CS250 Project Report

BeAvis Car Rental Software System

By Cameron Cobb

1) User Manual:

1.1) Overview and System Requirements:

The BeAvis Car Rental Service is a web application designed to facilitate the renting and returning of rental cars in San Diego and other cities across the United States. The software system is accessible on any smartphone and computer and is available via the Internet through the BeAvis Rental Website. The smartphone or computer must be connected to WiFi or cellular data in order to access the application and use its features. The application is also available for download on any smartphone store, and the smartphone must have at least 5GB of storage in order to use the application. If the user is accessing the software via a smartphone, the user must allow the software access to the phone's internal GPS chip so that directions and current location can be sent automatically to the server hosting the application. If the user is on a computer, then its current location is not required, however the user will not be able to get directions to the nearest rental lot in the event that the nearest car lot has reached maximum capacity. Each manager is required to be logged into the system with their provided credentials to verify user requirements and facilitate the transaction of vehicles. A server at each rental car lot location is necessary to store the required databases that contain important information about the current status and availability of vehicles, payment information, and user account information.

1.2) User Requirements:

Use of Application:

The most important use of this application is that it will allow users to rent cars from any mobile or computing device that has access to the Internet. Customers should be able to view a wide range of cars and filter these results based on their desired specifications. This web application is designed for both customers and managers working at BeAvis car lots across the country. Users and managers will have access to the website by logging in with their account information. Users will have the ability to navigate across the website by clicking on various links to other pages. These other pages serve specific functions, such as browsing the directory of cars, viewing account information on a user specific account page, and managing payment information. The users should expect the software application to display cars along with a description for each one indicating the make, model, manufacturer, color, mileage, condition,

availability, location of the car lot it is located at, and the car capacity. Users will have the option to add cars to their shopping cart so that they can rent whichever vehicle they choose. Each user account will have a rental history that allows each customer to view past transactions and create customer reviews for the vehicles. These reviews will be automatically uploaded to the website's database. Once a vehicle is purchased, users can get directions to the car lot that currently stores the vehicle.

User Interface:

The user interface should be fairly straightforward and easy to navigate. There will be a login box for the user to enter their username and password. After logging in they will be greeted with a welcome message and a main menu will be displayed. The main menu will contain buttons to the various web pages, including navigating the car directory, updating information in the account page, and managing payments in the transaction page. A shopping cart icon will be displayed at the top right corner of each web page so that the user can check out their selected cars at any time on any page. There will be a search bar on the car directory website to search vehicles; there will also be a checklist box that will allow the user to filter search results of vehicles. There should be a visual map that displays the current location and the closest route to the car lot where the vehicle is stored at once the transaction is complete. A slider on each car that is selected can be used to see the different images of the car's interior and exterior. The user account page will contain numerous text boxes to change a component of a user's account if they desire. A save button at the bottom of the page will ensure that the new information is saved into the software system's databases. The shopping cart will display all of the vehicles that the user has marked while searching in the directory. There will be a box at the bottom that will let the user checkout. Payment details will be displayed, and the user will have the option to enter new card details if they haven't done so already.

Constraints on Hardware/Software:

The user's device must be connected to the Internet at all times. This is because the website tracks the user's location and sends it to the database where the price of the rental is constantly changing depending on how far the user drives and to what destination. Also, the GPS chip installed on the car may be susceptible to water and or other physical damage, which may result in financial losses for the car company if not properly addressed . If the car battery dies, the car's GPS system will not function because of the loss of power; this may pose problems in case the managers need to locate vehicles that have been lost or stolen. Also, the server handling the requests of users may be temporarily down when there is a high volume of traffic entering the website; this may result in the loss of rental revenue if not solved in a timely manner.

2) Design Documents:

2.1) Functional Requirements:

As described above, users will be able to access the website from their mobile device or computer. Mobile users will have the option to download the app on the device to streamline the car rental process by enabling GPS location services. The software system consists of multiple webpages that separate the website into different components. The website should direct the user to a main menu page upon arrival. The main menu should ask the user to either log in with an existing account and create a new account (with an email address and password). The main menu will also ask the user to enable 2 factor authentication with a provided list of authentication apps. The user can access three web pages from here: a user page, a car page, and a payment page. The user page provides an organized interface with numerous buttons that allow the user (customer) to update their account information like their email, password, phone number, insurance provider, etc… It also provides the option for the user to view their rental history; this means that they can look at previous transactions and which vehicles they rented. The user page should contain a shopping cart that allows users to bookmark cars and save them for later. They have the ability to view these cars in the user page. Finally, the user will have the option to create a customer review of the vehicle that they have rented from their rental history. The car page provides the user with the ability to search for a car by a specification or location.Users can choose from the catalog of vehicles by filtering results based on condition, car make, model, type, color, and availability. Each car entry in the catalog will have a short description of the car as well as a gallery of images of the vehicle's exterior and interior so that the user can look at all of the car's characteristics and features. In the text description below the images of the vehicle is a phone number to contact the car lot that the vehicle is stored at. Users can save this phone number if they wish to rent the vehicle over the phone, however the shopping cart on the website allows the user to automatically reserve the vehicle at the particular car lot. Location can also be used to filter search results of vehicles from a vehicle directory such as the nearest specified car from the user. A directory of vehicles for each car lot is provided by the external databases; these external databases stored in the server will be illustrated in greater detail in the following pages. There will be a button that the user can press on the interface of the car page that will automatically get directions to the nearest car lot facility, assuming that the user will enable location services on their device. There will be a button that allows the user to add the desired car to their shopping cart, where they will be able to edit their payment information and finalize the transaction. Specific requirements are set so that only qualified users can rent vehicles from the website. One requirement is that users must be at least 21 years of age. Users must have a credit score of at least 600 for financial reasons. During the creation of their account on the website, users will be asked to provide proof of credit score and a SSN or birth certificate to verify U.S. citizenship and age. After the user uploads documentation, they will not be required to submit them again. All car and user account information will be stored in external

databases so that only managers and other authorized personnel can access and modify this data. Rental car prices will have different flat fee rates depending on the specifications of the vehicle. Vehicles will start at around $100 a day; additional fees such as extra mileage and taxes will be added to this rate as well. Users must specify how long they plan on renting the vehicle for and approximately the range (in miles) that they will be driving. If users fail to return the car by the specified date, they will be charged an additional amount for every day that the car is late. This fee will increase every day the car has not been returned to the original car lot. Scheduling the car rental in advance allows users to plan ahead and ensure the smooth process of rental transactions. The BeAvis Car Rental System accepts credit, debit, and other third party forms of payment like Venmo, Paypal, Apple Pay, and more. GPS chips are installed on each vehicle so that cars can be tracked down if not returned to the car lot. In the case where a car is stolen or broken before they drive it, users will be issued a full refund.

Non-functional Requirements:

The most critical non-functional quality of this system is reliability because without it, user satisfaction could not be guaranteed; the system could improperly display images of a vehicle or show that a vehicle is located in one car lot while it is actually stored in a different one. A reliable car rental service would provide the most accurate information so that user needs are fulfilled and customers are satisfied with their purchase and service.

The second most important quality of this system is the security of the software system. Users should feel confident that their payment information will not be stolen after storing it on the website. Other sensitive personal information is stored in the website's various databases, thus the security of the system prevents hackers and unauthorized persons from accessing and using this information for malicious purposes. Finally, security measures should be properly taken into account, especially with the rise of car break-ins and theft. Without proper security, cars could be stolen or users could lose valuables such as money and jewelry; this software system should alert the respective car lots of any security issues or breaches that occur.

Finally, performance is another key component of the software system because cars should be able to be added or removed from the current fleet at any moment. The performance of the website should be able to handle large amounts of traffic to ensure that all customers are able to rent vehicles. Without performance, inaccurate data would be stored and customers could face the risk of not receiving the vehicle they payed for.

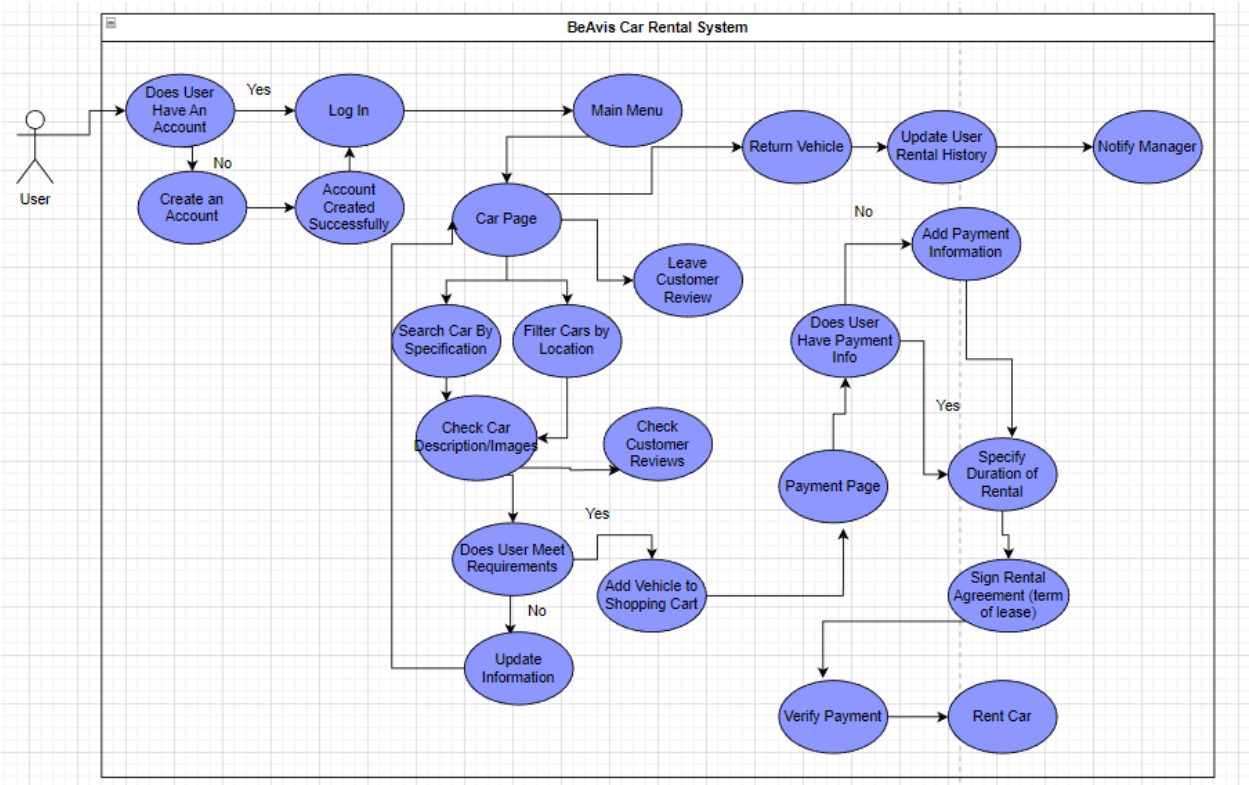
Planned or Anticipated Modifications:

This website will initially launch as a prototype by serving one physical car lot BeAvis location. There will be volunteers from the local community to act as the customers, and they will rent the vehicles from their phone or computer with online "credit" from BeAvis.

The car rentals will essentially be free for this test. BeAvis developers will monitor and track bugs in the software, and ensure that vehicles are properly displayed in the directories, their status to the public via the Internet and car lots will begin adopting the software system. We anticipate that more devices will have to be supported, so we will try to make modifications so that the system is universal. We anticipate more car lots to be constructed and more vehicles to be stored in every car lot as newer vehicles get released and manufactured in the country, therefore the web application must be able to support these new changes. For example: the high demand for the car rental service could convince BeAvis management that more car lots must be constructed in new cities. This system must be able to support this by ensuring that new car lots are properly registered and accounted for in the databases. Also, car manufacturers make this models of vehicles every year. Some companies may even release new models of vehicles every year, the website must be able to handle the changes in these names. The two anticipated changes above may require larger amounts of space to store in the database, therefore more servers may need to be installed to accommodate for these possibly large volumes of data being stored.

Use Case Diagrams:

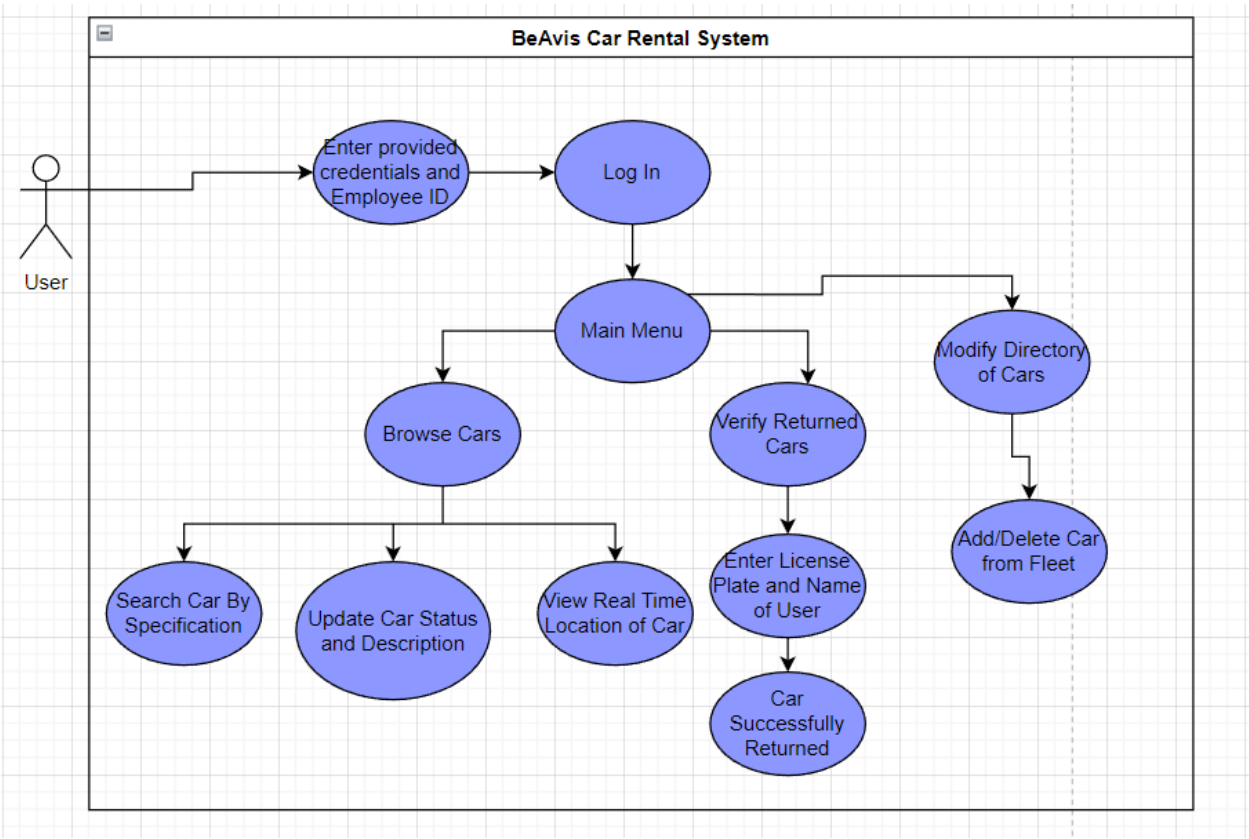
1. Use Case Diagram for Renting Vehicles (Customer Baseline Flow)



Here, the diagram illustrates that users must first have an active account in order to log in. After logging in, users will be taken to the main menu page where they can select to view the list of cars in the car page. This directory is constantly updated by the

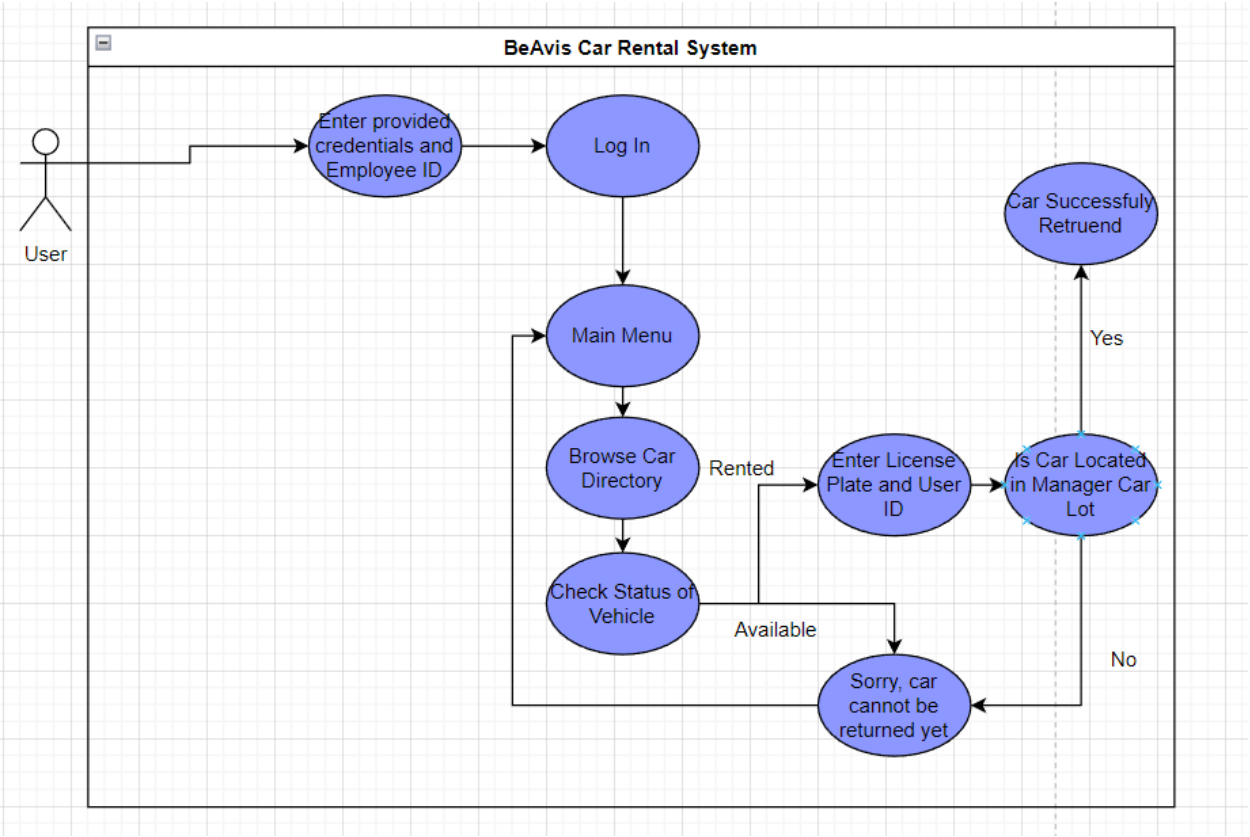
managers and will be stored in the databases described in a later section. The user will have the ability to search vehicles by specification (make, model, type, color, milage, condition, availability, etc.), filter cars by nearest location, or leave a customer review for a vehicle they have already purchased. The user will then select the car of their choice and be able to view the description of the car and the images of its interior and exterior. A list of previous customer reviews will also be attached to the page of the specified car so that the user can check if the vehicle is similar to its description. The software system will check if the user satisfies the requirements to rent a car; this means that it will check the user's credit score, insurance, age, and appropriate documentation to ensure that only reliable customers can rent vehicles. If one of these requirements are not met, the system will redirect the user back to the main menu and display an error message indicating that they must update their information. If the requirements are satisfied, the car can be added to the customer's shopping cart and the transaction process can begin. The system will check if the user has payment information attached to the account. If not then the system will ask the user to add a payment option so that the transaction process can continue. After the payment information is verified, the user will be able to specify the duration of the rental. Once this stage is complete, the user can sign the digital rental agreement. The system will verify that the payment information is correct, and afterwards the user will have successfully rented the vehicle.

2. Use Case Diagram For Managers Updating Car Directory and Verifying Returned Cars



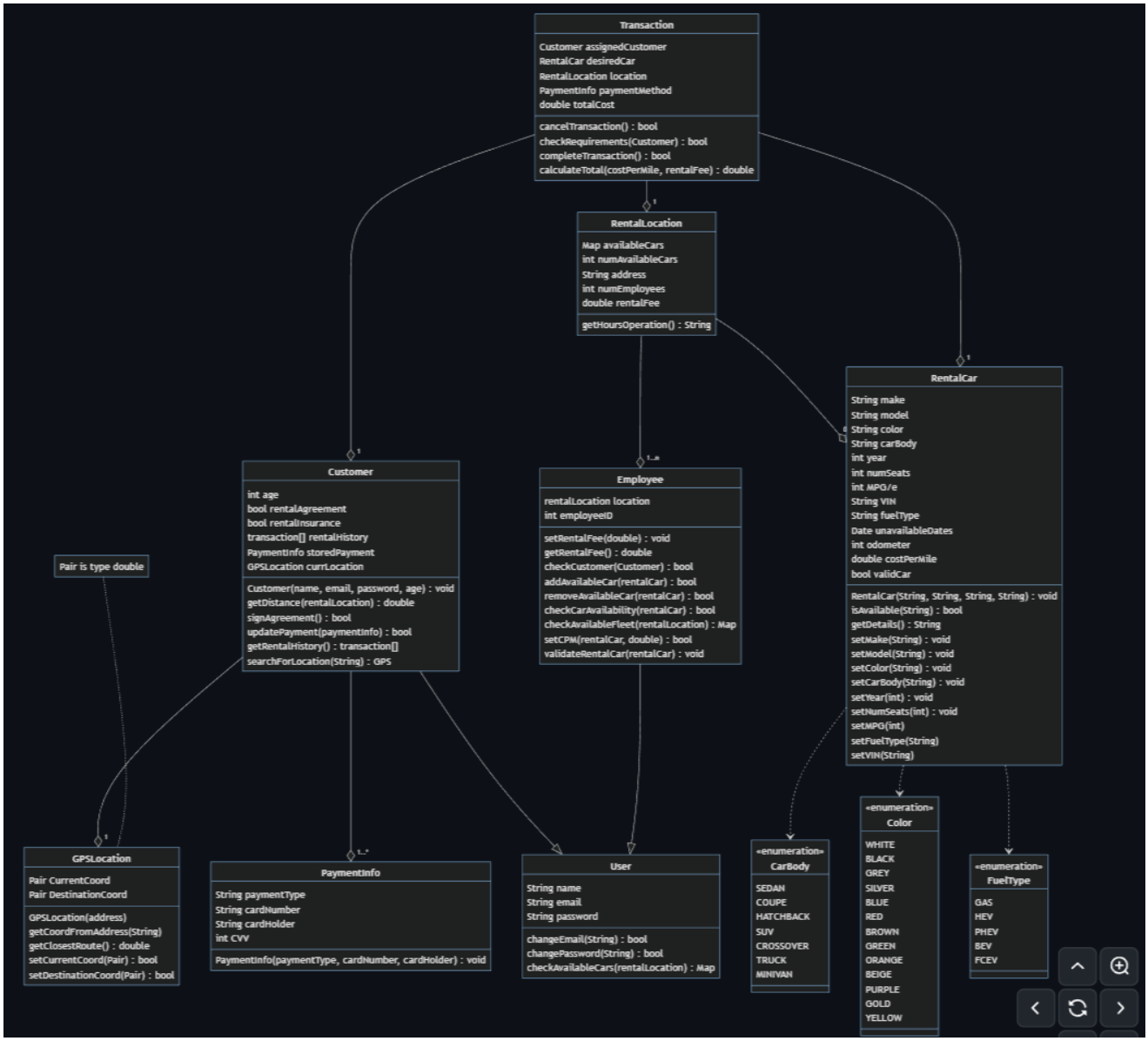
The diagram above shows that a manager must first log into the BeAvis website with their provided credentials. This includes their Employee ID, which prevents unauthorized people from accessing the system. After logging in, they are prompted to the main menu. The UI provides multiple options such as browsing cars, verifying returned cars, and modifying the current directory of cars available to users. If the manager selects “Browse Cars” on the interface, they will be able to see the list of vehicles that are currently visible to all users. Managers can search cars by location and specifications (just like customers), however they also have the option to modify or update the status or description of these vehicles. Since each car is equipped with a real time tracking GPS chip, all cars are visible by location to the manager in case emergency situations arise. Managers can also verify returned cars by entering the license plate number and name of the person driving the vehicle on a separate page. The car will be displayed to the manager where they can click a button confirming that the vehicle has been successfully returned. Managers can choose to enter another webpage that allows them to modify the current directory of vehicles available to users by adding or removing vehicle entries from the database. This will ensure that vehicles that are out of commission can be updated and removed from the directory to prevent users from renting this vehicle while it is unavailable.

3. Alternative Use Case Diagram for Managers Verifying and Handling Car Returns



This diagram first shows that the manager must enter their log in credentials to access the manager version of the web application. After logging into the system, they access the main menu where they browse the car directory. The vehicles are then sorted by availability or the license plate number is manually entered to find the desired vehicle. This desired vehicle is the vehicle that a user returns to the car lot. After pressing the button to confirm the returning of the vehicle, the system checks the status of the vehicle. If the status of the vehicle is set to “Available” then the car has not been involved in any recent transaction and is not currently being driven by a customer. The system will display a message saying the vehicle cannot be returned and will return back to the main menu. If the status of the vehicle indicates that a user is driving the vehicle, then the system will ask the manager for the license number and User ID of the person driving the vehicle. Then the system will check the current location of the car’s GPS chip and will compare this location to the car lot’s registered coordinates. If the car is within a certain radius of the car lot, then the vehicle can be successfully returned. If the vehicle is far away from the car lot, then the system cannot handle the return because the user is still driving the vehicle, so it will return back to the main menu and display an error message.

UML Class Diagram:


Transaction
Customer assignedCustomer
RentalCar desiredCar
RentalLocation location
PaymentInfo paymentMethod
double totalCost
cancelTransaction() : bool
checkRequirements(Customer) : bool
completeTransaction() : bool
calculateTotal(costPerMile, rentalFee) : double
RentalLocation
Map availableCars
int numAvailableCars
String address
int numEmployees
double rentalFee
getHoursOperation() : String
RentalCar
String make
String model
String color
String carBody
int year
int numSeats
int MPG/e
String VIN
String fuelType
Date unavailableDates
int odometer
double costPerMile
bool validCar
RentalCar(String, String, String, String) : void
isAvailable(String) : bool
getDetails() : String
setMake(String) : void
setModel(String) : void
setColor(String) : void
setCarBody(String) : void
setYear(int) : void
setNumSeats(int) : void
setMPG(int)
setFuelType(String)
setVIN(String)
Customer
int age
bool rentalAgreement
bool rentalInsurance
transaction[] rentalHistory
PaymentInfo storedPayment
GPSLocation currLocation
Customer(name, email, password, age) : void
getDistance(rentalLocation) : double
signAgreement() : bool
updatePayment(paymentInfo) : bool
getRentalHistory() : transaction[]
searchForLocation(String) : GPS
Employee
rentalLocation location
int employeeID
setRentalFee(double) : void
getRentalFee() : double
checkCustomer(Customer) : bool
addAvailableCar(rentalCar) : bool
removeAvailableCar(rentalCar) : bool
checkCarAvailability(rentalCar) : bool
checkAvailableFleet(rentalLocation) : Map
setCPM(rentalCar, double) : bool
validateRentalCar(rentalCar) : void
Pair is type double
GPSLocation
Pair CurrentCoord
Pair DestinationCoord
GPSLocation(address)
getCoordFromAddress(String)
getClosestRoute() : double
setCurrentCoord(Pair) : bool
setDestinationCoord(Pair) : bool
PaymentInfo
String paymentType
String cardNumber
String cardHolder
int CVV
PaymentInfo(paymentType, cardNumber, cardHolder) : void
User
String name
String email
String password
changeEmail(String) : bool
changePassword(String) : bool
checkAvailableCars(rentalLocation) : Map
«enumeration»
CarBody
SEDAN
COUPE
HATCHBACK
SUV
CROSSOVER
TRUCK
MINIVAN
«enumeration»
Color
WHITE
BLACK
GREY
SILVER
BLUE
RED
BROWN
GREEN
ORANGE
BEIGE
PURPLE
GOLD
YELLOW
«enumeration»
FuelType
GAS
HEV
PHEV
BEV
FCEV

Enumerations

We have three enumerations for CarBody, Color, and FuelType that the RentalCar class will use. Because of the variability in how an employee could input these values, we've streamlined it using these enumerations to keep it consistent across all cars.

User

The `User` class is a superclass of both `Customer` and `Employee`. The user class is what lets us create individual account access to the system. The `User` class contains name, email, and password that is required for either types of accounts. The functions that are in the class is general account manipulation and availability of vehicles given a location it gets passed

Customer

The `Customer` class inherits the `User` class that covers the functions and members that a customer would need to manipulate their rental account. The class also contains relevant documents such as `rentalAgreement` and `rentalInsurance` required to complete a transaction. In this class, the `GPSLocation`, and `PaymentInfo` are used as well which will be described later in the document.

Employee

The `Employee` class now includes a validation method that checks if the customer has the proper documents and is of age in order to rent a vehicle. The changes also include functions that allow an Employee to add, modify, or remove cars in an existing fleet at a location.

Payment info

The `PaymentInfo` class is another object that a customer creates either one or more instances of. This class contains credit card information that can be stored and used for future transactions.

GPS Location

The `GPSLocation` class manages directions to and from a customers location to a given rental location using coordinates that the class finds given an address string passed by the customer. Any function that requires location for finding directions of will use the `GPSLocation` class. It now utilizes Pairs and GPS coordinates that allows for precise locations.

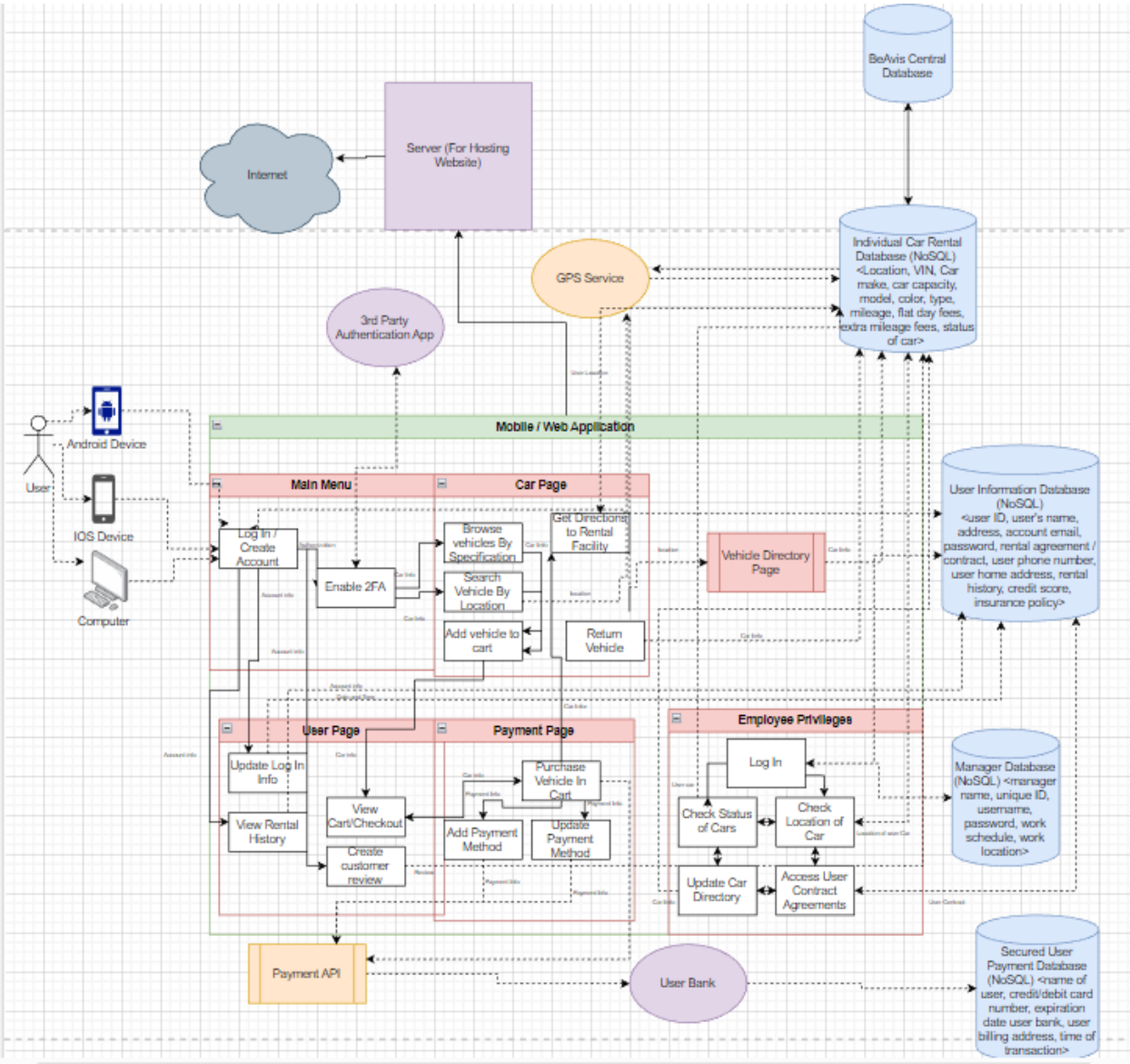
RentalCar

The `RentalCar` class has new setters to update the class's members such as `make`, `model`, `color`, `carBody`, `year`, and `setNumSeats`.

RentalLocation

Added an `availableCars<map>` that contains available vehicles at a given RentalLocation.

Software Architecture Diagram:



The Software Architecture Diagram for the Car Rental System starts from the user. Regardless of the device used, users are required to login to access the system. If enabled, the two-factor authorization will require users to verify their login with a 3rd party authentication app before they can proceed with the rest of the system. In addition to a valid login, users will need to have their rental requirements verified with the user information database containing the user's name, address, email, password, rental agreement, rental history, credit score, and insurance information. The user can view their user page and update it as necessary, and also add a payment method through the system's payment API. Once the user's rental requirements have been deemed valid by the system, they will be allowed to browse and rent cars. While browsing through the catalog of cars available to rent, a user can browse by nearest location or by specifications. Users can also set a customer review if their rental history shows they

have previously rented that car. When renting a car, users can view the make, model, colour, type, availability, condition, mileage, flat day fees, and extra mileage fees of that car. The car also has a GPS service that tracks its location for both the user and the employees to know. In addition to seeing any given car's location, employees can also check the status of any of those cars. Employees can also access the rental agreements of any user and update the car directory. Our user class consists of the user's basic login information: the user's name, email, and password stored as Strings. The user class also contains two functions to change the email and password associated with a user. The user class branches into the customer and employee classes to differentiate the two different types of users. The employee class is simpler than the customer class, as the employee class only stores the employee ID as an integer and the location of that employee's workspace as an object of the rentalLocation class. Besides that, the employee class contains three functions: one to check the customer information, one to check the available cars of a rental The rentalLocation class stores the available cars as rentalCar objects, the number of available cars as an integer, the address of the lot as a String, the number of employees as an integer, and the rental fee as a double. The rentalLocation class also contains a function to retrieve the hours of operation. A rentalCar object contains the make, model, colour, body, and fuel type of the car as Strings, and the number of seats and the odometer as integers. A rentalCar object also stores the miles per gallon and the cost per mile as doubles, and stores the year and unavailable dates as their own separate objects. The rentalCar object can also call functions to check the availability and information of a car. Back to the different types of users, a customer class contains the age of a customer as an integer, and if that customer has signed the rental agreement and has insurance as booleans. A customer class also contains the payment information and the current location of a customer, as well as their rental history in the forme of transaction objects. A customer class also contains the functions that can search for the customer's current location, get the distance from a customer to the rental location, check if the customer has signed the rental agreement or updated their payment, and get the customer's rental history. The transaction class, the last class that has not been gone over, stores the assigned customer, desired car, rental location, payment method, and total cost of a specific transaction. The transaction class also contains the functions that allow a customer to cancel their transaction, or allow the system to check the customer's rental requirements.

Software Architecture Diagram Revisions:

Compared to the previous version of our software architecture diagram, numerous improvements and additions have been made to adjust and reflect our data management strategy. First, we added an additional database to store manager information such as manager name, username, password, unique ID, work schedule,

and work location. This will be useful when we create relationships between various tables of data (like creating a relationship between users and managers) in such a way that will follow data normalization, where we will specifically isolate independent multiple relationships since many managers can help and assist many customers/users. We will discuss the various strategies for this in the tradeoff discussion below. The second improvement we made to the diagram was adding two additional pages: one page that stores the directory of cars available and another that serves as a transaction/payment page where users can edit/add payment methods and check out selected vehicles in their cart. We also added more use cases in the web pages such as an option to return a car, get directions to the nearest facility where the selected car is stored, view shopping cart, and updating user log in information. Finally, we distinguished between interactions between the webpages and the interactions with the databases: the interactions and flow of the website were signified with solid arrows while interactions with the databases and other 3rd party apps/systems were illustrated in dashed arrows. This makes it easier to visualize the relationships between the different users, use cases, and elements of our website application. We also attached additional text boxes next to each arrow relationship to show the inputs and outputs of each use case in the SWA diagram as well as how this information is used to edit and/or make additional entries in our databases. For example: searching a vehicle by location first requires a “user location”, which is seen in a text book as an output of the “Search Vehicle By Location” box in the diagram. This location is found by interacting with the 3rd party GPS app connected to the software system. This “user location” is returned and used by the vehicle directory page to filter vehicles from the car rental database based on their nearest location to the user. These vehicles are found by passing “car information” from the vehicle directory page to the car rental database. These vehicles are essentially filtered based on the user location because each car entity in the database is sorted based on their location attribute, where cars with a closer distance to the user’s location will be towards the top of the vehicle directory page and cars farther away from the user will be towards the bottom.

General Timeline and Responsibilities:

First, we would break down into two teams: one team that focuses solely on the mobile application for the car rental system and the other team that implements the website version of the system. We would have specific front and backend teams for each respective system. Vincent would be in charge of the backend team for both systems, Cameron would be in charge of the backend team for both systems, and Brendel would be in charge of the full stack team, ensuring that both front and backend aspects of the website and app are connected properly. The front-end part of the team would deal with implementing functionality and visual elements that make the system easy to use by the user. The backend part of the team would deal with managing databases and connecting them to our systems to store user information car directories and user

payment information. There would be a separate team for communicating and facilitating the functionality across the two systems.

Front-End

The tasks for Vincent and the front-end team members would be to implement an easy-to-use and intuitive UI, ensure functionality across web pages or different parts of the system internally, make sure that the information is displayed correctly across the system, verify that car images are clearly visible, and use a universal text format to ensure professionalism and cost-efficiency. Each of these tasks would be given to talented members of the front-end team selected by Vincent.

Back-End

The tasks for Cameron and the back-end team members for their respective systems would set up each database (car rental database, user information database, and user payment database), connect the car rental database to the central Beavis database, connect each database to the web and mobile applications, implementing a payment API to allow users to add/update payment information that will be stored in the payment database, testing the systems to ensure that users can log in and change personal information, ensure that the system can handle large amounts of traffic, test to see if the code is scalable across different devices, and verify that new cars can be added/removed from the rental database. Each of these tasks would be given to talented members of the back-end team selected by Cameron.

Timeline

This project is expected to be implemented with full functionality as specified by the client, by Month 8, no later than Month 12.

Timeline	Task Type
Month 0	Meet up with Software Engineers: front-end, back-end, and full-stack; to discuss expectations and specifications listed in the documents. In this time period, the designers (us) will make sure that all teams are on the same page before we start implementation, such as formatting of code and documentation to create a professional and unified style.
Month 1 – Month 2	Back-end and front-end meet with their respective teams and partition the listed tasks amongst the individuals. During this time period, each team respectively will start the implementation process before the full-stack starts looking at the interaction between the teams. This is meant for the back-end and front-end developers to get a proof of concept for the full-stack to start working on.

	Tasks To Be Completed: Front End: Create web pages, use universal text format Back End: Set up databases, connect databases, start creating payment API
Month 3 – Month 6	Full-stack developers start integrating back-end and front-end systems and verify that they work in unison. Back-end and front-end teams continue to develop beyond the proof of concept and work more toward full implementation. This is a pre-testing stage to make sure that the teams are on the same page. These developers can flex between the teams as necessary. Tasks To Be Completed: Front End: Verify that images can be displayed on webpages, ensure user information is displayed correctly, and create code that allow webpages to interact Back End: Test backend code to see if it is scalable across different devices, create disaster scenarios for high traffic situations, ensure system accurately stores and accesses user, payment, and car information.
Month 7 – Month 8	Present the project to the client, make sure it functions as desired. Also serves as extra time to clean up loose ends, bugs, etc. Tasks To Be Completed: Front End: Finalize UI to ensure efficiency and functionality across both systems Back End: Test edge cases and adjust implementation/code as see fit
Month 9 – Month 12	Flex Time: This period is a built in buffer in the event that the 8 month period is not sufficient for the team. Avoid taking more time beyond Month 12.

Verification Tests:

Unit Testing

Testing Unit: Car.setCarMake

> By Cameron Cobb

```
<void, setCarMake(name = “Toyota”)>
RentalCar.make == “Toyota”
```

This test case should not return anything, but it should set the string attribute “make” to “Toyota” in the RentalCar class. When this content of “make” is compared to the “Toyota” function argument, a value of true should be returned. This indicates that the make attribute in the RentalCar class was properly set with the function argument “Toyota.”

```
<void, setCarMake(name = "") >
Rentalcar.make == ""
```

This test case should not return anything, but it should set the string attribute “make” to an empty “” string in the RentalCar class. When RentalCar’s make attribute is compared to its argument, or the “” string, a value of true is returned. This indicates that the make attribute was properly set with “”.

```
<void, setCarMake(name = 5)>
Error: Invalid input
```

This test case should print an error message since the function argument, 5, is not a string argument. This error message should indicate that RentalCar.make should NOT equal 5 since an incorrect type is passed as an argument to the setCarMake function. This indicates that the make attribute in RentalCar class was not properly assigned.

```
<void, setCarMake(name = 'c')>
Error: Invalid input
```

This test case should print an error message since the function argument, c, is not a string argument but a char argument. This error message should indicate that RentalCar.make should NOT equal ‘c’ since an incorrect type is passed as an argument to the setCarMake function. This shows that the make attribute in RentalCar class was not properly assigned.

Testing Unit: GPSLocation [setDestinationCoord(Pair newDestination)]

> By Vincent Chu

```
setDestinationCoord(newDestination = 32.7774, 117.0714)
```

This test case should not return anything, and it will set the object DestinationCoord of the GPSLocation class to the two doubles. When this function is correctly executed, a value of true should be returned to the caller to indicate that it was successful.

```
setDestinationCoord(newDestination = 32.7774)
```

This test case should not return anything, and it will return an error because a Pair is two doubles, not one. When the error message is returned, a value of false should be returned to the caller to indicate that it was unsuccessful.

```
setDestinationCoord(newDestination = “32.7774”, “117.0714”)
```

This test case should not return anything, and it will return an error because a Pair is two doubles, not two Strings. When the error message is returned, a value of false should be returned to the caller to indicate that it was unsuccessful.

Testing Adding A New Vehicle To Available Car Map in Employee

> By Cameron Cobb

RentalCar newCar(string make, string model, string color, string bodyType) Employee.addAvailableCar(RentalCar)

```
RentalCar newCar(make = "Toyota", model = "Prius", color = "BLACK", bodyType = "HATCHBACK")
Employee.addAvailableCar(newCar)
Employee.validateRentalCar(newCar)
newCar.validCar == true

Output: Employee.checkAvailableCar(newCar) == true
```

This test case should add the newCar RentalCar object to the availableCars Map located in RentalLocaton. The newCar RentalCar object's attributes are set according to the string arguments: make is set to "Toyota", model is set to "Prius", color is set to "Black", and carBody is set to "Hatchback". First, Employee.validateRentalCar(newCar) should set newCar's validCar attribute to true since string arguments passed into the object instantiation are valid and each argument matches the data that is stored in the central database. Then, newCar.validCar should return true. Employee.checkAvailableCar(newCar) should return false, indicating that the RentalCar is not currently added to the availableCars map. newCar.validCar should return true, which means that the rental car matches the specifications stored in the database. Afterwards, the newCar object is added to the availableCars map located in the RentalLocation, which is an object of the Employee class. After the newCar is added to the availableCars map, checkAvailableCar(newCar) should return true. The test case passes for this scenario.

```
RentalCar newCar(make = 4500, model = "Tundra", color = "GREY", bodyType = "TRUCK")
Employee.addAvailableCar(newCar)
Employee.validateRentalCar(newCar)
newCar.validCar == false

Output: Employee.checkAvailableCar(newCar) == false
```

This test case should produce an error since the newCar object is not instantiated with valid arguments. Specifically, the make attribute for the RentalCar object is not properly set because the argument 4500 is an int value and not a string. First, Employee.validateRentalCar(newCar) should set newCar's validCar attribute to false since string arguments passed into the object instantiation is not valid. This is verified by checking that newCar.validCar is equal to false. An error message should appear saying that the newCar was not instantiated properly, so the newCar is not added to the availableCar map via the addAvailableCar(newCar) function in the Employee class since it is not a valid object of the RentalCar class. Therefore, Employee.checkAvailableCar(newCar) will return false since newCar is not a valid object, thus indicating that the new car is not added to the fleet of available cars. If any of the string arguments to the RentalCar object creation are invalid based on type, then a valid object will never be instantiated and inserted as an available car in the availableCars map.

```
RentalCar newCar(make = "", model = "", color = "", bodyType = "") >
Employee.addAvailableCar(newCar)
Employee.validateRentalCar(newCar)
newCar.validCar == false

Output: Employee.checkAvailableCar(newCar) == false
```

This test case will not produce an error since the attributes of newCar will be set to null. First, the newCar RentalCar object is instantiated with null values for its attributes make, model, color, and bodyType respectively. Employee.valiateRentalCar(newCar) should set validCar attribute of newCar to false since null values for the attributes will not be useful to the system. newCar.validCar returns false, indicating that the rental car does not meet the specifications of cars in the database. Employee.checkAvailableCar(newCar) should return false since the object should not be present in the availableCars map. Employee.addAvailableCar(newCar) adds the object to the avalableCars map, and Employee.checkAvailableCar(newCar) should return false, which means that the newCar object was not successfully added to the fleet of available cars and that an object with nonnull and valid string values for its members should be added to the availableCars map in RentalLocation via the addAvailableCar(newCar) function in Employee class.

```
RentalCar newCar(make = "Subaru", model = 500, color = 65.7, bodyType = "SEDAN")
Employee.addAvailableCar(newCar)
Employee.validateRentalCar(newCar)
newCar.validCar == false

Output: Employee.checkAvailableCar(newCar) == false
```

This test case should produce an error since the newCar object is not instantiated with proper arguments. Specifically, the model and color arguments are of type int and double respectively for the RentalCar object. This means that the object will not be instantiated correctly. First, Employee.validateRentalCar(newCar) should set newCar's validCar attribute to false since multiple arguments passed into the constructor are not valid types. An error message should appear saying that the newCar was not instantiated properly, so the newCar is not added to the availableCar map via the addAvailableCar(newCar) function in the Employee class. Therefore, Employee.checkAvailableCar(newCar) will return false since newCar is not a valid object, thus indicating that the new car is not added to the fleet of available cars. If any of the string arguments to the RentalCar object creation are invalid based on type, then a valid object will never be instantiated and inserted as an available car in the availableCars map.

```
RentalCar newCar(make = "25252325gffgfdg", model = "fafxcv", color = "RED", bodyType = "fsftt352")
Employee.addAvailableCar(newCar)
Employee.validateRentalCar(newCar)
newCar.validCar == false

Output: Employee.checkAvailableCar(newCar) == false
```

This test case should produce an error since the newCar object is not instantiated with proper arguments. Specifically, the make, model, and bodyType attributes are assigned with string values,s but these string values do not match data that is stored inthe BeAvis central car rental database and it will not be useful to our software system. First, Employee.validateRentalCar(newCar) should set newCar's validCar attribute to false since multiple arguments passed into the constructor do not match data that is stored in the rental database. An error message should appear saying that the arguments to the CarRental class' constructor did not match data that is stored in the database, so the newCar is not added to the availableCar map via the addAvailableCar(newCar) function in the Employee class. Therefore, Employee.checkAvailableCar(newCar) will return false since newCar's attributes are not accurate; If any of the string arguments to the RentalCar object creation do not match information that is stored in the database, then it should not be added to the fleet of available cars.

Testing Checking User Requiremens With Test Customer Objects

> By Vincent Chu

Customer newCustomer(string name, string email, string password, int age) checkRequirements(newCustomer)

```
Customer.Hubert(name = "Hubert", email = "hubertsmith@yahoo.com", password = "password", age = 58)
valid = checkRequirements(Hubert)
if (valid) print("The customer has met the requirements")
else print("The customer has not met the requirements")

Output: The customer has met the requirements
```

This test case successfully checked for Hubert's requirements and found him to have valid documentation to be able to rent a car. The variables in the customer class are all valid and his information checks out.

```
Customer.Barry(name = "Barry", email = "barrywomanilow@compuserve.com", password = "password", age = 17)
valid = checkRequirements(Barry)
if (valid) print("The customer has met the requirements")
else print("The customer has not met the requirements")

Output: The customer has not met the requirements
```

This test case successfully checked for Barry's requirements and found him to have invalid documentation to be able to rent a car. The variables in the customer class are all valid but he is not old enough to rent a car.

```
Customer.Vincent(name = "Vincent", email = "vchu5012@sdsu.edu", password = "password", age = "nineteen")
valid = checkRequirements(Vincent)
if (valid) print("The customer has met the requirements")
else print("The customer has not met the requirements")

Output: The customer has not met the requirements
```

This test case unsuccessfully checked for Vincent's requirements and found that his age variable was incorrectly defined as a String. The variables in the customer class are invalid so by default he cannot rent a car.

System Testing

Testing Customer Account Creation and Process of Execution Inside Website

> By Brendel Zuniga

Creation of new customer account, and Employee verifying the customer

```
// Assume that the database is already filled with cars that have been validated
Customer validCustomer(name = "Johnny", email = "johnny_appleseed@tester.com", password = "suP3rSecR3tP@SS", age = 26)
validCustomer.searchForLocation("5500 Campanile Dr, San Diego, CA 92182")

Customer invalidCustomer(name = delta, email = "", password = "weakpass", age = 12)
invalidCustomer.searchForLocation("1024 Totally Not Fake Address, ThisIsARealCity TrustMe, FK 10000")

Output:
    validCustomerGPS.CurrentCord == (32.778, -117.071)
    invalidCustomer.CurrentCord == void
```

First part of the test is to create a new customer from scratch and initialize all relevant user data: email, password, name, and age. Following that is a test to make sure that the GPS object was initialized with the proper coordinates from the address provided. On the other hand, an invalid customer would throw an error and prevent the customer from being created. Since the address provided is not a real address, there should not be any `CurrentCord` value because it couldn't be found.

```
// Assume an employee `testerEmployee` object exists
testerEmployee.checkCustomer(validCustomer) == false
```

After account creation of a customer, there are a few more steps that the customer needs to take before being able to process a transaction, namely signed the rental agreement and check the rental insurace on the account, as well as have a valid paymentInfo object. Because neither of these steps were taken before attempting to verify the customer, the employee function should return false. Regardless of whether or not the customer was created properly, since the `invalidCustomer` wouldn't be created in the first place.

```
validCustomer.signAgreement()
PaymentInfo businessDebit(Mastercard, "3423378501973298", "Johnny Appleseed", 123)
validCustomer.updatePayment(businessDebit)

Output:
    storedPayment.rentalInsurance == true
    storedPayment.rentalAgreement == true
    storedPayment.storedPayment == businessDebit
```

In order for an employee to verify a customer as a valid customer for the purposes of the transaction functions, there must be a signed agreement and a valid payment method stored. A valid customer check by an employee should then pass at this point.

```
testerEmployee.checkCustomer(validCustomer) == true
```

Testing Employee Ability to Update Vehicle Fleet

> By Brendel Zuniga

Adding, removing, and checking current vehicle fleet

```
// Adding and editing current fleet
RentalCar validCar("Toyota", "Prius", SILVER, HATCHBACK)
validCar.setYear(2019)
validCar.setNumSeats(5)
validCar.setMPG(53)
validCar.setFuelType(HEV)
validCar.setVIN("JT2BK12U530083835")

RentalCar validTruck("Ford", "F-150 Lightning", BLACK, TRUCK)
validTruck.setYear(2023)
validTruck.setNumSeats(5)
validTruck.setMPG(78)
validTruck.setFuelType(BEV)
validTruck.setVIN("1FTNW21L41EB18470")

RentalCar invalidCar("Subuwu", "Impretzel", GOLD, SEDAN)
invalidCar.setYear(1000)
invalidCar.setNumSeats(-1)
invalidCar.setMPG(-20)
invalidCar.setFuelType("None")
invalidCar.setVIN("21DSFJKLJF9320")

RentalCar inProgressCar("Toyota", "Corolla", RED, SEDAN)
inProgressCar.setYear(2013)
inProgressCar.setNumSeats(5)
inProgressCar.setMPG(24)
inProgressCar.setFuelType(GAS)
inProgressCar.setVIN("21DSFJKLJF9320ERS9820")

validCar.getDetails()
validTruck.getDetails()
invalidCar.getDetails()
inProgressCar.getDetails()
Output:
    "2019 Toyota Prius"
    "2023 Ford F-150 Lightning"
    "Error: invalid vehicle"
    "2013 Toyota Corolla"
```

What we're testing here is if we were to try to first, create vehicles to be added to a location's fleet. In order to check if a car was successfully created, then a `.getDetails()` function would return with the proper output with included details. The function call on an invalid vehicle, or a vehicle that does not exist, would simply throw an error.

```
// Assuming that there exists a `testerEmployee` whose location is set to SDSU
Employee testerEmployee(SDSU)

testerEmployee.validateRentalCar(validCar) == true
testerEmployee.validateRentalCar(validTruck) == true
testerEmployee.validateRentalCar(invalidCar) == false
```

Once the objects have been created properly, the `validateRentalCar` check should return true because all of the necessary information has been populated. It should only fail on the object that did not meet the requirements. We purposefully did not validate the Toyota Corolla because we will use it's invalidated status to test against later on.

```
// Assuming that there exists a rental location `SDSU` and `UCSD` with cars already in the fleet
SDSUFleet = testerEmployee.checkAvailableFleet(SDSU)
UCSDFleet = testerEmployee.checkAvailableFleet(UCSD)
```

This should return a Map of rentalCars that is associated with that specific location.

```
testerEmployee.addAvailableCar(validCar) == true
testerEmployee.addAvailableCar(validTruck) == true
testerEmployee.addAvailableCar(invalidCar) == false
testerEmployee.addAvailableCar(inProgressCar) == false
```

The `addAvailableCar` function should only ever modify the testerEmployee's rentalLocation's fleet. The program should add `validCar` and `validTruck` without any issue because it's been validated by an employee. The `invalidCar` should fail because the object was not created properly, therefore making it ineligible for validation. The `inProgressCar` should also return false; even though the object was created properly with valid inputs, it has yet been validated by an employee, therefore not being able to add the vehicle to the fleet.

```
(SDSUFleet == testerEmployee.checkAvailableFleet(SDSU)) == false
(UCSDFleet == testerEmployee.checkAvailableFleet(UCSD)) == true
```

These two lines compare the fleet before and after cars have been added to verify that the cars were only added to the employee's fleet and not a different fleet. This also checks if they were added properly since the number of cars in the fleet should have changed

```
testerEmployee.removeAvailableCar(validCar) == true
testerEmployee.removeAvailableCar(validTruck) == true
testerEmployee.removeAvailableCar(nonExistentCar) == false
```

And finally, these two lines should determine whether or not the remove function works as expected; since `validCar` and `validTruck` were previously added, they could be removed without hitch. Since nonExistentCar is not in the fleet, it would return false because there's nothing to remove in the first place.

Validation Tests:

There are a few methods that we could implement to test our software system with our clients to ensure that user requirements are met along with user feedback being provided. First, we would ensure that all user requirements are satisfied by walking through each use case diagram and comparing the functionality of the website's use cases to what the client wanted. One example is the first use case diagram; it illustrates how the user can rent a car on their device. The website first asks if the user has an account: if they do then the system asks for their current login username and password. Assuming that they enter valid credentials, the website will prompt them to the main

menu. If they do not have an account, the user will be able to make a new account by providing the contact information and a desired username and. At this point, multiple user requirements have already been satisfied; according to the client, users must create an account if they do not have one already. Asking the user if they have an account as soon as they enter the website ensures that new users do not forget to create an account. The user is then taken to the main menu where they can perform a variety of functions such as browsing cars, leaving a customer review, and returning a vehicle. Renting a vehicle involves numerous steps (for example finding the desired vehicle). This portion of the diagram satisfies client requirements by streamlining functionalities that serve to help the user find and rent vehicles based on their preferences. The diagram then specifies that users have the option to either search the car by specification or select filters from a dropdown menu on the website. Again, this satisfies user requirements by creating a universal system for finding and narrowing down the entire inventory of vehicles to give the user more freedom and control over the car rental process. After selecting each car, the user can check the images and description of the vehicle to see if the car matches their interests or is in suitable condition. After selecting the vehicle, the system checks whether or not the user satisfies requirements by verifying their insurance policy, credit score, and age. While not specified by the client, it can be assumed that the client only wants eligible and qualified people from renting vehicles. To ensure that this is the case, we will consult with the client about the specific details that will either allow or prevent a user from renting a vehicle. The vehicle is added to the customer's cart after the system checks their information. The user can then navigate to the payment page where it checks if the user has a pre-existing payment method. If so, then the user does not need to add another payment option. If not, then the user must add their desired payment method and info in order to proceed. This stage of the diagram addresses the user requirement that users must be able to link a payment method to their account for purchases. Then the user must specify the duration of the car rental and sign the rental agreement. After the payment is verified, the user now has successfully rented the vehicle. As we can see, this satisfies the user interface portion of the user requirements and how implementing buttons to checkout and adding new APIs to save new payment information helps efficiently manage and store payment information. This makes it more convenient for both parties by removing the difficulties of traditional payment issues with physical cards while streamlining the entire car rental process in one web application. The other use case diagram that fulfills user requirements is the one regarding managers adding and updating vehicles from the fleet. It is apparent that both managers and users must have login credentials to access the website, so this use case diagram shows that both managers and users can log in to the website. Managers are able to browse the car directory and directly modify the descriptions and images of each car. They can also search vehicles, similar to users, and they can view the real

time location of cars via a visual map displayed on screen. One requirement for the user was to ensure that vehicle descriptions and statuses are modified so car lots can distinguish between vehicles. So this portion of the diagram satisfies that user requirement. Similarly, managers can verify returned cars by entering the license plate and user ID of the person renting the vehicle; this fulfills the requirement that vehicle inventory is properly managed by the software system and statuses are updated as vehicles are both entering and exiting car lots across the country. In general, the steps I chose to verify that client/user requirements are met by this software system was by traversing through each use case diagram and comparing how the results of the diagram fulfilled user requirements. Finally, communicating with the client periodically and ensuring that the website is meeting up to their standards and requirements ensures that we are building the right system for the user. This would mean a presentation of the BeAvis software system and its capabilities and functionalities. The client could then approve or disapprove of the prototype and give suggestions for improving the system to further achieve expectations and user requirements.

Data Management Diagram:

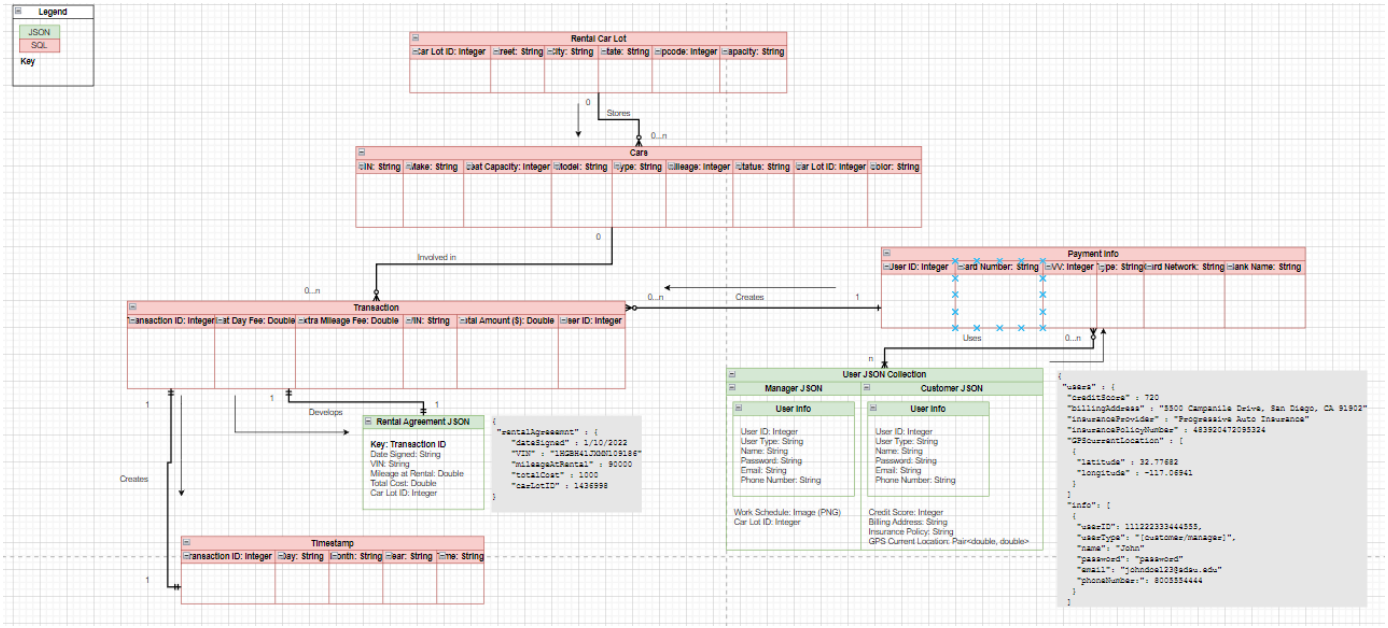


Diagram Description:

Our diagram utilizes a combination of SQL schema and a key-value NoSQL JSON format. The table begins with an SQL table representing Rental Car Lots in the Rental Car Lot database. The table contains 6 fields: Car Lot ID, Street, City, State, Zipcode,

and Capacity. The Car Lot ID is a unique 5 digit integer that distinguishes rental car lots from each other. Therefore each rental car lot will be given a different car lot ID from one another. The street is a string that represents the address of the car lot. The city field is a string that describes the city of where the car lot is located. The state field is a string that represents which state the car lot is located. The zipcode is a 5 digit integer where each car lot is located. It can be assumed that some car lots may be located in the same city, state, and/or zipcode, which is why each car lot should have a unique ID. Finally, the capacity filed in the Rental Car Lot SQL table is an integer value that represents the maximum number of cars available for storage. The reason why we chose these fields to represent a car lot in our database management system is because it is important for distinguishing car lots from one another as well as useful for getting directions based on where the location is located. Specifically, the unique car lot ID serves as a way to differentiate one car lot from another since each ID will be unique: no car lot can have the same ID. This will also help in normalizing our data and reducing redundancy between our SQL tables, as discussed in the tradeoff section. Storing the address, city, state, and zipcode of each car lot is necessary so that cars can be tracked and filtered by location. These fields are useful in assigning locations to cars since every car lot may store many cars, all of which require a specific location in order to be filtered by the customer based on their location. Connecting the Rental Car Lot and Cars SQL tables is a 1 mandatory to many optional relationship. This is accomplished by the Car Lot ID field stored in the Cars table in the Rental Car Lot database. This means that a rental car lot can store from 0 up to n number of cars, where n is equal to the capacity of the rental car lot. The word “stores” indicates that each rental car lot will store a certain number of cars. This relationship means that a Car Lot can store from 0 to many cars. The Cars SQL table has many fields: the VIN field stores the unique serial number string for each vehicle to be properly distinguished and identified, the Make field represents the car’s manufacturer name that stores it in the form of a string, the Car Capacity field stores an integer for the maximum number of occupants that can fit in each vehicle, the Model field is the string name of the vehicle model (e.g. Prius) by the manufacturer, the Type field stores the string corresponding to the type of vehicle (e.g. truck, SUV, sedan, etc.) for each vehicle entry, the Mileage field stores the integer number corresponding to each vehicle’s current mileage, the Status field corresponds to a string for each vehicle that indicates if the vehicle is not available, is already being driven, or is available at the specified car lot ID, the Car Lot ID field stores the unique integer that is associated with the Rental Car Lot ID (this is so that we can reduce redundancy for the number of fields in these datatables). Finally, the Color fields store the string color of each associated vehicle in each row of the table. The justification for incorporating these fields in the Car SQL table is because based on the software architecture diagram, the user will be able to filter what cars they wish to add to their shopping car based on a variety of specifications; these specifications could include the

make, model, seat capacity, type, mileage, status, and the color of the vehicle. Therefore, all of these fields must be stored in a database so that the directory of cars stored can be filtered by the user. It also helps differentiate vehicles from one another by adding more descriptive elements that may be useful to the manager in finding vehicles in the car lot. A 0 to many optional relationship arrow exists from the Cars SQL table to the Transaction SQL Table stored in the User Payment Database: this means that a Car can be involved in 0 to many (n number of) transactions according to how many transactions have been made with a specified vehicle. The Transaction SQL table has 6 fields: Transaction ID field stores the unique ID for each transaction in the form of a 6 digit integer, Flat Day Fee field stores a double value that represents the price for renting the car for one day, Extra Mileage Fee field represents the cost in dollars of type double to rent a car and drive over a certain predetermined mileage set by the manager, the VIN field provides a 17 character long string that acts as a unique ID for each vehicle, the Total Amount field describes the cost in dollars in the form of a double of the total price of the transaction after fees, taxes, and other user expenses; and the User ID is a unique integer identifier that is used to show which user was involved in the transaction. It is also useful in linking the Transaction SQL table with the Payment SQL table since every transaction uses some sort of payment information set by the user. The reason why these fields are critical to the Transaction SQL table and the database management system as a whole is because every transaction should involve a car, a user, fees, and a total amount that is due. Every car is differentiated by a unique car ID, therefore storing the car's VIN will allow us to avoid creating redundant data such as storing other information about the car in the same table. The user ID is required to distinguish the user of the transaction since it is very important for the managers to know who is purchasing which vehicle and for how much at a specific time in case of issues that arise in the rental process of a vehicle. Fees are necessary to store separately since fees may change depending on the type, make, and model of the vehicle; this would affect the total amount differently for each vehicle. Finally, a total amount field is required because it displays the cost of the entire transaction in dollars (stored in the form of a double) that will be displayed to the user on the web page; storing and displaying this information is critical to both the user (since they should know how much it costs to rent the vehicle) and other employees of the company. Multiple relationships exist from transaction: a 1 to 1 arrow connects from the Transaction SQL table to the Timestamp table; this means that one transaction should create one timestamp that will be used to indicate when a transaction has occurred. It will also be helpful for the user because, through the 3rd party Bank API based on the user's credit card, it will send this timestamp and transaction information to the databases of the user's respective bank. The Timestamp SQL table has 5 fields: Transaction ID, Day, Month, Year, and Time. The Transaction ID field stores unique IDs for each transaction in the form of a 6 digit integer. Since many vehicles may be

purchased in one transaction, storing redundant information about vehicles and fees would be redundant since it is already stored in the Transaction table in the User Payment Database. Therefore, a Transaction ID will limit the redundant information between the two SQL tables. The Day field will store the corresponding day of the week for each transaction. The Month field will store the month that the transaction took place in the form of a string. The Year field will be used to store what year the transaction occurred also in the form of a string. These fields were chosen to represent a Timestamp database object because every transaction needs some sort of data and time to indicate when a car was purchased for rental. Storing this in a separate table allows more flexibility in the database by allowing for more simpler queries to access information about dates. It also is easier to send this information without the use of search queries by breaking the Transaction table into manageable portions. Finally, the Time field will store string entries that signify the approximate time (e.g. 9:30PM) that a vehicle was rented. A 1-to-1 relationship also exists between the SQL Transaction table and the NoSQL JSON file Rental Agreement. This 1-to-1 relationship indicates that a Transaction develops 1 Rental Agreement JSON file. It is important that Rental Agreement is stored as a key-value NOSQL JSON file because Rental Agreements are usually in the form of documents, so using a key-value type of NoSQL representation allows us to create this requirement. The document form of NoSQL representation allows managers to easily visualize and send Rental Agreements to customer accounts. The Rental Agreement document contains 5 fields: Date Signed, VIN, Mileage at Rental, Total Cost, and Car Lot ID. The reason why we chose these attributes for this document NoSQL representation is because all rental agreements outline the contract that the user makes and signs when purchasing a vehicle; information such as the date the document was signed (in the form of a string) is used to update the status of vehicles as well as ensure that vehicles are returned in a timely manner; the VIN is used to store vehicles by using unique information such as the VIN that act as an identifier in the database of vehicles; the mileage at rental is used to calculate fees and the total price (double value in dollars) which is necessary in the normal operations of a business; the total cost provides both the user and managers with critical information that affect both parties, and the car lot ID which indicates where the vehicle is purchased from and should be stored to ensure proper protocols are met. Finally, a 1-to-many optional relationship exists between the Payment Info SQL table and the Transaction SQL table. This means that a user’s payment information (credit card) can create from 0 to many transactions: one user’s payment information stored on their account may have been used to rent 4 vehicles over a period of time. Meanwhile another user’s payment information may not have been used to rent any vehicles thus far. The Payment Info SQL table contains 6 fields: User ID stores integer representations of the User ID of each payment method. This is relevant to include because users are linked to transactions, so in order to indicate which user was

involved in a purchase of a rental vehicle then a user ID is required. This unique ID also ensures that transactions are properly identified since every user has a unique ID so there will be no confusion between multiple users for a certain transaction. The second field is the Card Number that stores the string representation of each customer's card number. The card number is important to store since it will be used to verify with the bank that the customer has sufficient funds to purchase a rental vehicle. The CVV field stores the 3 digit integer on the back of the customer's card to verify that the customer is the rightful owner of the card. The Type field indicates which type of payment (debit card, credit card, check, etc.) in the form of a field for each entry in the table. The Card Network stores a string for each user payment information, indicating if the card is a Visa or Mastercard. Finally, the Bank Name field stores a string for each user to provide more information about where the user's funds are located or where the transaction should be sent to (Chase Bank, Schoolsfirst, etc...). The Bank Name field is significant because it ensures that the transaction is properly sent to the right bank, and it eases communication between the third parties involved. A many mandatory to many optional relationship exists between the User JSON document collection NoSQL representation and the SQL Payment Info data table. This means that many users (either managers or customers) could have from 0 to many (up to n) payment information objects. We used a document NoSQL database to represent the users of our system. These two users are the customers (the people purchasing the rentable vehicles) and the managers (the employees of the company responsible for updating and managing transaction and inventory across car lots). Each user and manager will inherit attributes such as User ID (data type is a string), User Type (data type is a string), Name (data type is a string), Password (data type is a string), Email (data type is a string), and Phone Number (data type is a string). Every manager and user JSON in the User JSON document collection will share these attributes because the software system requires storing contact information and log in information in order to ensure that user accounts are properly established. As a result, this saves space in our NoSQL database since redundancy in these two groups can be eliminated by inheritance. The Manager JSON of the User JSON collection has extra attributes including Work Schedule and Car Lot ID. Work schedule, which is a PNG image, is critical because there will be more accountability for transactions and it eases the burden for finding the managers working at a current Car Lot ID on a specific day. Car Lot ID is also important because it allows the database management system to keep track of the current managers at a particular car lot location. This helps mitigate problems that arise with either transactions or car inventory in the future. Finally, the Customer object of the User JSON has a Credit Score (Integer), Billing Address (String), Insurance Policy (String), and GPS Current Location (Pair<double,double) attributes. The reason why there are attributes like credit score and insurance policy is to ensure that the use case for checking eligibility requirements for renting a vehicle is met. Therefore this information must be stored in our database

system. A GPS location of type Pair<double, double> is required because the user's location is tracked by the software system to determine the miles they have driven thus far and calculate the total based on how far they have driven and the location they are currently at (since prices may vary depending on where the customer is driving to).

Tradeoffs for Data Management:

Design Decisions For Databases:
We chose to utilize 7 databases to separately store critical information to our software car rental system. 5 of these databases/tables are SQL while the other 2 are a form of document JSON NoSQL database. We decided to use SQL databases for the car lots, the rental vehicles, the transaction, the timestamp, and and the payment info because they each encapsulate different and distinct use cases from the software architecture diagram while ensuring data normalization. First of all, we decided that car lots and cars should be in their own separate database because cars and car lots follow a distinct relationship where one car lot can store many cars. Also, a directory page is required for our software system, so it should be easily accessible for the user to browse cars without the need for long querying times if they were in the same database as a car lot. We realized that because cars and car lots will frequently be used for complex queries, it would be more effective and less time-consuming to keep them separate by indexing the Car Lot IDs in the Cars SQL database. Since many transactions are occurring at a time, using SQL over NoSQL is more beneficial due to the Atomicity and Isolation properties of SQL. If one part of our system is flawed, then the entire system should shut down because private information such as users' payment information, contact information, and more could be vulnerable to malicious people. Also, the Durability property of SQL in this situation would ensure that these key components of customer data are preserved in the event of failure because users would have to create new accounts everytime the database system experienced a failure. Storing important information such as car capacity is necessary in the operation of our software system because the storage, retrieval, and available cars for customers all depend upon the capacity of the car lot. For example, if a car lot is full and a customer is trying to return a vehicle they just rented, then it would be more efficient for the capacity of the car lot to be integrated into the web application and alert the user that they must travel to the next closest car rental lot; this would be better for both parties since if this value was not stored for each car lot, efficiency would be lost since managers would have to manually update the current capacity of the car lot while having to physically deal with customers. For storing the various vehicles in our software system, we opted for an SQL table database because complex queries like filtering transactions and searching databases on a specific input from the user (like model of car, color, type of car, etc). We chose to integrate these fields into the table of our database because our software architecture diagram outlines a use case for searching and filtering vehicles from a wide variety of

vehicles. Thus, SQL would be more efficient for performing queries on already normalized data. Our data is normalized since we have already reduced repeating groups and we have reduced redundancy within the tables themselves. We believed that the attributes in the Cars SQL table provides concise enough information to perform queries based on user input while ensuring that vehicles can be grouped together based on shared attributes. These list of attributes provide more characteristics for the user to select and filter from.

SQL was a straightforward decision for our data management system because the types of data that we needed to store for the car lots, rental vehicles, transactions, timestamps, and payment info can be generalized in data types that are covered by the SQL syntax. The data we store is primarily relational; each car we have in the databases is tied to a car lot which is tied to a specific location, each of which has a myriad of employees. Transactions are made with payments related to a customer, etc. This makes the decision to go with SQL straightforward since the syntax is inherently relational. There are a few aspects of our system where SQL falls short of what we need. For example, when we're storing document based data, like information concerning user profiles. We define two types: manager and customer, they each have similar attributes but each have their own unique attributes. We decided to organize this type of data using JSON files and collections since SQL isn't capable of unique attributes to the same type. It also makes conceptual sense to represent users as documents. On a similar level, rental agreements work best in a NoSQL structure because that type of document often has blocks of text that wouldn't make sense to make it a String type in an SQL database. It would also make organizational sense as we're storing literal documents in this case, representing data that is already in an SQL, but with the customer in mind.

Security Analysis:

There are numerous threats and vulnerabilities inherent in our software system. There is a potential device that could allow hackers or unauthorized personnel access to our software system and its databases. Since our website requires a server to process requests and maintain databases, a person with some sort of USB dongle could tamper with the server. This USB could infect our server with malware, preventing the server from performing requests and resulting in the shutdown of our server and website. This malware could also perform surveillance over the entire network of car lot networks. This means that communication between LANs and users across the country could be exposed. This means that personal and private information such as credit card information, names, social security numbers, and more could be at risk if the hacker has access to the server hosting the website. This information is strictly confidential under the CIA categorization, so it should be stored as securely as possible. Although the

information may be less accessible for users and managers, it ensures that outside attackers cannot easily obtain this data stored in the databases. Hackers could also use IP address spoofing to deceive the system into thinking this “user” is a normal car customer. This would be problematic for our system because it would expose the IP address of the server, leaving it vulnerable to distributed denial of service attacks. This could mean that a hacker could establish botnets to send large amounts of packets that disrupt and interrupt the server’s requests to users. This would mean that information such as cars (and their description/status), car lot locations, and other web page information that relies on availability under the CIA categorization would be unavailable due to this loss of service and functionality created by the DOS attack. The first countermeasure for these vulnerabilities would be to install trusted platform modules (TPMS) so that access devices like USB dongles would not be able to access the server without proper detection. These devices utilize microprocessors so that hardware can detect and authenticate unauthorized devices from accessing data or networks. The next countermeasure would be to install and implement more secure firewalls so that traffic between the server, the LANs at each car lot, and the Internet can be regulated. This would ensure that the server could maintain operation in the event of a denial of service attack since the packets could be filtered by the firewall, ensuring that confidential user information (like payment data, SSN, addresses) and data that relies on availability (directory of cars, location to nearest car lot, etc.) would be available despite this attempted attack. Finally, in the event a hacker has access to a customer’s login information, 2FA could mitigate the risks of unauthorized access by sending a code to a user’s phone number (which the hacker does not have access to). If the user is notified of a data leak and that their information was stolen, they would be able to change their credentials and continue to use the website.

3) Life Cycle Model

The life cycle model that would best suit our project is the evolutionary prototype model. The evolutionary model is so simple yet effective for our software system because most of the time of the development phase will be towards implementing an initial prototype and constantly improving it to make it more scalable when it comes to storing data of all types. Many models like the spiral model and the waterfall assume that each phase of the development cycle will take roughly around the same amount of time. In our case, the software system that we are building is a web application that is supposed to support all devices. Building a website and integrating the various databases to store and manipulate customer, payment, and car data requires significantly more time than designing or obtaining requirements from the client. Since the project is clear yet simple in its design, a spiral model would not be necessary since it would be too complex for such a self-explanatory system design. The emphasis on iterative development when it comes to producing a reigning a prototype in the evolutionary prototype model reflects

the development of our BeAvis car rental system because constantly adding new SQL and NoSQL databases along with the expansions and creation of new car lots necessitate the creation of a robust and scalable system that can handles lots of requests and data simultaneously. To ensure that this system can handle these requirements, constant iterative development and testing is required. The first phase of our development according to the evolutionary prototype model is the initial concept. This would only take 1 week to complete since all of the user requirements emphasize a web application that is both universal to all devices and scalable in terms of handling data (whether it be user information, rental vehicles, or payment data). The next phase of the development cycle is the design and implementing the initial prototype. We would first start creating our use case, SWA, UML, and data management diagrams because this establishes the framework for the functions/operations on our data that need to be considered. This would take roughly 3-4 months to complete due to complexities involving what type of data we are storing, the design of the website (front end), the server side request handling (backend), the functionalities and use cases of the website, and the various APIs that would need to be integrated. Some issues that could arise are bugs and failures in code from changing or adding new webpages, implementing new APIs for transactions or integrations with other applications (like GPS and payments), or creating personalized ports to other devices could create unexpected complexity and events that weren't taken into account in our design. Additionally scalability would be a potential issue because there are large amounts of data that need to be stored; when new car lots are constructed in the future, we want to make sure that we are developing a system that can both store these large amounts of data (each car lot stores numerous vehicles and each vehicle has many characteristics and specifications) while being able to make queries in an efficient manner to help maintain customer satisfaction. After implementing and creating an initial website design that is able to store at least perform all of the functions required to rent a vehicle based on the design documents, the next phase would be to refine the prototype until requirements are met and the web application can handle requests like renting and returning vehicles. This would take roughly 3-6 months to complete as this would involve debugging and creating unit tests to ensure that our software system can handle edge cases and other unexpected instances that can happen. Finally, completing and releasing the prototype would only take a few weeks since it does not take too long to create a running website. An issue that could arise at this stage would be that certain features or code cause issues for the server hosting the website; this could affect performance or compatibility across devices. One example could be that the website runs perfectly normally without issues on a computer using a Google search engine while a person using the app on their iPhone cannot see certain images or perform certain functions. Incompatibility is an important issue that needs to be addressed if prevalent after release.

Here is a life cycle model that accurately reflects the development cycle of our system. It takes inspiration from the evolutionary prototype model and addresses its pitfalls through a greater emphasis on comparing results with requirements, involvement with the client, and testing: