BeAvis Car Rental Software Design Specifications

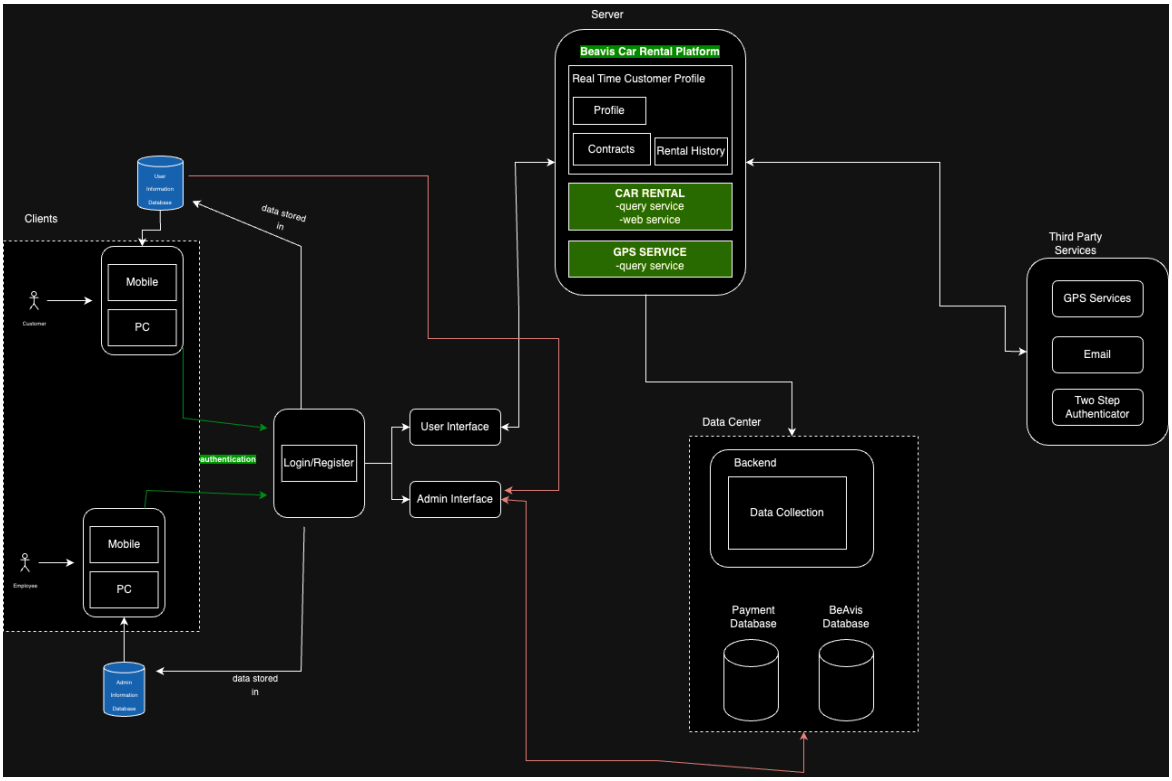
Prepared by: Caroline Wilkins, Yearly Gonzalez, Ken Robinson

System Description

This document contains the necessary information regarding the BeAvis Car Rental system in order to build the client's preferred product. The text below will include the necessary components to construct the system itself. This document provides an Architectural diagram of the major system components, a UML Class Diagram, a detailed description of said diagram, an approximated timeline for the development process, and a general software development plan. The client, BeAvis Car Rentals, has requested the following system with the intention of creating a more convenient and accessible process for their customers to rent vehicles with. The BeAvis Car Rental system is a multi-platform program that allows customers to complete the car rental process, locate BeAvis establishments, and prepare to both pick up and drop off their reserved vehicle. During the usage of the BeAvis Car Rental system, customers will be able to virtually complete the entirety of the car rental process. Customers will be able to virtually view, book, and locate their BeAvis rental car in a single space through the utilization of this system.

Software Architecture Overview

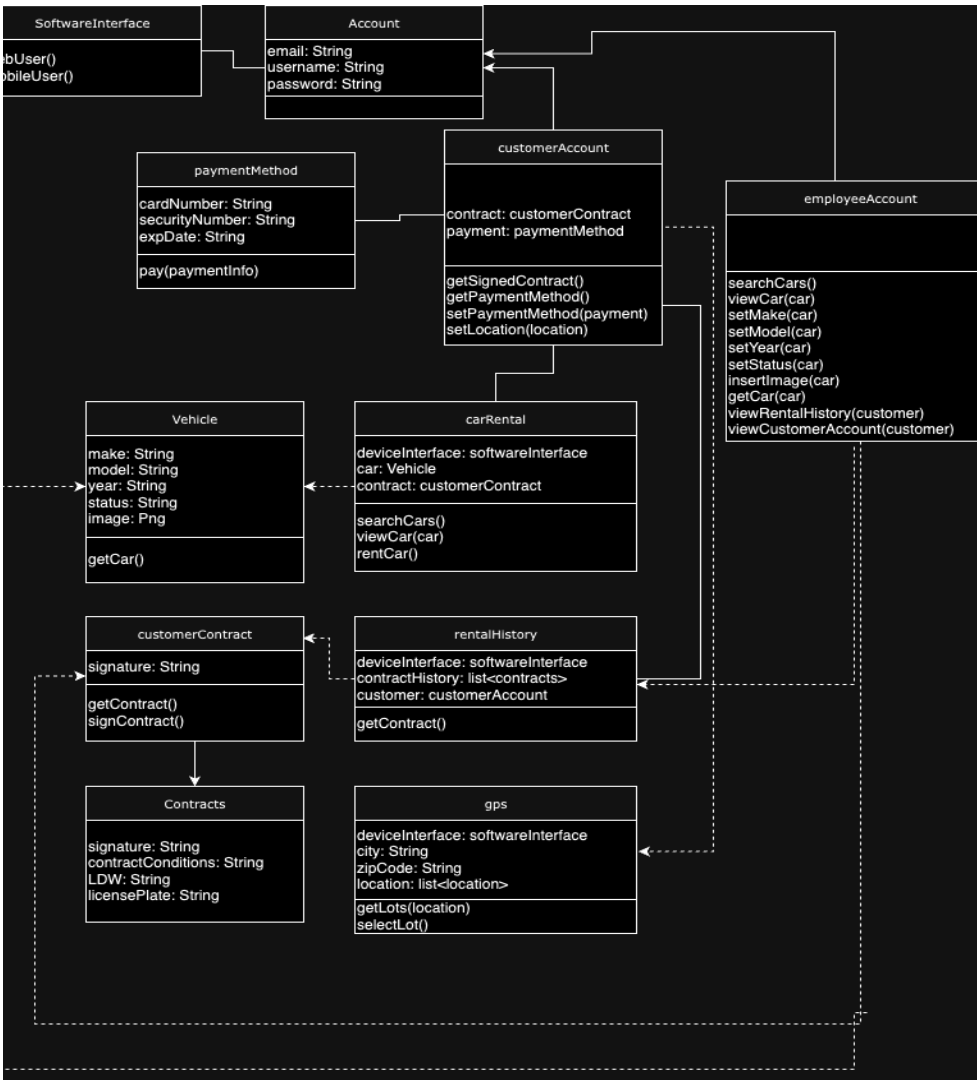
Software Architecture Diagram



Software Architecture Description

The included software architecture diagram represents the software system components of the BeAvis Car Rental system. The flow follows a left to right pattern beginning with the potential clients, those being customers and employees each with the option of PC or mobile. The clients are both connected to the login or registration component through authorization. There are two databases that store client account information, one is designated for customers and the other for employees. Data collected from the authorization is stored in one of the two account information databases with respect to the client's credentials. Due to the difference in functionality in regards to client types, there are two components for software interface. Following the path of a customer client, the interface both sends and receives data from the BeAvis platform itself. This server consists of several components in order to fulfill all necessary operations. The real time customer profile component includes the various elements needed to fulfill all customer rental management functions. It stores the data for customer profile, rental contracts, and rental history. The remaining two components of the BeAvis server are the car rental functions and GPS functions. The former will consist of query and web service operations whilst the latter requires only that of query. Outside of the user interface, the server also connects to the data center and third party services. The latter includes gps services, email and two step authenticator connectivity. The former collects the data from the system, it consists of the backend for data collection, the BeAvis database, and a database holding customer payment methods to adhere to system requirements of security. This data center component also has a two way connection between itself and the administrator interface to allow employees to view and alter system information. Finally, the admin interface has data sent to it from the user information database to allow the viewing of customer account data when necessary.

UML Diagram



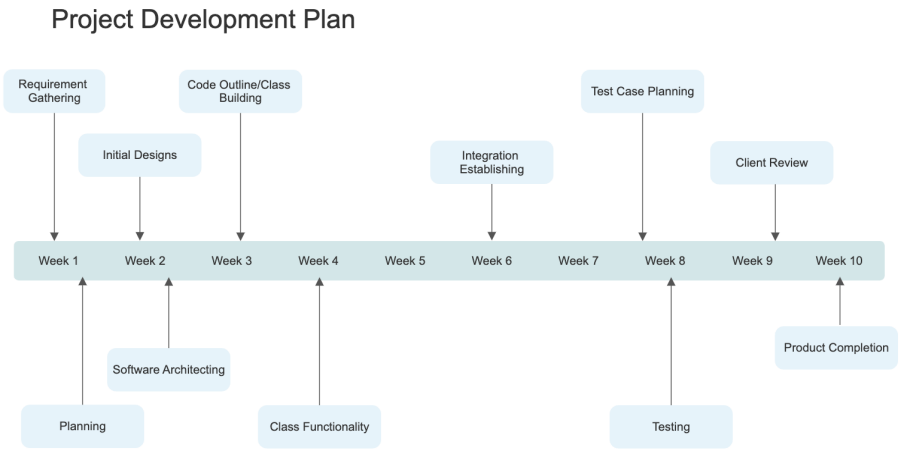
UML Class Overview

The included figure is a UML diagram for the BeAvis system. The diagram begins with the **SoftwareInterface** class. This class is responsible for whether the software will boot the web or mobile version of the application. It includes no operations and two attributes being that of *webUser()* or *mobileUser()*. Once the device version is set, the diagram follows the user through account creation or login. There are three account classes in the software: Account, customerAccount, and employeeAccount. The **Account** class is the general class for the accounts. Its attributes consist of the various components necessary to create and maintain an account through the system, those being an *email*, *username*, and *password*. Each attribute here is a string data type to be inherited by the other two account classes. From there the customerAccount and employeeAccount are specialized classes that deal specifically with customers and employees accounts respectively. The **customerAccount** class includes two attributes to coincide with its four available operations. The *contract* attribute is a customerContract object whilst the *payment* attribute is a paymentMethod object. These components are necessary for the functionality of the getSignedContract(), getPaymentMethod(), setPaymentMethod(payment), and setLocation(location) operations which allow for the obtaining of completed customer contracts, establishment and viewing of payment methods, and

setting of customer location. The **employeeAccount** class is our most complex class as the employee should be able to access most information in the software as well as modify it. There are no attributes for this class, however, it includes the ten following operations: searchCars(), viewCars(car), setMake(car), setModel(car), setYear(car), setStatus(car), insertImage(car), getCar(car), viewRentalHistory(customer), and viewCustomerAccount(customer). The **paymentMethod** class is the object that holds all the relevant payment information that is saved in the user's account within the software. It includes three attributes, *cardNumber*, *securityNumber*, and *expDate*, all of the string data type. The singular operation of this class is the function to complete a payment, or pay(paymentInfo). The **carRental** class is responsible for all functions related to the renting of cars themselves, it can be considered the core of the software. There are three components within this class, those being, *deviceInterface*, *car*, and *contact*. These attributes are objects of the softwareInterface, Vehicle, and customerContract classes respectively. The operations for this class, or searchCars(), viewCars(car), and rentCar(), allow users to search through all available cars, view a singular car, and begin the renting process. The **vehicle** class will be responsible for creating objects for every individual vehicle, including all the details of the vehicle. It is largely composed of attributes rather than operations, with its only function being that of getCar(). The components of this *class*, *make*, *model*, *year*, *status*, and *image*, are almost all string variables that cover general information of the vehicles with an exception of the *image* attribute being of png datatype. The **rentalHistory** class is responsible for creating an object that will hold the history of the customer's contract history. It has an individual operation of getContract() and three attributes begin, *deviceInterface*, *car*, and *contract*. The aforementioned variables are data types softwareInterface, vehicle object, and customerContract object respectively. The **customerContract** and the **Contracts** classes are both related to the contracts that will be signed with the customers, the difference being the Contracts class itself will be responsible for creating the contracts while the customerContract will create an object that is responsible for holding the proof of the users agreement with said contract. The customerContract class has a single attribute, *signature* of the string data type, to allow for contracts to be signed. It includes two operations, those being getContract() and signContract(). The Contracts class lacks any operations, instead having four string attributes including *signature*, *contractConditions*, *LDW*, and *licensePate*. The last class is the **gps** class which is responsible for creating objects that hold the gps data for vehicles and lot locations in order to find the nearest lot to the customer. Its four attributes consist of a softwareInterface object named *deviceInterface*, two string variables named *city* and *zipCode*, and a list of the two variables called *location*. There are only two operations for this class with those being getLots(location) and selectLot().

Development Plan and Timeline

Project Timeline



Timeline Description

The above figure details a general software development timeline. The development of the BeAvis software is to be divided into eleven tasks over a period of ten weeks. The first task to be completed is **requirement gathering** to be designated to the *business analyst team*. They will obtain a list of software components and needs from the client within half of a week. The latter half of the first week is set for **planning** by the *project manager*. A general development plan for product development is to be created including risk analysis, scope, and testing. The former half of the second week focuses on the planning of **initial designs**. The *design team* will create proposals for design UX/UI and settle on general design direction in accordance to requirements given by the client. The latter half of this week revolves around **software architecting**. The *software architect* will create a software development plan including software architecture and unified modeling language diagrams to be used by the development team. Over the entirety of the third week the *software developers* will be tasked with **code outline and class building**. This covers the implementation of a general version of each class. The following week will stretch into **class functionality**, also designated to the *software developers*. The development team will implement attributes and operations to each class in accordance with the architecture diagram at this time. Once completed, *software developers* will move to a two week period of **integration establishing**. This brings the tasks of securing necessary integration with external systems and allowing for connectivity with all noted third-party applications. Over the first half of the eighth week, *QA engineers* will be in charge of **test case planning**, or establishing a form of testing and developing necessary test cases. The task of **testing** itself will fall onto the *testers and software engineers* as they spend the latter half of week eight as well as the former half of week nine running through various test cases. This is to cover the conducting of unit, integration, and system tests, completing static code analysis, and fixing bugs as they are found. Once the system itself has been thoroughly tested, the remainder of the ninth week will be spent with the *business analyst team* as they conduct a **client review**. This will consist of the presenting of the finished product to the client for assessment of features and performance. The final week of this project is to be reserved for **project completion** as a whole. All teams will work together to deploy the current system to the production field. Members will monitor and fix bugs with user feedback as well as plan for future improvements.

Software Development Plan

The BeAvis Car Rental Software is being developed to bring a more user friendly dynamic to the prior car rental system in place at BeAvis. Prior to the development of this software, BeAvis users experienced difficulty in accessing the car rental information, as well as navigating the different steps in the car rental process without in-person assistance. The proposed car rental software system resolves these issues on the user end by providing a simple virtual interface that allows the user to view, reserve, and locate their rental car in a single designated space. In increasing visual simplicity on the user end, the software system successfully addresses the issues that the prior rental system posed.

The main development tasks required of this system include design, coding, integration, and testing. The design development process entails creating a UX/UI design that aligns with the views of our clients at BeAvis. Our design team works closely with our software architects to create the software architecture required of our proposed system. The first and second phase of our design process was completed by members of our design team. The next phase of development, coding, is outlined by the development and functionality testing of different class implementations. The UML diagram created can be referenced as a visual to the coding phase of the rental system development process. The integration of these proposed architecture and coding systems to be implemented in this software will be closely tested with the pre-existing hardware and external systems that BeAvis uses at their car rental centers. The implementation phase of development will be passed along to members of our software development and engineering teams prior to the testing and client review phases. Finally, our testing phase of development will require cross-functionality testing between the elements of the car rental software architecture and coding scripts. All teams will collaborate in daily meetings to ensure software efficiency and confer about elements of the software that may need to be debugged or enhanced.

The development will closely follow a predetermined project schedule, as demonstrated by our project development timeline. Before the true stages of development begin, our software development team will take approximately half of a week to gather system requirements which will be followed by project manager review, taking another half week to conduct risk analysis, scope, testing, and development planning. After a week of planning and creating proposals for UX/UI design, the development process will begin.

The BeAvis Car Rental Software development team will utilize the evolutionary prototyping model of software development. Our clients have provided us with the initial concept, design, and implementation of the prototype through their system requirements documentation. As developers, we will create and refine the software architecture and coding scripts until they run in tandem effectively. In working closely with our clients to implement client feedback, the car rental system prototype will not be released until our development team and clients reach a formal agreement on the software's release. The evolutionary prototyping model promotes quality assurance and will allow us to put forth high-quality software.

Development of the new rental software will require a number of resources that will need to be funded on the client-end. Databases for payment information, administrative information, and vehicle information will need to be set up electronically for the management and storage of user and corporate data. Beavis clientele have the option of choosing the most efficient and cost-effective platform for doing such. A financial agreement will also need to be set in place for third-party service providers.

Risk analysis will be conducted on all features throughout the development process. Our main concern lies in information security. Dealing with personal payment and identification information on the user-end will require a high-level of security and specific access-points for clientele. Two-factor authentication will

mitigate this challenge on the user-end. For software administrators, only a select few members of the Beavis corporate team should have access to the financial databases set in place. Limiting access and verifying one's identity should encourage and instill information security within the system.

Throughout the development process we will track different metrics to monitor system performance. Time complexity of our coding scripts will be reported after major changes are made to the script in order to track system performance and run-time. Each department within the software development team will participate in biweekly progress meetings to bring potential challenges to light, and to celebrate successes of the evolving software program. As a result of performance tracking, reports will be sent over to Beavis corporate to communicate progress after each of these biweekly meetings.