Car Rental System Software Requirements & Design Specification

Prepared by: Isabella Lawlor, Khoi Nguyen, Arthur Pasquinelli

October 9, 2024

Contents

1 Introduction and Overview

1.1 Purpose

This car rental system is a solution designed to make renting cars more convenient and efficient by offering both a website and mobile application that cater to customers and employees. The purpose of the system is to provide customers with an efficient platform to browse, select, and rent cars from the nearest location. The system will have location services to provide customers with directions to the nearest rental location, the cars available at that location, and they can complete the entire renting process from their phone. Additionally, employees will be able to manage customer data, contracts, and car inventory. This system is needed to make the car rental process more efficient for customers and employees.

1.2 Contents of This Document

- User Requirements: This section of the document includes information about the use, from the user perspective, of the application. It includes customer and employee functionalities and other constraints.
- Functional Requirements: This section describes the software product's capabilities and attributes. It explains the uses and interactions from both the customers and employees.
- Non-functional Requirements: This section addresses the qualities and constraints of the system such as security, usability, reliability, and portability. It explains how the system should behave during certain conditions.
- Other Considerations: This section addresses current constraints and risks along with possible solutions to these issues that could be implemented in the future.

2 User Requirements

2.1 User Types and Interaction

Customer Functionalities:

- Account Creation: Users will be able to create an account with their email and a password. They will get a verification code sent to their email to activate the account. Users will also be able to store payment information in their account for future use.

- Browse Available Cars: Users will be able to browse available cars and filter based on make, model, color, year, mileage, and location.
- Book and Manage Rentals: Users will select the car they would like to rent, pay using a credit/debit card or Apple Pay, then complete a rental contract.
- Rental History: Users will be able to view their past rental history through their account. The rental history will include type of car, duration of rental, and location.
- Location Services: Users can use location services to find nearby rental locations.
- Contracts: Users will be able to view rental contracts associated with their past rentals.

Employee Functionalities:

- View Customer Data: Employees will be able to view rental history information including car type, rental location, rental duration, and car condition.
- Inventory Management: Employees will be able to update the status of cars, including cars that require maintenance.
- Access Contracts: Employees can view and manage rental contracts.

2.2 Context and Constraints

- User Interface: The application will have a neat, user friendly interface on the web and mobile platform. It should be easy to navigate, quick to access cars, and clear and easy to understand contract details.
- Hardware Requirements: The application will be accessible on standard mobile devices and desktop computers
- Software Constraints: The system will use standard technologies for both the web and mobile app versions.
- Security: Contracts and payment information will be stored in a separate, secure database.

3 Functional Requirements

3.1 Uses and Interactions

Customer Use Cases:

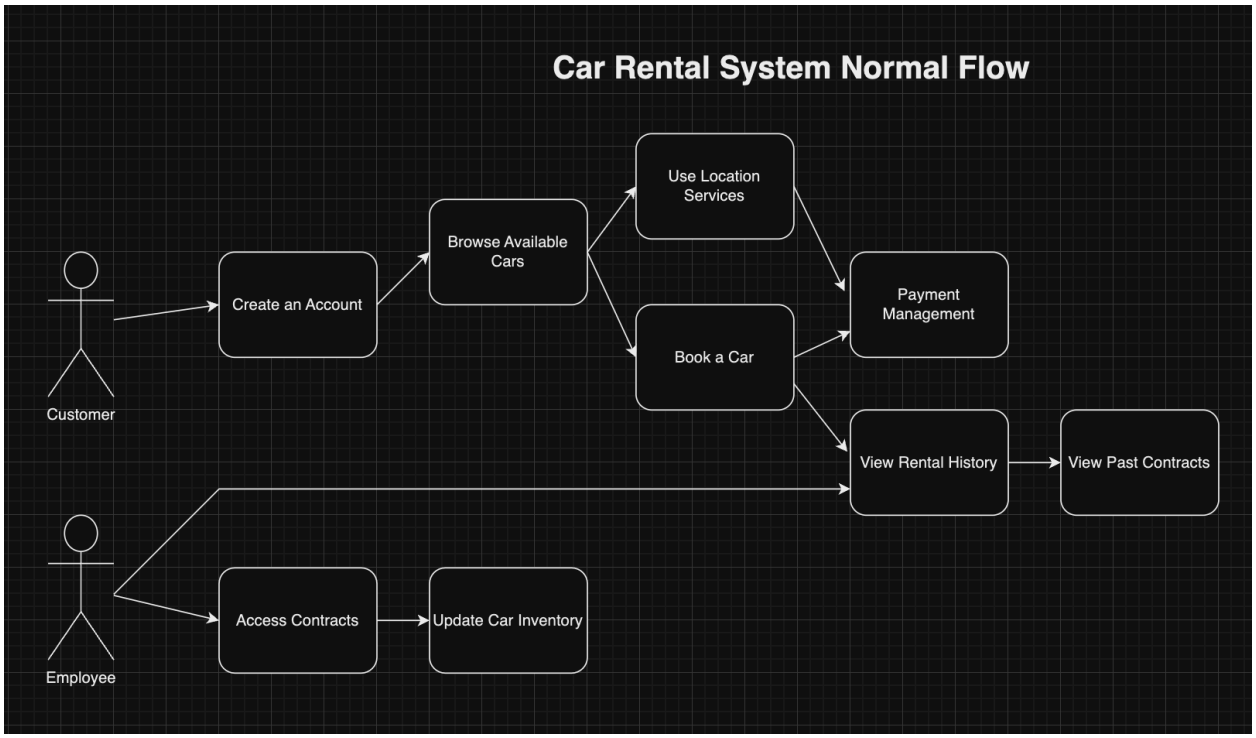
- Create an Account: Users will sign up using an email address and password. A verification code will be sent to the email to complete registration. Users will be able to add payment details to their account.
- Browse Available Cars: Customers can filter through the available cars based on location, make, model, year, color, and mileage.
- Book a Car: Users can choose a car, select the dates they want to rent, and then will confirm the rental after filling out the contract.
- View Rental History: Customers can view past rental cars, including car details, rental duration, and rental location.

- Payment Management: Users can add or change payment information in their account. Acceptable payment methods include: credit cards, debit cards, and Apple Pay.
- Use Location Services: Users can view the nearby locations and get directions to a selected rental site.
- View Past Contracts: Customers can view past contracts they've had when renting cars on their account.

Employee Use Cases:

- Access Contracts: Employees have access to all customer contracts to view and manage when needed.
- View Rental History: Employees can view a customer's rental history which will include rental dates, car information, contracts, and car condition.
- Update Car Inventory: Employees can add, remove, or update cars from the system. They can also mark cars that require maintenance.
- View Past Contracts: Employees can view and manage personal rental contracts.

3.2 Use Case Diagram



4 Non-functional Requirements

4.1 Security Requirements

- Data Security: Payment information and rental car contracts will be stored in a separate database.
- User Authentication: The platform will have email verification to ensure account security.
- Employee Access Control: Employees will have restricted access to important information based on their roles.

4.2 Usability

- User Interface: The platform will have an easy-to-use interface for the mobile app and the website.
- Performance: The system will be able to handle large amounts of users and car inventory and still perform efficiently.

4.3 Reliability

- Data Backup: Regular data backups will be implemented.

4.4 Portability

- Cross-platform Capability: The system will support major web browsers and mobile operating systems.
- Data Portability: The platform will allow seamless transfer of data between devices and platforms for both customers and employees.

5 Other

5.1 Risks

- Data Breach: There is a risk of sensitive data being compromised if security measures aren't enforced.
- Scalability Issues: As the platform grows, the system must be able to handle the increased number of users and transactions.

5.2 Constraints

- No Cancellations/Refunds: The system will not handle rental cancellations or refunds.

5.3 Future Changes

- Cancellations and Refunds: These functionality may be added in future versions of the platform
- Additional Payment Methods: Support for other payment methods may be introduced.
- Loyalty Program: A customer loyalty program with a points system could be implemented. Customers could accumulate points with each rental which could be exchanged for discounts or free upgrades.

- Mobile Notifications: Push notifications on the mobile app can be implemented to remind customers of upcoming promotions, upcoming reservations, discounts, booking confirmation, and vehicle return deadlines.
- Multilingual Platform: The system can be expanded to support many different languages to bring in international users.
- Car Delivery: The software could add a delivery service, for an extra charge, where employees will deliver a customer's rental vehicle to their current location.

Software Design Specification (6)

6.1 Developer Description

Having gathered the general requirements for the car rental system from the client(s), the development team can now work internally using a blueprint of the implementation. Using the requirements specified, the following design specification of the system will outline the planned architecture for the interactive services, servers, hardware, and storage included in the system. In addition to an architecture scaffolding, the specification provides an elementary blueprint of the system's software implementation, according to UML standards. Finally, the specification plans implementation, assigning tasks and their overall development to specific team members, along with time estimates. This creates a timeline for system implementation, which structures execution so that the team can transition from product design to development.

Following Document Structure:

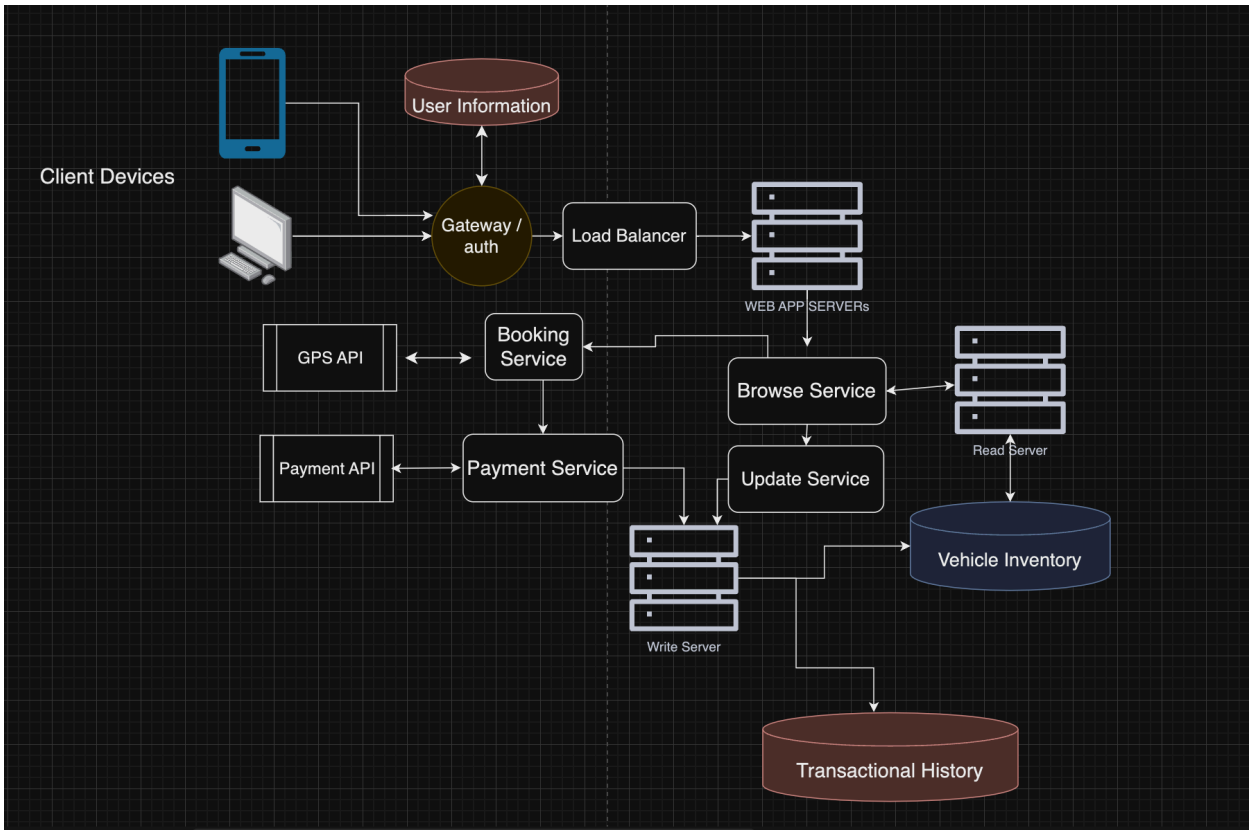
- Architecture
- Software UML
- Team Responsibilities; Timeline

System Overview

The system refers to the same requirements described in the preceding requirements specification. The client(s) are pursuing a mobile web application service to streamline the processes of a car rental company. Customers will be accessing the system from their own devices, as will employees from company devices, from the application in their web browser or on their mobile device. From their device, they will connect to the server under an account type, and be able to therefore perform their associated actions with the services provided. Customers will browse stored inventories, specify cars in their transactional bookings, navigate to the lots using integrated GPS navigation, and finalize rentals using integrated payment services. Employees will directly manage the local inventories, car information, user accounts and bookings, and access stored business history.

6.2 System Architecture

Software Architecture Diagram (Below)



SWA Diagram Description:

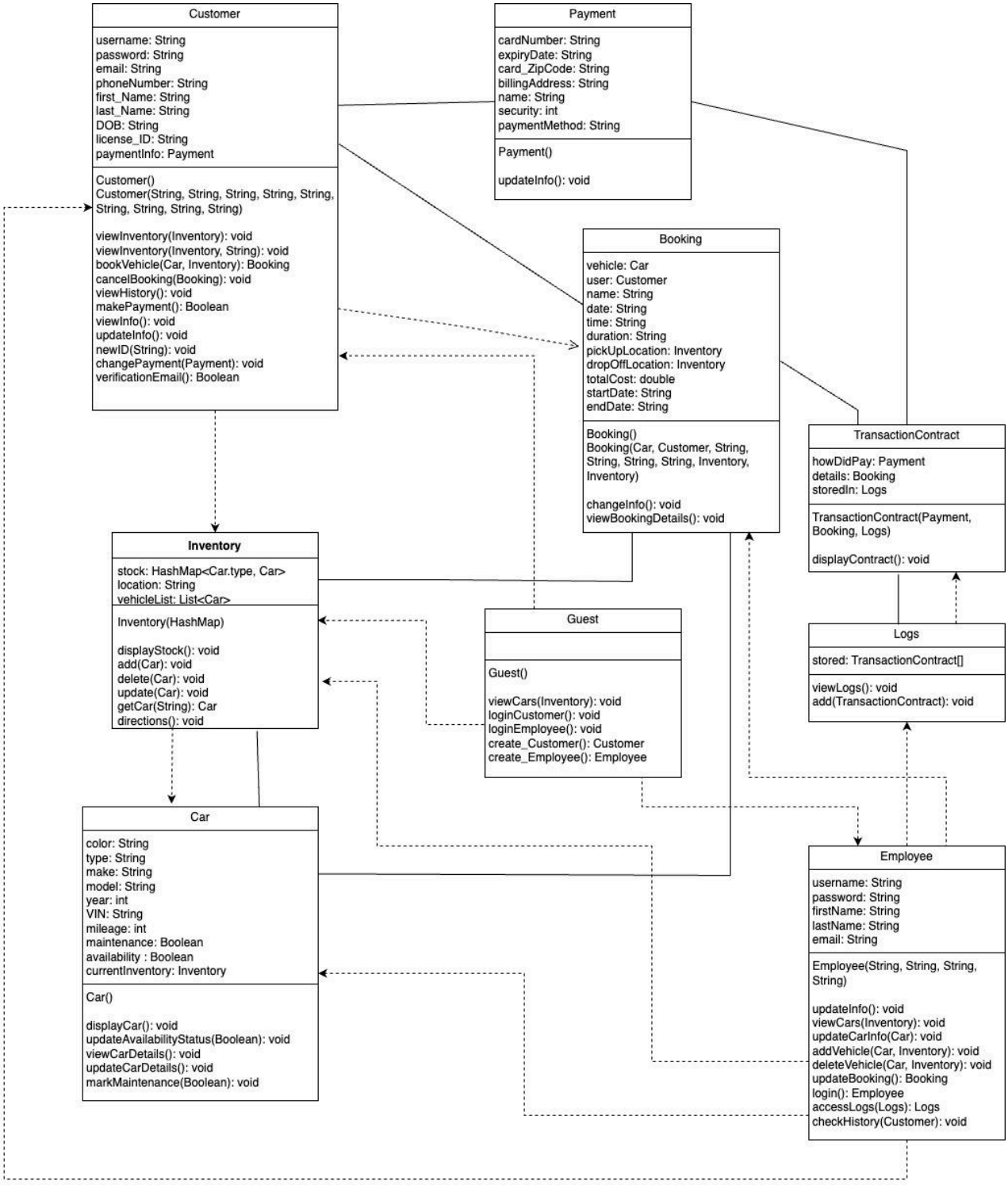
The architecture diagram begins with the client devices accessing the system, which supports both mobile and desktop platforms. Client devices proceed through the gateway authentication, a service which retrieves user information from our User Information Database to verify whether the client is a guest, employee, or customer, signing them in or creating an account accordingly. Upon verification, the authenticator continues to the load balancer, which monitors server traffic so that device speeds are balanced in use of the Web App Servers. The

Web App Servers keep the service online and accessible, and would then make use of the central Browsing service for all inventory services moving forward.

For reading, the browsing service connects to a Read Server which pulls information from the Vehicle Inventory for display. After observing current inventory from the central Browsing service, employees can update said inventory using the Update service. This service connects to the Write Server, which changes the information in the Vehicle Inventory Database directly.

To follow through with booking a rental, the Browse service connects to a Booking service. This service is connected to an external GPS API for directions to and from rental sites. Additionally, the Booking Service connects to the Payment Service to follow through with transactions. The Payment Service uses an external payment API for an outsourced card transaction. Successful payment finalizes the booking; the Payment Service connects to the Write Server, which updates the Transactional History and Vehicle Inventory Databases accordingly.

6.3 UML Depiction



UML Diagram (Above)

UML Description:

Customer Class:

Overview

The Customer class represents an instantiable customer in the system. It stores personal information for an account, like username, password, contact details, and payment information. Customers can view vehicle inventory, book and cancel rentals, view their history, and update their profile and payment details. They can also verify their account via email.

Attributes:

- username: String - customer's username for login
- password: String - customer's password for authentication
- email: String - email address of the customer
- phoneNumber: String - customer's phone number
- first_Name: String - customer's first name
- last_Name: String - customer's last name
- DOB: String - customer's date of birth
- license_ID: String - customer's driver's license number
- paymentInfo: Payment - Customer **has a** field of Payment type, handling payment details by storing the preferred transaction method

Constructors:

- Customer() - a default constructor that creates a new Customer object
- Customer(String, String, String, String, String, String, String, String, Payment) - constructor that allows you to create a Customer object with given values; **uses a** Payment object

Methods:

- viewInventory(Inventory): void - allows the customer to view the entire inventory of vehicles, **using** the passed in Inventory
- viewInventory(Inventory, String): void - allows the customer to filter the inventory based on specific criteria
- bookVehicle(Car, Inventory): Booking - allows the customer to book a vehicle from the inventory, **using** a specific Car object as an argument
- cancelBooking(Booking): void - cancels a customer's existing booking ; **uses** a provided Booking object for deletion
- viewHistory(): void - displays the customer's past bookings
- makePayment(): Boolean - makes a payment **using** their payment field, and returns boolean based on the transaction success

- viewInfo(): void - allows the customer to view their account details
- updateInfo(): void - allows the customer to update their account details
- newID(String): void - allows the customer to update their ID details
- changePayment(Payment): void - allows the customer to update their payment information, setting their preferred payment field to the new, updated payment method; **uses** Payment object as argument
- verificationEmail(): Boolean - sends a verification email to the customer and returns a Boolean indication whether it was successful

Payment Class:

Overview:

The Payment class contains the customer's payment details, including card information, billing address, and payment method. It allows a customer to update their payment details.

Attributes:

- cardNumber: String - customer's credit card number
- expiryDate: String - expiration date of the card
- card_ZipCode: String - billing zip code
- billingAddress: String - address to which the card is registered
- name: String - name of the cardholder
- security: int - the card's verification code
- paymentMethod: String - type of payment method (debit, credit, paypal)

Constructors:

- Payment() - a default constructor that creates a new Payment object

Methods:

- updateInfo(): void - allows customer to update their payment information

Booking Class:

Overview:

The Booking class represents a rental booking made by a customer. It includes details regarding a reservation, such as the vehicle, customer information, booking time, duration, and locations for pickup and drop-off. This class also allows modification of booking details and viewing booking information.

Attributes:

- vehicle: Car - Booking objects **have a** Car object, specifying the vehicle the customer reserved
- user: Customer - Bookings **have a** Customer object, storing which customer made the booking
- name: String - name associated with the booking
- date: String - date of the booking
- time: String - time the vehicle is scheduled to be picked up
- duration: String - duration of the booking
- pickUpLocation: Inventory - **has an** Inventory field; the location from which the vehicle will be picked up
- dropOffLocation: Inventory - **has a** secondary Inventory field; the location where the vehicle will be dropped off
- totalCost: double - total cost of the booking
- startDate: String - start date of the booking
- endDate: String - end date of the booking

Constructors:

- Booking() - a default constructor that creates a new Booking object
- Booking(Car, Customer, String, String, String, String, Inventory, Inventory) - constructor that allows you to create a Booking object with given values

Methods:

- changeInfo(): void - allows the customer to modify details of an existing booking (date, time, location)
- viewBookingDetails(): void - displays the full details of the booking (vehicle, duration, totalCost, etc.)

Inventory Class:

Overview:

The Inventory class manages the stock of rental vehicles available at specified locations. It stores and manages the collection of cars by type and location. This class's methods include adding, removing, updating cars, giving directions, and displaying available inventory.

Attributes:

- stock: HashMap<car.type, Car> - **uses** Car objects; map that stores vehicles categorically by their Car type field, allowing for efficient lookups and filtering
- location: String - physical address of the lot location
- vehicleList: List<Car> - list of all cars in the inventory; **uses** Car

Constructors:

- Inventory(HashMap<Car.type, Car> stock) - constructor that initializes the inventory

Methods:

- displayStock(): void - displays the current inventory of cars
- add(Car): void - adds a new car to the inventory
- delete(Car): void - removes a car from the inventory
- update(Car): void - updates information about an existing car
- getCar(String): Car - **uses** Car object; retrieves a Car from the inventory based on its type
- directions(): void - provides directions to the location of the car you booked

Guest Class:

Overview:

The Guest class represents users who haven't logged in or created an account yet, enabling them with limited functionality. Guests can browse available cars and create either customer or employee accounts. They can also log in to an existing account of Customer or Employee types.

Constructor:

- Guest() - a default constructor that creates a new Guest object

Methods:

- viewCars(Inventory): void - allows guests to view available cars in the inventory, **using** Inventory and therefore Cars
- loginCustomer(): void - prompts the guest to login as a customer; **uses** Customer
- loginEmployee(): void - prompts the guest to login as an employee; **uses** Employee
- create_Customer(): Customer - allows the guest to create a new customer account and returns a Customer object after creation; **uses** Customer
- create_Employee(): Employee - allows the guest to create a new employee account and returns a Employee object after creation; **uses** Employee

Car Class:

Overview:

The Car class represents the actual rental vehicles. It stores details such as the car's type (SUV, Sedan), make, model, year, mileage, availability, and maintenance status. It allows employees to update the car's details and mark it for maintenance when needed.

Attributes:

- color: String - color of the car
- type: Spring - type of car (SUV, Sedan, Truck, etc.)
- make: String - manufacturer of the car (Toyota, Ford, Kia, etc.)
- model: String - specific model of the car (Camry, Bronco, Sorento, etc.)
- year: int - year the car was made
- VIN: String - the car's license plate information
- mileage: int - the number of miles on the car
- maintenance: Boolean - indicates whether the car currently needs maintenance
- availability: Boolean - indicates whether the car is currently available
- currentInventory: Inventory - **has an** Inventory field to specify which lot the car is located within

Constructors:

- Car() - a default constructor that creates a new Car object

Methods:

- displayCar(): void - displays the car that the user selected
- updateAvailabilityStatus(Boolean): void - changes depending on whether or not the car is available
- viewCarDetails(): void - returns a String showing the car's details
- updateCarDetails(): void - allows employees to update details about a car
- markMaintenance(Boolean): void - checks if maintenance is needed on a car

Employee Class:

Overview:

The Employee class represents employees of the rental service. Employees can update their profile, manually manage the inventory by adding, updating, or removing vehicles, access logs, and view customer booking history.

Attributes:

- username: String - the employee's username for their account
- password: String - the employee's password for their account
- firstName: String - the employee's first name
- lastName: String - the employee's last name
- email: String - the employee's email address

Constructors:

- Employee(String, String, String, String, String) - a constructor that initializes the employee with the login details and name

Methods:

- updateInfo(): void - allows the employee to update their personal information
- viewCars(Inventory): void - **uses** Inventory; allows the employee to access specified inventory to view cars within
- updateCarInfo(Car): void - **uses** Car; allows the employee to update information about a specified car
- addVehicle(Car, Inventory): void - **uses** Car and Inventory; allows the employee to add a specific vehicle to a specific inventory
- deleteVehicle(Car, Inventory): void - **uses** Car and Inventory; allows the employee to delete a specific vehicle from a specific inventory
- updateBooking(): Booking- allows the employee to update information on a booking
- login(): Employee - authenticates the employee's username and password to log them in
- accessLogs(Logs): Logs - **uses** the Logs class; allows the employee to view logs of transaction and contract history
- checkHistory(Customer): void - **uses** Customer; allows the employee to view the specific customer's past booking information

TransactionContract Class:

Overview:

The TransactionContract class represents a record of a booking transaction. It stores the payment details, booking information, and the associated logs. This class allows for viewing the contract details for reference in the future.

Attributes:

- howDidPay: Payment - Contract **has a** Payment object, which used by the customer for the booking
- details: Booking - **has a** Booking; details of the booking (car, renter, rental time, etc.) are associated by keeping the entire Booking object within the Contract
- storedIn: Logs - **has a** Logs object; specifies logs where the contract is stored

Constructors:

- TransactionContract(Payment, Booking, Logs) - constructor that initializes the contract with payment method, booking details, and logs where the contract is stored

Methods:

- displayContract(): void - displays the entire contract to review before finalization or for reference

Logs Class:

Overview:

The Logs class keeps a sequential history of all transaction contracts. It allows employees to view past logs and add new transaction records to the log system.

Attributes:

- stored: TransactionContract[] - list that stores all the contracts **using** TransactionContract objects

Methods:

- viewLogs(): void - displays all stored logs
- add(TransactionContract): void - **uses** TransactionContract; adds a new TransactionContract to the logs

6.4 Development Timeline

Weeks 1-3: First three weeks will focus on the development of the core parts of the system

Khoi will focus on the back-end infrastructure of the system, specifically handling the authentication and gateway integration of the system. He will develop code for user authentication, load balancing logic, and ensuring smooth traffic distribution to the Web App Servers. He will also work on integrating secure API calls for both the Payment API and GPS API.

Arthur will focus on front-end and service integration. He will develop the foundation for the Web App Servers, and the Browse Service which connects users to the inventory data. Arthur will also work on the Booking Service, ensuring it communicates effectively with the Browse Service and external APIs. He will ensure the front-end design integrates seamlessly with the back-end code.

Isabella will handle the database and services interaction. She will develop code for the Vehicle Inventory management, ensuring that the Read and Write Servers interact properly with

the Inventory and Transactional History databases. She will also implement the Update Service, ensuring all car inventory changes, such as adding, deleting, and updating, reflect in the system in real-time.

Weeks 4-5: Two weeks of collaboration to ensure cohesion across all parts of the system

After establishing the system's foundation, the team will work together to ensure all components connect and function smoothly. Khoi will continue to integrate external APIs, while Arthur and Isabella will collaborate on ensuring the booking and payment flows operate cohesively, connecting the front-end services with the database.

Weeks 6-8: Last two weeks for testing and debugging the system

The last weeks are for testing and debugging. The team will work together on debugging, optimizing code, and improving performance based on initial test feedback.

Test Plan (7)

7.1 Design Retention

SWA Justification:

Upon placing the existing Software Architecture Diagram under review, it was determined that no changes were necessary to the general blueprint of the system and its component design. The provided architecture is not overcomplicated, saving time and resources during implementation, however, it also lists all of the necessary structures so that the system can

work as intended. In other words, all functionality can be implemented using the given structures, while no resources are wasted by unnecessary components.

UML Justification:

The provided UML diagram thoroughly covers all specified requirements from the client, ensuring that all functionalities can be achieved using the code design already established. The design is straightforward, without excessive complexity, which reduces the risk of implementation errors. Most importantly, the current design is granular enough for testing, as each of the required functionalities are listed in the form of clear, tangible methods that relate between the provided classes. This way, during testing, each of said functionalities can be targeted for confirmation. since they all hold their functions by name, and naturally relate to the other classes. This strategy paves the way for system testing in the form of use cases, integrated testing between classes, and isolated unit tests of the methods themselves.

7.2 Testing

- These are referred to as Unit Tests since they target specific methods within a single class

Unit Test I: **Toggling Car Availability using updateAvailabilityStatus()**

```
Car testCar
testCar.availability = false
testCar.updateAvailabilityStatus(true)
// test
if (testCar.availability == true) return PASS
else return FAIL
```

Explanation:

This unit test verifies that the updateAvailabilityStatus method in the Car class correctly updates the availability status of a Car object. The test begins by creating an instance of type Car called testCar and setting its initial availability attribute to false. Then, the updateAvailabilityStatus method is called with the argument true to indicate that the car is now available for booking. The test checks whether the availability attribute of testCar is now true. If the availability status is updated correctly, the test passes, confirming that the updateAvailabilityStatus method functions as intended by changing the car's availability to the new availability status for booking. If the method doesn't work properly, the test is guaranteed to fail since availability was deliberately set to false at the beginning.

Unit Test II: **Customer updates ID using newID()**

Customer testCustomer
testCustomer.license_ID = "5432"
newCustomerID = "1234"
testCustomer.newID(newCustomerID)
//test
if (testCustomer.license_ID.equals(newCustomerID)) return PASS
else return FAIL

Explanation:

This unit test verifies that the newID method in the Customer class correctly updates the license_ID attribute. It starts by initializing a Customer object and setting an existing ID directly. Then, the test defines a new ID value to be used for the update. By calling newID with this new value, we check that the license_ID updates as expected. The test passes if the license_ID matches the new value, confirming that newID works as expected. If newID doesn't work properly, the test is guaranteed to fail, since the license_ID was manually set to something else prior to attempting the method.

Unit Test III: **Update payment information (Expiry Date)**

Payment momsCard
momsCard.expiryDate = "07.24"
momsCard.updateInfo()
INPUT: "-", "07.27", "-", "-", "-", 0, "-"
// test
if (momsCard.expiryDate.equals("07.27") return PASS
else return FAIL

Explanation:

This unit test verifies that the updateInfo method in the Payment class accesses and updates the fields of the Payment object correctly. The test manually sets the expiration date of a new Payment object to 07/24. The updateInfo method is called, with input of placeholders in all fields except the expiryDate of 07/27, which would hopefully update that field to "07/27". Finally the test checks if the expiryDate was updated and passes if so, or fails otherwise. This test confirms that the updateInfo method accesses and updates payment fields correctly, and is guaranteed to fail otherwise.

Functional tests:

- These tests are more integrated, using methods that exercise functions between multiple interrelated classes.

Integration Test I: **Employee Updating Booking Field**

Employee tester
Booking testBooking
testBooking.endDate = "10/20/24"
testBooking = tester.updateBooking()
INPUT: "-", "-", "-","11/02/24"
// test
if (testBooking.endDate.equals("11/02/24")) return PASS
else return FAIL

Explanation:

This integration test checks whether the Employee class can update existing Booking objects using the updateBooking() method. The test starts by initializing Booking and Employee objects, testBooking and tester, and the initial end date set to "10/20/24". The testBooking object is set to the updateBooking() method that was called from the tester employee, with input of placeholders in all fields except the endDate of "11/02/24". After updateBooking() is executed, the endDate field in testBooking should update to "11/02/24". Finally, the test checks if testBooking.endDate matches "11/02/24" and passes if it does and fails if not. This test confirms that the updateInfo method works correctly, and is guaranteed to fail otherwise.

Integration Test II: **Employee Adds Car and Checks Inventory Accordingly**

Employee testEmployee
Car testCar
testCar.updateInfo()
INPUT: "red", "Convertible", "Ferrari", "488 Spider", "2020", "4S4BRBPC5E3227809", "1200"

Inventory inven
int oldVehicleCount = inven.vehicleList.size()
testEmployee.addVehicle(testCar, inven)
boolean carInInventory = inven.vehicleList.contains(testCar)
boolean sizeIncreasedTest = (inven.vehicleList.size() == oldVehicleCount + 1)
if (carInInventory && sizeIncreasedTest) return PASS
else return FAIL

Explanation:

This integration test verifies that the addVehicle method in the Employee class correctly adds a Car object to an Inventory. The test begins by creating an instance of Employee and Car, test Employee and testCar, and then updating the car's information using the updateInfo method with specified attributes as input, such as color, type, make, model, year, VIN, and mileage. Next, the test initializes an Inventory object and records the current number of vehicles in inven with oldVehicleCount. After calling testEmplyoee's addVehicle(testCar, inven), the test checks two conditions: first, whether testCar has been successfully added to the inventory with a "contains" check, and second, whether the total number of vehicles in the inventory has increased by one. The test passes if both conditions are true, confirming that addVehicle works as intended by correctly updating the inventory and ensuring that the new vehicle is included. Otherwise, it fails.

System Tests:

- These tests are meant to encapsulate the system as a whole. They are each designed to represent a series of events which together comprise a realistic use case of the user utilizing the system in practice.

System Test I:

Customer goes on the website as a guest, and chooses to log in with their personal login information (Guest.loginCustomer()). The customer chooses to update their payment information (Customer.changePayment(new Payment)) . They choose to book a vehicle (Customer.bookVehicle(Car, Inventory) -> Customer.makePayment() -> new Booking(...)). A

new contract representing their rental will include booking details about payment, dates, identity, etc. (new TransactionContract(Payment, Booking, Logs)). The company logs will be updated with the new contract (Logs.add(TransactionContract)).

This works as a system test, since the flow is practically covering an entire use cas from the perspective of a customer. A user utilizes the Guest, Customer, Payment, Car, Inventory, Booking, TransactionContract, and Logs classes and their functions, all in one go.

System Test II:

Employee goes on the website as a guest, and chooses to log in with their personal login information (Guest.loginEmployee()). The employee checks a customer's history (Employee.checkHistory(Customer) -> Customer.viewHistory()), and then updates information about a booking belonging to that customer (Employee.updateBooking()). The employee wants to verify the update they just completed after the fact, so they access the logs holding the contracts to view the updated contract, which is derived from the booking, for themselves (Employee.accessLogs(Logs) -> Logs.viewLogs() -> TransactionContract.displayContract()).

This works as a system test, since the flow of the test interrelates most of the program's classes. Their related functionalities are encapsulated in one holistic use case from the perspective of an employee logging in to change a booking for a customer, and then verifying the successful update by checking the database.