first have to enter their reservation number. (b) After their reservation number has been confirmed, users will have to fill this page out in order to make changes to their reservation. An error message will let the user know if the newly requested time slot is not available.

Design of Tests

Unit Testing

The test cases for this system will be based of the generic class diagram, non-PHP specific, mentioned earlier. The five main classes in this system are the registration, reservation, management, authorization, and account classes. Unit testing will involve going through each of these classes and their modules, asserting them with different parameters.

The breakdown of the testing that will be performed or the individual class is as follows:

Registration

- Registering a new user with information that matches data of another existing user in the system. All users are uniquely identified by their email and this will be the essential factor in testing this.

Reservation

- The registration process can lead to a variety of situations and will therefore include testing try to expose any leaks. This will involve inserting any reservations that conflict with existing reservations, including the parking spot ID, start points, and end points. In addition to this, the design will also test that confirmed and guaranteed reservations do not conflict as they both pertain to the same spots, but are looked at with two different statuses. Lastly, any revisions made to upcoming reservations will also be tested to find conflicts as they arise.

Authorization

- Authorizing is an a crucial key that grants certain users, certain privileges and testing this class is key to making sure order is maintained in the system. As users attempt to identify themselves by either the Vacancy Display or through the website, their log on credentials will be passed through this class. Testing will involve asserting the modules with log on credentials that both match up and differentiate themselves from the ones in the database.

Account

- The account class is responsible for maintaining the information regarding the user and their account. Testing in this class will involve asserting a validation process that will take any requested changes or additions and look them over before they are carried out.

Management

- Just like the account class, management is the same principle except it involves pricing and regulation for the entire system. Therefore it will assert the modules in this class with parameters to validate all changes.

The goal for the design of tests is to cover the interactions a user has with the system covered from end to end. As the user can interact with the system in a variety of ways through a few channels, it is important to have coverage to account for all the scenarios, no matter what the situation.

Integration Testing

In order to keep things simple, the big bang integration approach will be used to carry out the testing design described above. The goal of the big bang method is to take all the individual unit test cases and combine them to form a complete or major part of the system. One type of big bang integration, known as Usage Model Testing, runs testing by carrying out user-like workloads in user-like integrated environments. The individual components are proved through this method by proving through the environment itself. The main focus will be to not avoid any smaller cases that may arise through the individual components.

This integration approach was chosen due to its simplicity and efficiency. Its simplicity stems from its capability to cover a significant amount of individual unit cases. One item that may be looked at for unit testing is the SimpleTest framework for PHP, very similar to JUnit. It is built around test classes that are written as extensions of base test classes that extend the modules found in the actual code. It works by asserting various values to methods a developer expects to be true. This will be covered in more detail in the final report as it is worked into the system.

Project Management and Plan of Work

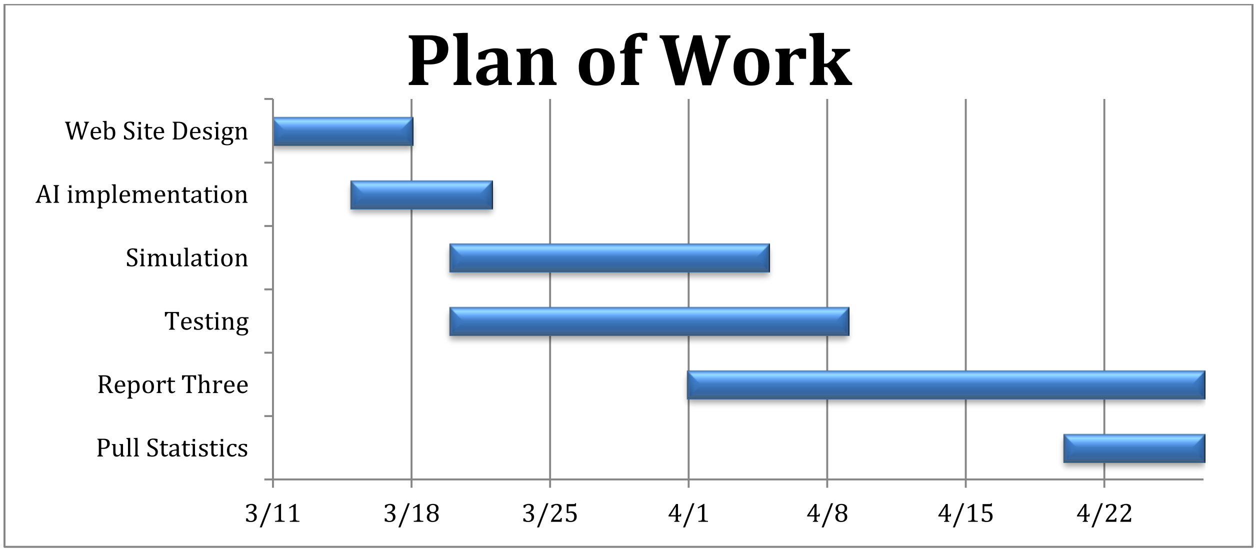
Merging the Contributions from Individual Team Members

When it comes to preparing a report in teams, many issues arise as different members will have their own way to go about it. In order to ensure consistency, once all the individual contributions were completed, all the members of this team got together and went over all aspects of the report. Once we were comfortable with what everyone had done, all the individual files were compiled by one or two members in order to ensure consistency through uniform formatting and appearance. From there, the final rough draft is distributed amongst the team for proofreading and any last minute recommendations or changes. It is after this step that the report is approved for submission.

Project Coordination and Progress Report

At this point, the website is up and running with several items still in the works. A user can go to the parking website and register their own account. In addition to this they can log onto their account through the home page and make changes to their account, credit card, and vehicle information. Lastly, the reservation page allows users to place their reservation by selecting a start and end date. Some items that are still being tackled are reservation conflicts, adaptation to drivers using their spot outside reservation time, and the page to view the garage and its availability. These modifications have been worked since the last report submission, but still haven't been fully implemented where they can be uploaded to the site for regular use.

Plan of Work



- Web Site Design
    - As mentioned before, there are still a few aspects of the site that need to be added/modified. The plan is to have it done at the end of the week in order to start preparing for the first demo on March 27, 2012.
- AI Implementation
    - In order to get a better feel as to how the site will function as a large quantity of users interact with it, artificial intelligence will be implemented to demonstrate. Once activated, the threads will pull all the available spots in the garage from the database and then place random reservations.
- Simulation
    - The simulation aspect of this project will be a hidden page. It will be strictly used to demonstrate how the artificial intelligence will interact with regular user actions. Its purpose is to activate the artificial intelligence and demonstrate how the 'View Garage' portion of the website will update itself to change the status of the individual spots.
- Report Three
    - As the final report, report three will have an exact explanation of how the final product will work and function. It will specify any changes made going back to the original proposal. It will demonstrate how the user will interact with the site, what's going on in the background, and what the management staff can do with it behind the scenes.
- Testing

    - Predicted testing methods will primarily include unit testing. Since the user will interact with the website for their needs, the plan is to work on each page individually in order to find any faults in the design and functionality. It will first begin with testing the site alone and then performing the same testing methods as the artificial intelligence is running in the background.
- Pulling Statistics
    - If time allows, this will be an additional page on the website. As of now, plans are to keep this strictly for the use of the parking garage management staff. It'll be a simple run down of how many clients over stay their reservation, have a credit card on file, park in the garage on a day-to-day basis, and much more. The purpose of this is to give the staff a better idea as to what type of people they are dealing with on a consistent basis and how to predict future ideas/improvements to the garage based on these numbers.

Breakdown of Responsibilities

- Bartosz Agas
    - Worked on Report 1, Web Site Design, Database Structuring, Report 2, Coding and Implementation, Simulation, Report 3
- Christopher Tran
    - Worked on Report 1, Web Site Design, Database Structuring, Report 2, Coding and Implementation, Simulation, Report 3
- Marvin Germar
    - Worked on Report 1, Web Site Design, Database Structuring, Report 2, Coding and Implementation, Simulation, Report 3
- Michael Van Genderen
    - Worked on Report 1, Web Site Design, Report 2, Coding and Implementation, Simulation, Report 3, Testing
- Justin Levatino
    - Worked on Report 1, Web Site Design, Report 2, Coding and Implementation, Simulation, Report 3, Testing
- Tarun Katikaneni
    - Worked on Report 1, Web Site Design, Report 2, Coding and Implementation, Simulation, Report 3, Testing

References

- Marsic, Ivan. *Software Engineering*. 2012.
- Bruegge, Bernd, and Allen H. Dutoit. *Object-oriented Software Engineering: Using UML, Patterns, and Java*. Boston: Prentice Hall, 2010
- Bardi, James A. *Hotel Front Office Management*. Hoboken, NJ: John Wiley & Sons Inc., 2007.
- Duckett, Jon. *Beginning HTML, XHTML, CSS, and JavaScript*. Hoboken, NJ: Wiley, 2010.
- Miles, Russ, and Kim Hamilton. *Learning UML 2.0*. Sebastopol, CA: O'Reilly, 2006.
- Nixon, Robin. *Learning PHP, MySQL, and JavaScript*. Beijing: O'Reilly, 2009.
- Ramakrishnan, Raghu, and Johannes Gehrke. *Database Management Systems*. Boston: McGraw-Hill, 2003.
- Website Template - *ChocoTemplates - The Sweetest CSS Templates WorldWide*. Web. 1 Feb. 2012. <http://chocotemplates.com/>.
- "Event-Driven Architecture," Wikipedia, http://en.wikipedia.org/wiki/Event-driven_architecture