BeAvis Car Rental Software Design Specification

Authors: Spencer Schmitt, Alex Zucker, Sean Tran

Delivered 11/8/2024

Section 1: Introduction

1.1: Document contents

1.2: System overview

The client, BeAvis, has ordered the creation of the software system that will be used to replace the pen-to-paper car rental process, making it electronic-based and useful for managing the inventory of rental cars. The client sees physical documents and records as impractical and wants to keep up with an evolving electronic world. The client aims to digitize their day-to-day operation. We expect that overhauling the process will increase customer satisfaction and employees' workload efficiency.

The client's users, customers, and employees will interact extensively with the system. The users, managers, and administrators will have access to the backend of the software system for their respective tasks.

Successful design and implementation of the software system will improve the ease of customer experience, making it comfortable for them. The system will

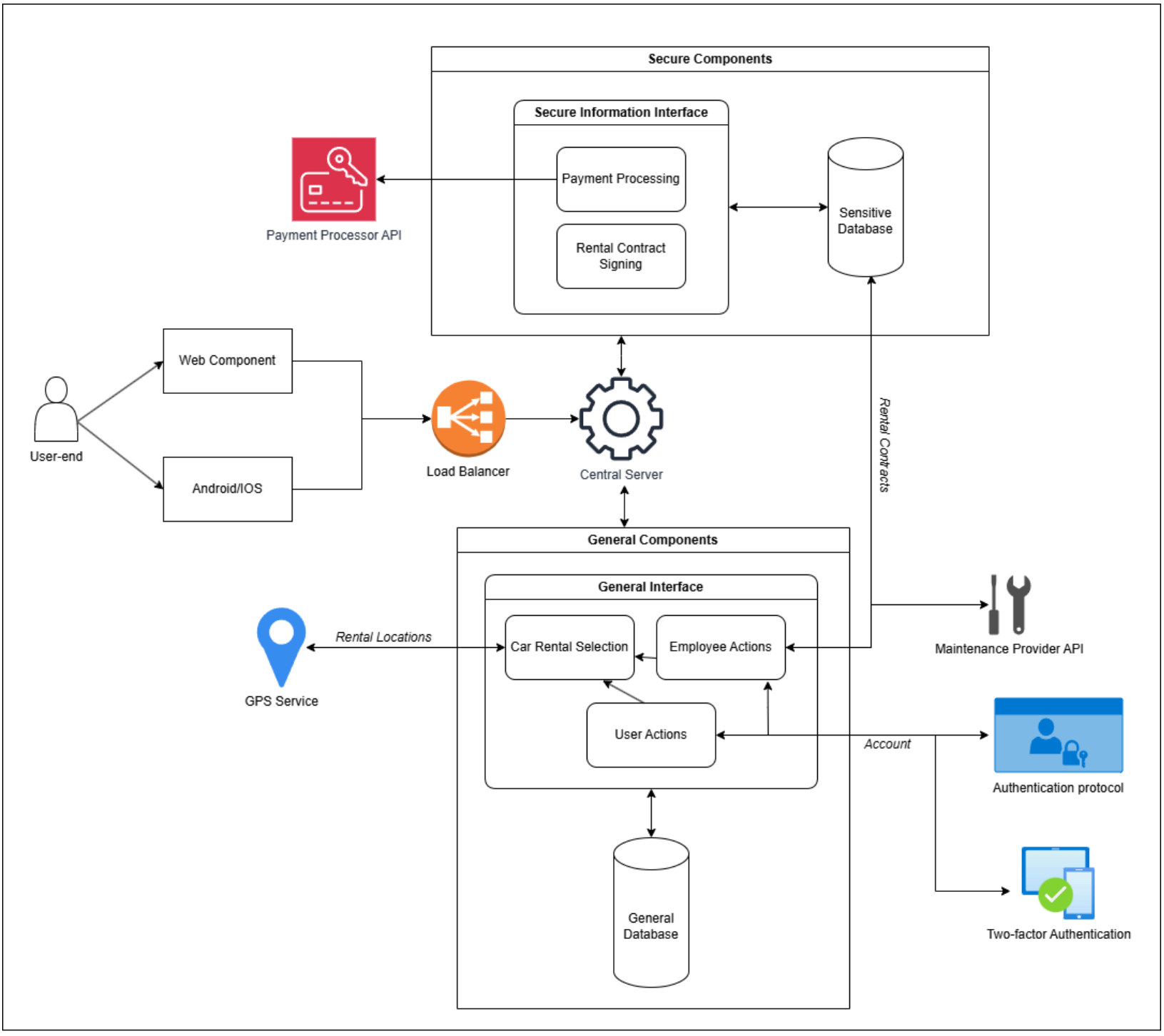
also increase employee productivity and decrease the time for workload by several hours per week.

Section 2: Software Architecture Overview

2.1: Architecture Diagram of all major components

This software architecture diagram (see next page) covers all the major components necessary for the BeAvis Car Rental software.

BeAvis Software Architecture Diagram



Following this diagram is a description of all the components and connections within it, starting from the very left.

User End

- This component represents the point of interaction between the user and BeAvis Car Rental Software. The user-end has two arrows to “Web Component” and “Android/IOS”, representing the different interfaces a user can interact with.
    - Web Component
        - The user should be able to interact with the software via a web-app interface.
    - Android/IOS
        - The user should be able to interact with the software via a mobile interface, accounting for both IOS and Android.

Load Balancer

- Arrows pointing from the Web Component and Android/IOS to the load balancer represent how traffic from both interfaces will need to be throttled and distributed between BeAvis servers (which will be done by the Load Balancer).

Central Server

- Traffic and inputs passed through the load balancer will then be directed by a central server based on if the input requires access to secure or general components. It also handles communication and processes between the two major types of components.

General Components

- General Interface
    - The general interface is an organizational distinction that represents user actions (Including both customers and employees) that interact with the main database or general 3rd party APIs including:
        - GPS Service
            - Necessary for finding nearby BeAvis Car Rental Locations using GPS.
        - Authentication Protocol
            - Necessary for authenticating login attempts.
        - Two-Factor Authentication

            - Necessary for allowing the user to set up two-factor authentication for their account to better secure it.
        - Maintenance Provider API
            - This API allows an employee to efficiently submit maintenance requests through the BeAvis software system.
- Main Database
    - This database contains all data deemed “low-priority” for security purposes. This includes data like user car rental history and car status.

<u>Secure Components</u>

- Secure Interface
    - The secure interface is an organization distinction that represents user actions related to sensitive information like payment processing and rental contract signing.
    - Payment Processor API
        - The secure interface interacts with the payment processor API to handle payments.
- Payment Info/Rental Contract Database
    - This database is designated as a high-priority database that holds sensitive information like payment information and rental contracts. It may also hold other sensitive information like login credentials.

<u>Miscellaneous Connections</u>

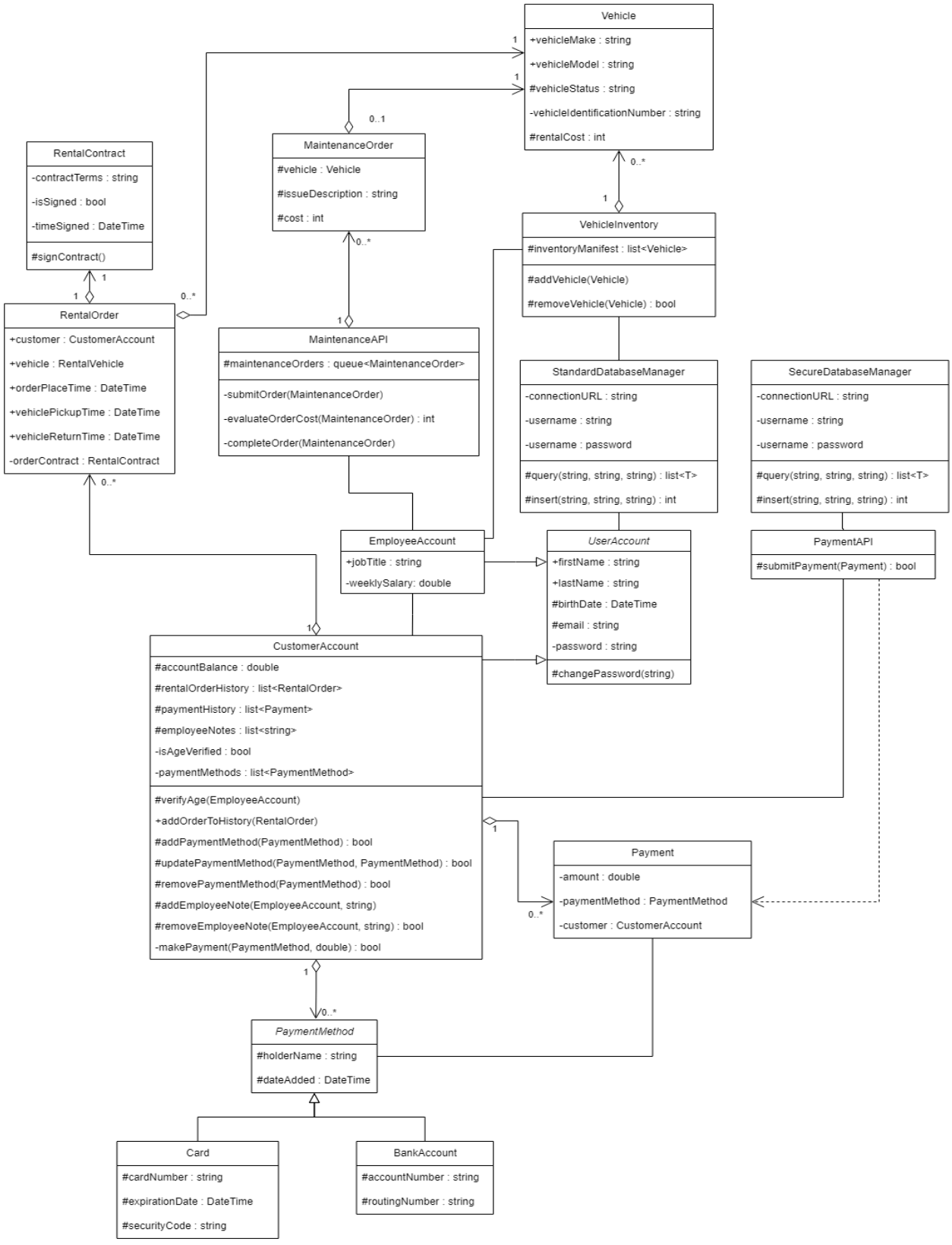
- The majority of connections with 3rd party systems represent necessary implementations that the BeAvis system must interact with to properly function.

2.2: UML Class Diagram

This UML class diagram (see next page) illustrates the classes representing the backend of the BeAvis Car Rental software, as well as their member variables, functions, and relationships to one another. **Note that in order to prevent bloat, basic accessor and mutator methods that possess no additional functionality are not explicitly reflected in the diagram**. Instead, existence of these methods is implied via the use of access modifiers:

- Public member variables have public accessor and mutator methods
- Protected member variables have protected accessor and mutator methods
- Private member variables lack accessor and mutator methods

BeAvis UML Class Diagram



This section will provide a textual summary of the functionality of each depicted class, as well as its relationships to other classes.

Nonvolatile storage classes

- StandardDatabaseManager
    - Acts as an interface between the software system's code and its permanent data storage solution. This class connects to the database used to store all information not related to customer payments. Runtime data-heavy classes, such as those that store lists of objects, query this class to obtain and update their respective member variables.
- SecureDatabaseManager
    - Functions nearly identically to StandardDatabaseManager, but connects to the secure database that is solely responsible for storing and retrieving information relating to customer payments and payment methods. As a result, this class is used by PaymentAPI for read/write requests.

API classes

- PaymentAPI
    - Acts as the interface between the software system's code and the third party service used to process customer payments. This class is dependent on the Payment class to function, and simply reads data from parameterized Payment objects in order to submit payment requests for processing.
- MaintenanceAPI
    - Acts as the interface between the software system's code and the third party service used to repair rental vehicles. Currently outstanding orders are represented via a queue member variable of type MaintenanceOrder, which can be added to by EmployeeAccount instances and removed from by the maintenance service.

User classes

- UserAccount
    - An abstract class which serves as the superclass for all instantiable account class objects. Contains member variables and functions that are required for both customers and employees alike. UserAccount

instances derive their initialized data values by querying StandardDatabaseManager.

- EmployeeAccount
    - A subclass of UserAccount which represents employee users of the software system. Compared to their customer counterparts, EmployeeAccount instances have elevated data read/write permissions within the software system. These elevated permissions are illustrated by association relationships with CustomerAccount, MaintenanceAPI, and VehicleInventory.
- CustomerAccount
    - A subclass of UserAccount which represents customer users of the software system. This class contains a large number of member variables and functions which are used to store, access, and modify data pertaining to individual customers. These member variables include lists of RentalOrder, PaymentMethod, and Payment objects, all of which are illustrated via aggregate relationships to the respective classes.

<u>Payment information classes</u>

- Payment
    - A simple class which serves as the representation of a single payment made by a customer user of the software system. Instances of this class are used to represent past payments that have been made, as well as payments that are currently being processed by the third party payment processor service.
- PaymentMethod
    - An abstract class which serves as the superclass for all instantiable payment method objects. Contains member variables which are relevant to all payment methods.
- Card
    - A subclass of PaymentMethod which represents a debit or credit card that has been registered to a CustomerAccount instance.
- BankAccount
    - A subclass of PaymentMethod which represents a bank account that has been registered to a CustomerAccount instance.

<u>Rental order classes</u>

- RentalOrder

    - A class representing an individual order for a rental vehicle placed by a CustomerAccount instance. Among other member variables, contains a RentalContract and Vehicle, which are visualized by aggregate relationships with 1:1 multiplicities.
- RentalContract
    - Represents the legally binding contract between vehicle renter and rentee. A RentalContract instance is always related to 1 and only 1 RentalOrder instance.

<u>Vehicle inventory classes</u>

- VehicleInventory
    - Serves as the digital inventory manifest of the software system. This inventory is directly represented by a member variable list of type Vehicle, which is initialized via queries to StandardDatabaseManager.
- Vehicle
    - A class representing an individual rental vehicle registered to the VehicleInventory digital manifest. Corresponds to any number of RentalOrder instances, only one of which can be a currently active order. May also correspond to 1 currently outstanding MaintenanceOrder.
- MaintenanceOrder
    - Represents a single maintenance request that has been sent to the third party maintenance service. Contains member variables representing the vehicle to be serviced, as well as a single member function allowing the maintenance provider to specify the quoted cost for fulfilling the service request.

Section 3: Data management strategy

The software system utilizes two separate SQL databases as its nonvolatile data storage solution. As depicted in the software architecture diagram in subsection 2.1, the majority of the system's data is stored in the primary database, while any payment details or contract terms are stored in a separate sensitive database for enhanced security.

The tables contained within each database closely correspond to classes depicted by the UML diagram in subsection 2.2. This relational database structure allows for an extremely simple conversion between nonvolatile and volatile data. At runtime, the software system queries a table, as the result of that query is turned into an object that can then be accessed by memory without having to be queried again.

The downside of this relational database structure is the heavy reliance on foreign keys; as tables are closely related to one another, a commonly referenced table may have its primary key stored as foreign keys by numerous other tables. These duplicate data values result in additional storage overhead for the software system, but this has been deemed an acceptable cost to pay for the intuitive and streamlined nature of the databases. While utilization of a NoSQL data storage solution could yield lower spatial requirements, a nonrelational database structure would ultimately overcomplicate the data management needs of the software system.

This section contains spreadsheet-like diagrams of the structure of all SQL database tables utilized by the software system. Each diagram is accompanied by a brief description of the table's purpose, as well any fields which serve as primary or foreign keys. As the software system's nonvolatile data storage is split between two separate databases, this section is partitioned into subsections, each corresponding to a different database.

3.1: Primary database tables

primary_db.customers

customer_id : INT	first_name : VARCHAR(25)	last_name : VARCHAR(25)	birth_date : DATE	is_age_verified : BOOLEAN	email_address : VARCHAR(40)	password_hash : CHAR(60)	account_balance : DECIMAL(9,2)
0	Ella	Ferrell	2024-11-11	TRUE	Sidney20@gmail.com	$2y$10$.vGA1O9wmRjrwAVXD98HNOgsNpDczlqm3Jq7KnEd1VAGv3Fykk1a2	280.65

The customers table acts as a record of all currently existing user accounts created by customers. Each record has a unique numerical identifier, customer_id, which serves as the table's primary key. As the customer account acts as the central unit of information for the software system, the customer_id field is a commonly referenced foreign key for several other tables.

primary_db.employee_notes

employee_note_id : INT	customer_id : INT	employee_id : INT	note_message : TEXT(16384)
0	0	0	She prefers the BMW series

The employee_notes table stores all employee-only notes which have been added to customer accounts. Each note has a unique numerical identifier, employee_node, which serves as the table's primary key. As every note must belong to a single customer account and is created by a single employee, both the customer_id and employee_id fields serve as foreign keys.

primary_db.employees

employee_id : INT	first_name : VARCHAR(25)	last_name : VARCHAR(25)	birth_date : DATE	email_address : VARCHAR(40)	job_title : TINYTEXT	weekly_salary : DECIMAL(8, 2)	password_hash : CHAR(60)
0	John	Smith	1986-11-23	johnsmith345@gmail.com	Rental Agent	2307.87	$2a$04$hUkS8olMqh21M9hY9vAtK.QkYVv5YkEsXU3nxtSqZRZHpuT1WVNmC

The employees table acts as the employee counterpart to the customers table; it stores a record of all currently existing user accounts created by employees. Each employee has a unique numerical identifier, employee_id, which serves as the table's primary key. While this field is not as heavily referenced as its customers counterpart, it is still utilized by the employee_notes table as a foreign key.

primary_db.vehicles

vin : CHAR(17)	vehicle_make : VARCHAR(20)	vehicle_model : VARCHAR(20)	vehicle_status : TEXT(8192)	rental_cost : DECIMAL(6,2)	rental_branch : VARCHAR(40)
WBAAD1302LED13147	BMW	1990 BMW 3 Series	Rented	235.99	BeAvis San Diego

The vehicles table acts as the persistent vehicle manifest for the software system. The table contains a record for every vehicle currently registered by the system, with each vehicle's vehicle identification number (VIN) serving as the table's primary key. This key is referenced by both the rental_orders and maintenance_orders table as a foreign key.

primary_db.rental_orders

rental_order_id : INT	customer_id : INT	vin : CHAR(17)	contract_id : INT	order_time_placed : DATETIME	vehicle_pickup_time : DATETIME	vehicle_return_time : DATETIME
0	0	WBAAD1302LED13147	0	2024-10-14 15:40:32	10/30/2024 7:00:00	11/03/2024 20:00:00

The rental_orders table stores all vehicle rental orders that have been played by customers. Each rental order has a unique numerical identifier, rental_order_id, which serves as the table's primary key. As a rental order references several other clusters of information, this table makes heavy use of the customer_id, vin, and contract_id foreign keys in order to relate to the respective components of the software system.

primary_db.maintenance_orders

maintenance_order_id: INT	vin : CHAR(17)	cost : DECIMAL(8,2)	issue_description : TEXT(8192)	is_complete : BOOLEAN
0	WBAAD1302LED13147	100.89	Needs headlight fluid	TRUE

The maintenance_orders table serves as a record of every vehicle maintenance order that has been placed via the software system. Each order has a unique numerical identifier, maintenance_order_id, which serves as the table's primary key. As every maintenance order is for a specific vehicle, the table also utilizes the vin field as a foreign key to reference the vehicles table.

3.2: Sensitive database tables

sensitive_db.payment_methods

payment_method_id : INT	customer_id : INT	serialized_json : TEXT(2048)	object_type : VARCHAR(10)	date_added : DATETIME
0	0	$2y$10$.vGA1O9wmRjrwAVXD98H	Credit	2024-10-14 15:33:05

The payment_methods table stores all payment methods registered to customer accounts. Each payment method has a unique numerical identifier, payment_method_id, which serves as the table's primary key. As each payment method belongs to a specific customer account, the table utilizes the customer_id foreign key to reference the customers table. Since the payment_methods table must be capable of storing records for both credit/debit cards and bank accounts, the credentials for every payment method are consolidated into a JSON object. This JSON object is then encrypted, and the resulting string is stored in the serialized_json field.

sensitive_db.payments

payment_id : INT	payment_method_id : INT	customer_id : INT	payment_amount: DECIMAL(8,2)
0	0	0	263.68

The payments table stores a record of every payment that has been made by customer accounts via the software system. Each payment has a unique numerical identifier, payment_id, which acts as the table's primary key. The table makes use of the payment_method_id and customer_id foreign keys to relate its records to both the customer who placed the order, and the payment method that was used to do so.

sensitive_db.rental_contracts

contract_id : INT	rental_order_id : INT	time_signed : DATETIME	is_signed : BOOLEAN	serialized_json : MEDIUMTEXT
0	0	2024-10-14 13:00:00	TRUE	$2y$10$.vGA1O9wmRjrwAVXD98H

The rental_contracts table stores a record of every rental contract associated with each vehicle rental order. Each contract has a unique numerical identifier, contract_id, which acts as the table's primary key. As each rental contract belongs to a specific rental order, the table utilizes the rental_order_id foreign key to relate to the rental_orders table. Much like the payment_methods table, the body of each rental contract is converted to a JSON object, which is then serialized and encrypted to be stored in the serialized_json field.

Section 4: System verification test plan

4.1: Rental order test set

Unit test

- Target function: Card.setCardNumber(String)
- Test set-up:

```
DateTime myDateTime = DateTime(2024, 12, 15);
Card sampleCard = new Card();

sampleCard.cardNumber = 6011000990139424;
sampleCard.expirationDate = myDateTime;
samplecard.securityCode = "392";

String newCardNumber = "378282246310005";
```

- Test body:

```
sampleCard.setCardNumber(newCardNumber);

if(sampleCard.getCardNumber().equals(newCardNumber)) {
    return PASS;
}

return FAIL;
```

- Test vectors:
    - Initial card number: "6011000990139424"
    - New card number: "378282246310005"
    - Expected output: new card number assigned to sampleCard object

This unit test represents a lower level test that tests the functionality of the "setCardNumber()" function, which is responsible for changing the card number associated with someone's payment method. Being able to change your card number is a feature that is necessary when paying for a rental car.

This test functions by creating a sample card with sample values. It then updates the card's number with the setCardNumber() function and then compares the card number's value to the argument used to call the setCardNumber() function.

Should the comparison evaluate to true, the card's number has been set correctly and the test passes. Otherwise, the card's number differs from the intended value and the test fails.

Integration test

- Target function: CustomerAccount.addPaymentMethod(PaymentMethod) : bool
- Test set-up:

```
//Assume the CustomerAccount constructor initializes all member fields
CustomerAccount account1 = new CustomerAccount();
DateTime newDateTime = DateTime(2023, 11, 5);

//Use "sampleCard" object creating during the unit test
sampleCard.holderName = "Joe Smith";
sampleCard.dateAdded = newDateTime;
```

- Test body:

```
account1.addPaymentMethod(sampleCard);

for(PaymentMethod method : account1.paymentMethods) {
    if(method.equals(sampleCard) {
        return PASS;
    }
}

return FAIL;
```

- Test vectors:
  - CustomerAccount object: account1
  - PaymentMethod object: sampleCard
  - Expected output: sampleCard exists within the paymentMethods list

This integration test tests the “addPaymentMethod()” function which takes a payment method as an argument and adds said payment method to the list, “paymentMethods” that is associated with a customer's account. A customer needs to add a payment method to their account in order to pay for a rental car, which is the feature being tested. It is an integration test due to its utilization of multiple classes, including CustomerAccount, PaymentMethod, and Card.

The test works by creating a CustomerAccount object and a Card object (which inherits from the abstract class, *PaymentMethod*). It then calls addPaymentMethod() with the card object as a parameter. In order to check that this object has successfully been added as a payment method, the test then loops through all the elements in CustomerAccount object’s paymentMethods list and compares each element to the payment method that was supposed to be added. If it finds it, the test passes. If the loop finishes without finding a match, the test fails.

<u>System test</u>

- Test sequence:
    - Open the mobile interface (the app)
    - Log in to account
    - Add a payment method to your account
    - Update details of payment method
    - Remove that payment method
    - Add another payment method
    - Find a rental location
    - Select a rental car
    - Confirm rental order
    - Make a payment
    - Log out of account

The system test goes through a user's full interaction with the BeAvis Car Rental system to test that they’re able to pay for a rental car. In order to do this, this test includes steps like adding, removing, and updating payment methods.

If the user is able to complete this process by successfully paying for their rental car order, the test will pass. If the order is unsuccessful due to any reason other than invalid data inputted by the user, the test fails.

4.2: Maintenance order test set

Unit test

- Target function: Vehicle.updateVehiclStatus(String)
- Test set-up:

```
Vehicle vehicle = new Vehicle();

vehicle.vehicleMake = "Infiniti";
vehicle.vehicleModel = "G37";
vehicle.vehicleStatus = "Operational";
vehicle.vehicleIdentificationNumber = "JH4DB1550NS000306"
vehicle.rentalCost = 750;

String newVehicleStatus = "Needs repairs";
```

- Test body:

```
vehicle.updateVehicleStatus(newVehicleStatus);

if (vehicle.getVehicleStatus().equals(newVehicleStatus))
{
    return PASS;
}else{
    return FAIL;
}
```

- Test vectors:
    - Initial vehicle status: "Operational"
    - New vehicle status: "Needs repairs"
    - Expected output: new vehicle status assigned to vehicle object

The test's purpose is to test the updateVehicleStatus method from the Vehicle class. The features of the design we are using are the vehicle's information and the updateVehicleStatus method. The test vectors are the vehicle's status and a new status for the vehicle. The selected test covers the targeted features because

we are updating the vehicle's status from "operational" to "need repairs" and we check to see if the status is changed as intended.

Integration test

- Target function: MaintenanceApi.evaluateOrderCost(MaintenanceOrder) : int
- Test set-up:

```
MaintenanceOrder currentOrder = maintenanceOrders.peek();
```

- Test body:

```
int quotedCost = evaluateOrderCost(currentOrder);


currentOrder.setCost(quotedCost);


if (quotedCost == currentOrder.getCost()){
    return PASS;
}else{
    return FAIL;
}
```

- Test vectors:
    - MaintenanceOrder object: currentOrder
    - Quoted order cost: quotedCost
    - Expected output: quotedCost value assigned to currentOrder's cost member variable

The test's purpose is to test the evaluateOrderCost(MaintenanceOrder) from the MaintenanceAPI. The targeted feature will be the evaluatedOrderCost method and it will return the cost of the order. The test vectors will be a Maintenance Order object provided by the order queue and an integer set to the evaluated order cost of the current order. The selected test covers the targeted feature because we are using it to get the quoted cost and make sure that the current order's cost is set to the quoted cost.

System test

- Test sequence:

    - Employee logs into account
    - Employee selects a vehicle from the VehicleInventory manifest
    - Employee submits a maintenance order for the vehicle, providing a description of the issue with the vehicle
    - The order's cost will be evaluated and set by the maintenance provider
    - The maintenance provider adds the order to the order queue
    - At the time of completion, the maintenance order will be marked as complete by the maintenance provider
    - The maintenance order is deleted from the order queue

The test involves the Vehicle, VehicleInventory, MaintenanceOrder, and MaintenanceAPI classes, utilizing many of their respective member variables and functions. The tested features of the designs are selecting a vehicle from the inventory manifest, creating a maintenance order, evaluating the cost of the maintenance order, submitting a maintenance order into the queue, and completing said maintenance order.

If a MaintenanceOrder is able to be created, submitted, updated, and completed successfully, then the test passes. Should any of these steps fail to advance the process to the next stage, the test will fail.

Section 5: Development plan and timeline

5.1: Partitioning of tasks

Development plan

For the current phase, the software requirements should have already been finished and given to the development and implementation team to start the process of designing the software system via the software architecture diagram and the use case diagram. With the completed diagrams, the team should be able to start implementing the software requirements in accordance.

Projected timeline

- Week 1: Dedicated to gathering user and system requirements
- Week 2: Dedicated to designing the software architecture diagram and the UML diagram following the user and system requirements of the software system.

- Weeks 3-6: Dedicated to the implementation of the software system following the user requirements, system requirements, functional and non-functional requirements, the SWA diagram, and the UML diagram
- Weeks 7-11: The most important phase where we have to perform quality checks on the system, making sure that the system has performed the necessary actions that the client wants without fail and bugs.
- Weeks 11-14: Maintaining the software system.

5.2: Team member responsibilities

Alex Zucker

- Team roles:
    - Documentation & formatting
    - Software design
    - Database design
    - Implementation
- Responsibilities
    - Responsible for the UML Diagram of the software system.
    - Collaborated with Spencer Schmitt and Sean Tran to implement the software system following the software requirements.
    - Collaborated with Spencer Schmitt and Sean Tran to design and format the current software design specification document.

Spencer Schmitt

- Team roles:
    - Software design
    - Database design
    - Test suite creation
    - Implementation
- Responsibilities:
    - Responsible for the Software Architecture Diagram of the software system
    - Collaborated with Alex Zucker and Sean Tran to implement the software system following the software requirements.
    - Collaborated with Alex Zucker and Sean Tran to design and format the current software design specification document.

Sean Tran

- Team roles:
    - Software design
    - Database design
    - Test suite creation
    - Implementation
- Responsibilities:
    - Collaborated with Spencer Schmitt and Alex Zucker to implement the software system following the software requirements.
    - Collaborated with Spencer Schmitt and Alex Zucker to design and format the current software design specification document.

Section 6: Document updates and changelog

6.1: Verification and validation update

Before beginning the creation of system verification test suites, the design team conducted a review of the software architecture and UML class diagrams. The purpose of this review was to assess whether the aforementioned diagrams were complete and accurate, and by extension, determining what changes would need to be made in the event that the diagrams were found lacking.

Our team concluded that the primary structure of both the software architecture and UML class diagrams sufficiently reflected the desired functionality and organization of the software system. However, several minor changes were made to the UML class diagram in order to facilitate the validation testing process. These changes primarily dealt with renaming member variables and functions, altering access modifiers, and updating method parameters. Each of these alterations will be enumerated in the document changelog, located in the following section.

Additionally, many formatting changes have been made to this document in order to improve readability and ease of use.

6.2: Document changelog

- Verification and validation update
    - UML class diagram updates
        - Added diagram foreword specifying that basic accessor and mutator methods are omitted from the UML diagram to prevent bloat
            - Pruned remaining accessor and mutator methods from the UML diagram
        - Added PaymentMethod parameter to CustomerAccount.removePaymentMethod()
        - Updated MaintenanceApi.evaluateOrderCost(MaintenanceOrder) function return type to int
            - This function's purpose is to now estimate the cost of completing a MaintenanceOrder object based on parts and labor. The quoted cost is the value returned by the function.
        - Removed MaintenanceOrder.evaluateCost(int) function
            - The name of this function was misleading, and has been replaced by a simple mutator. As mutator methods are generally omitted from the UML diagram, this function is only visually represented by the access modifier on the cost member variable.
    - Formatting changes
        - Added table of contents
        - Streamlined text font, size, other stylization aspects