UE22CS352B - Object Oriented Analysis & Design

Mini Project Report

Vehicle Rental Management System

Submitted by:

Name : SRN (Team Members)

Amisha Jain
PES1UG22CS074
Amritaa Kalanee
PES1UG22CS077
Archisha Janawade
PES1UG22CS105
Arushi Katta
PES1UG22CS109

6th semester, B section

Asst Prof Priya Badrinath

January - May 2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013) 100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement:

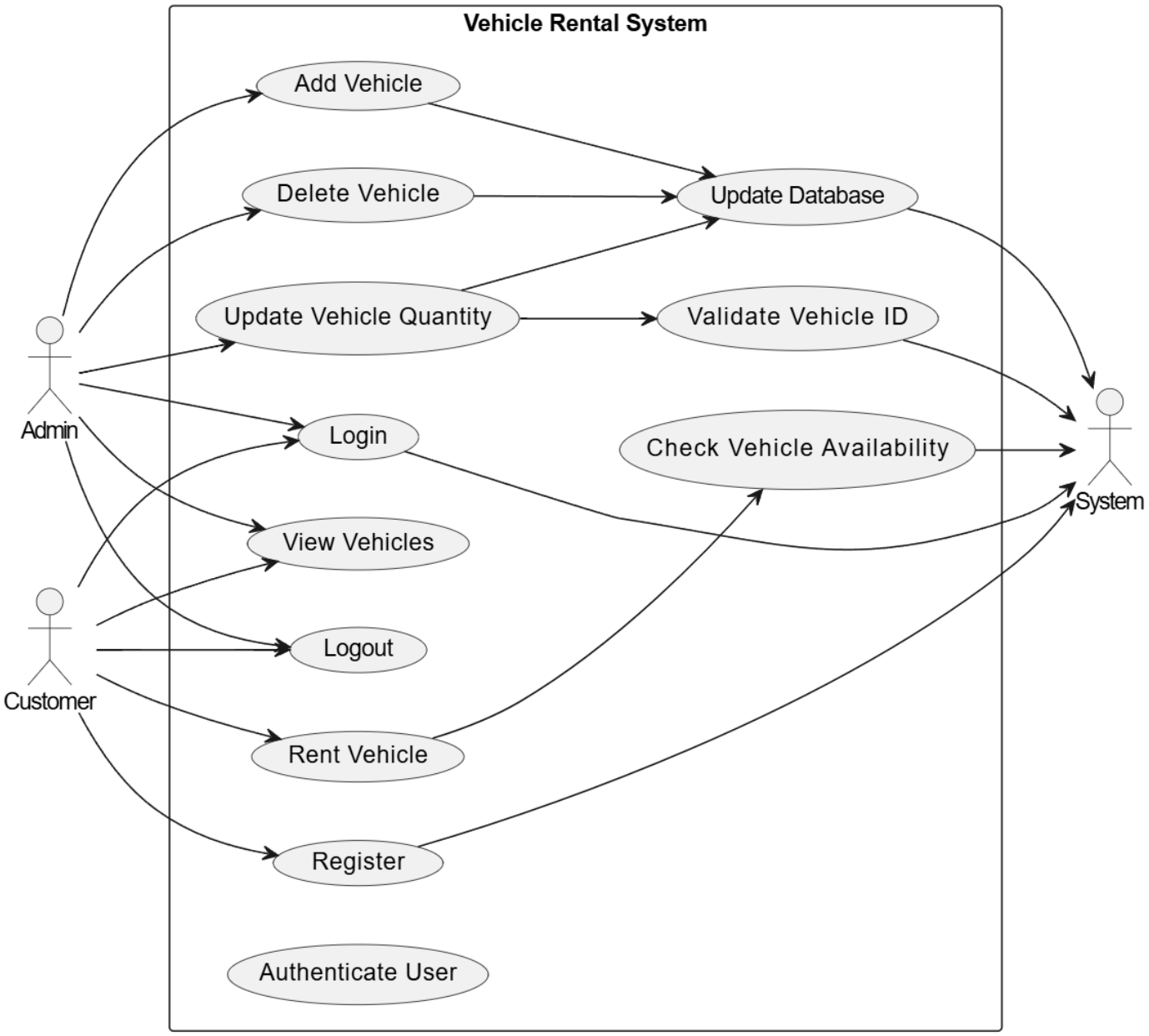
To develop a comprehensive Vehicle Rental Management System that efficiently manages the rental process for various types of vehicles, including cars and two-wheelers. The system should allow administrators to manage vehicle inventory and enable customers to browse and rent vehicles for specific time periods.

Key Features:

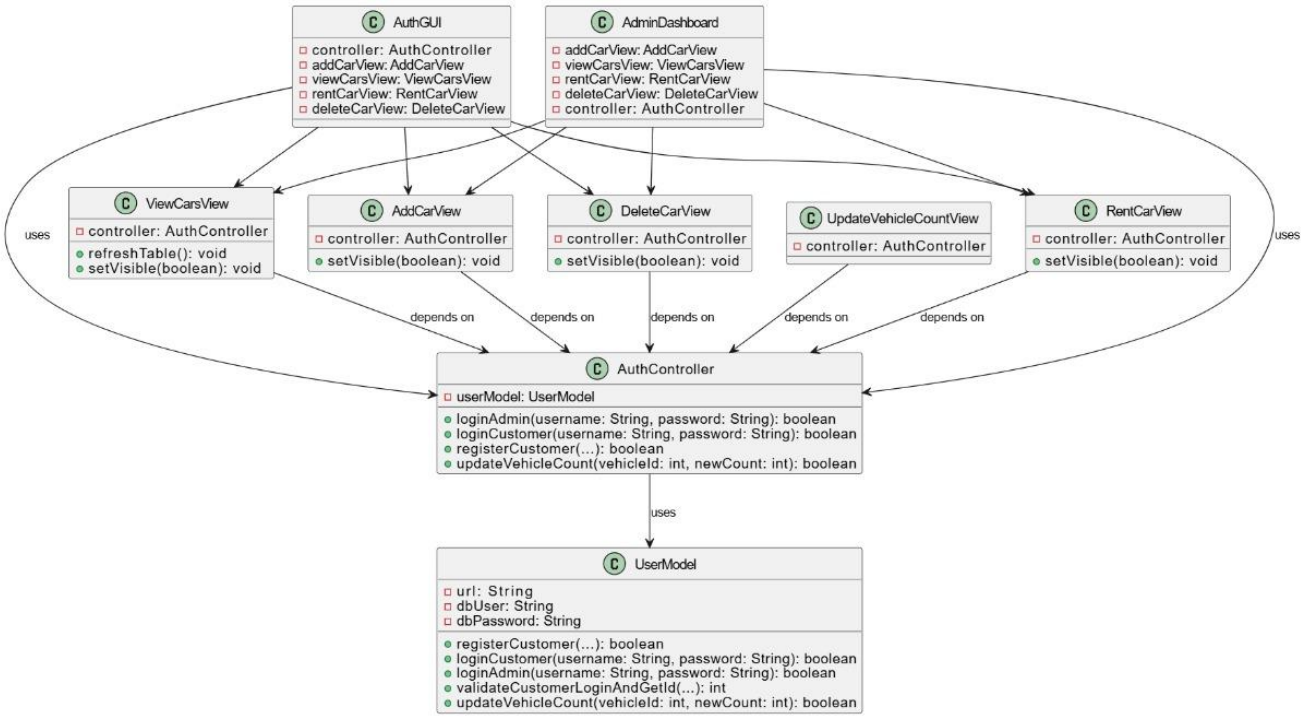
1. User authentication with separate admin and customer roles
2. Vehicle inventory management (add, view, update count, delete)
3. Vehicle rental processing with date selection
4. Automatic payment calculation based on rental duration
5. User-friendly GUI interface built with Java Swing
6. Database integration for persistent data storage

Models:

Use Case Diagram:

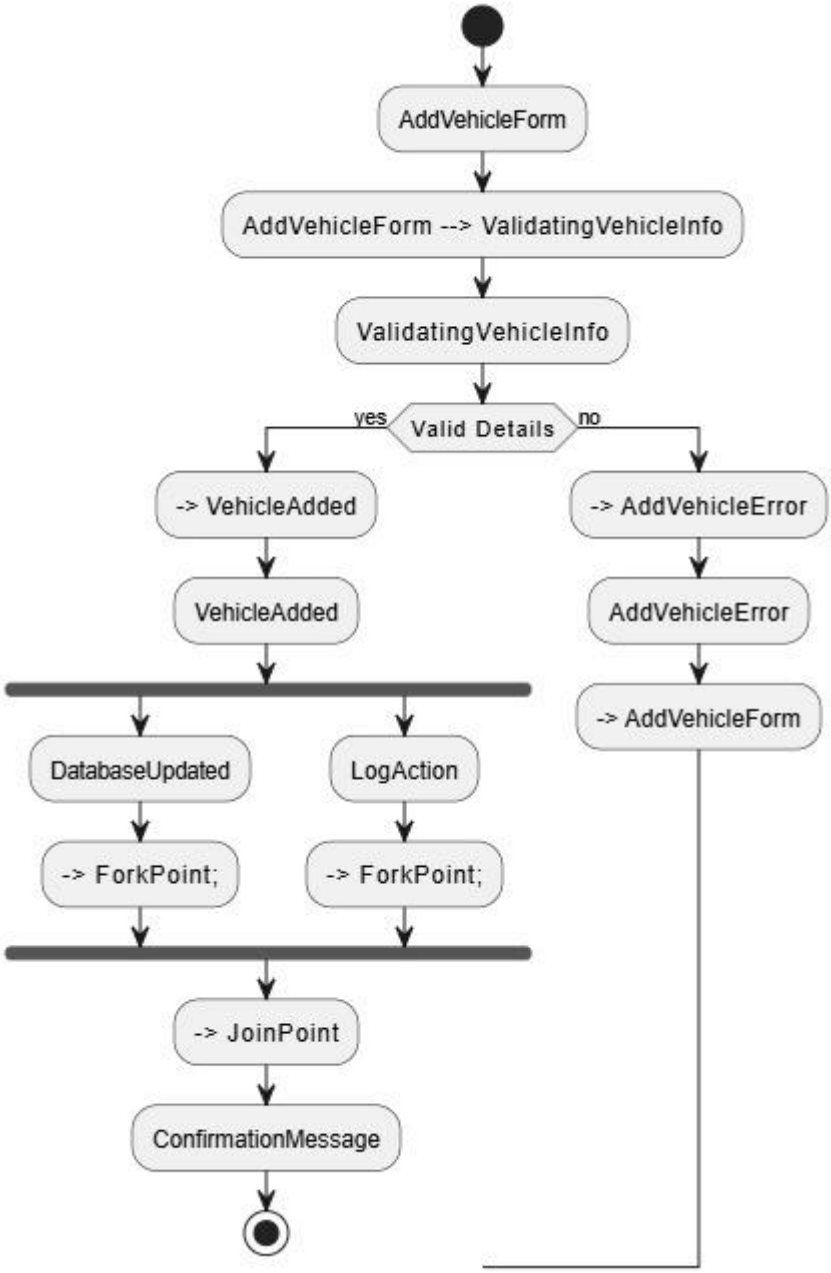


Class Diagram:

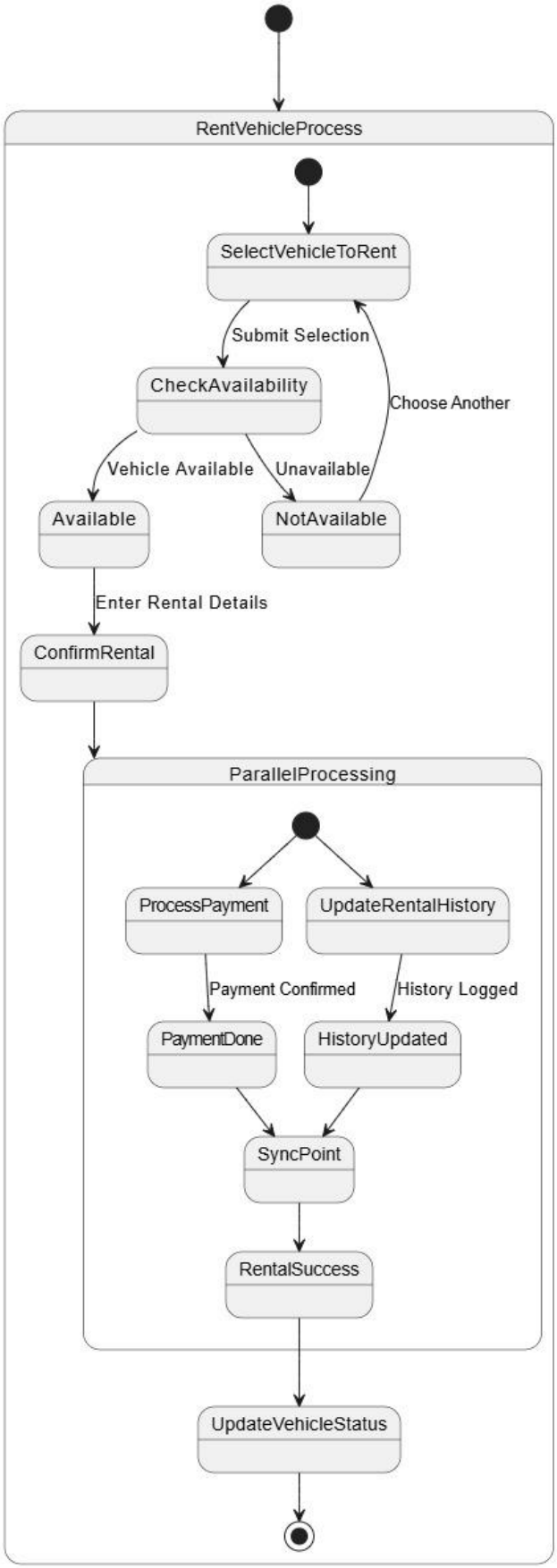


State Diagram:

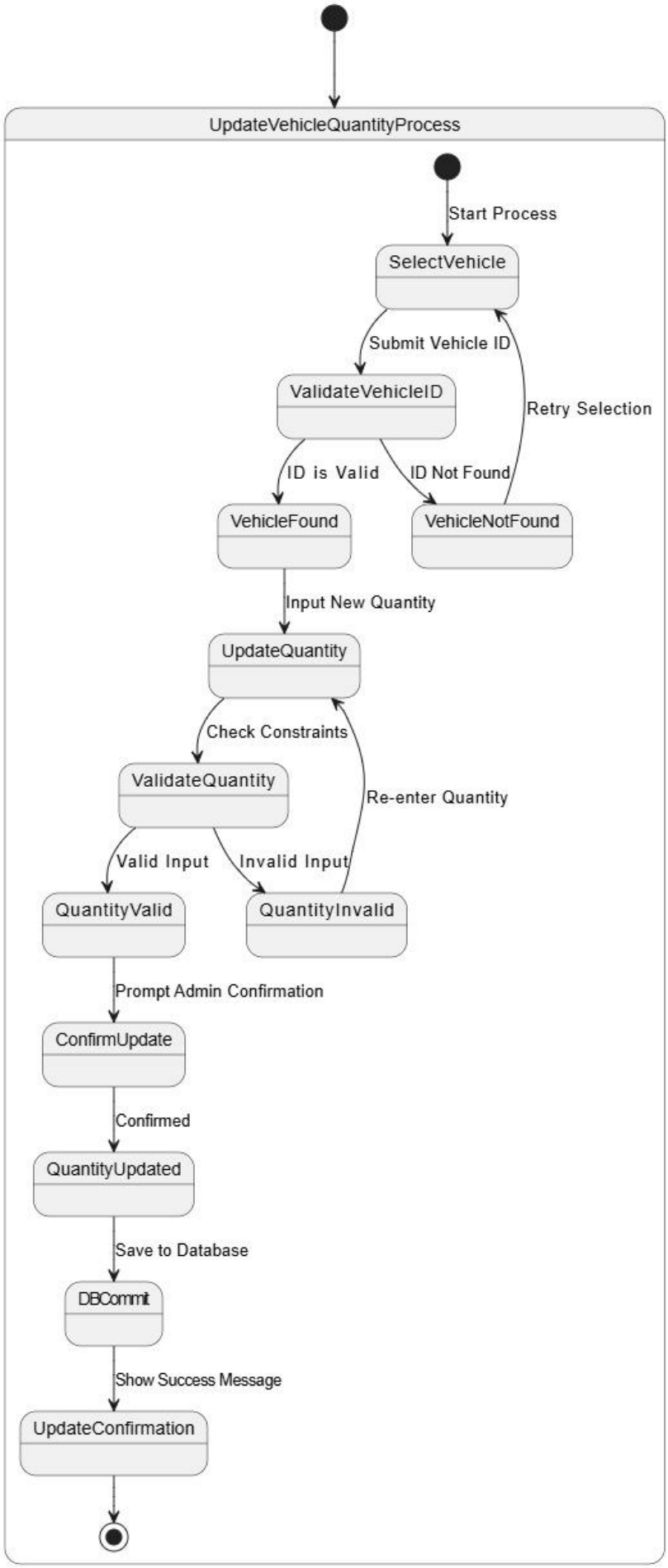
Adding a car



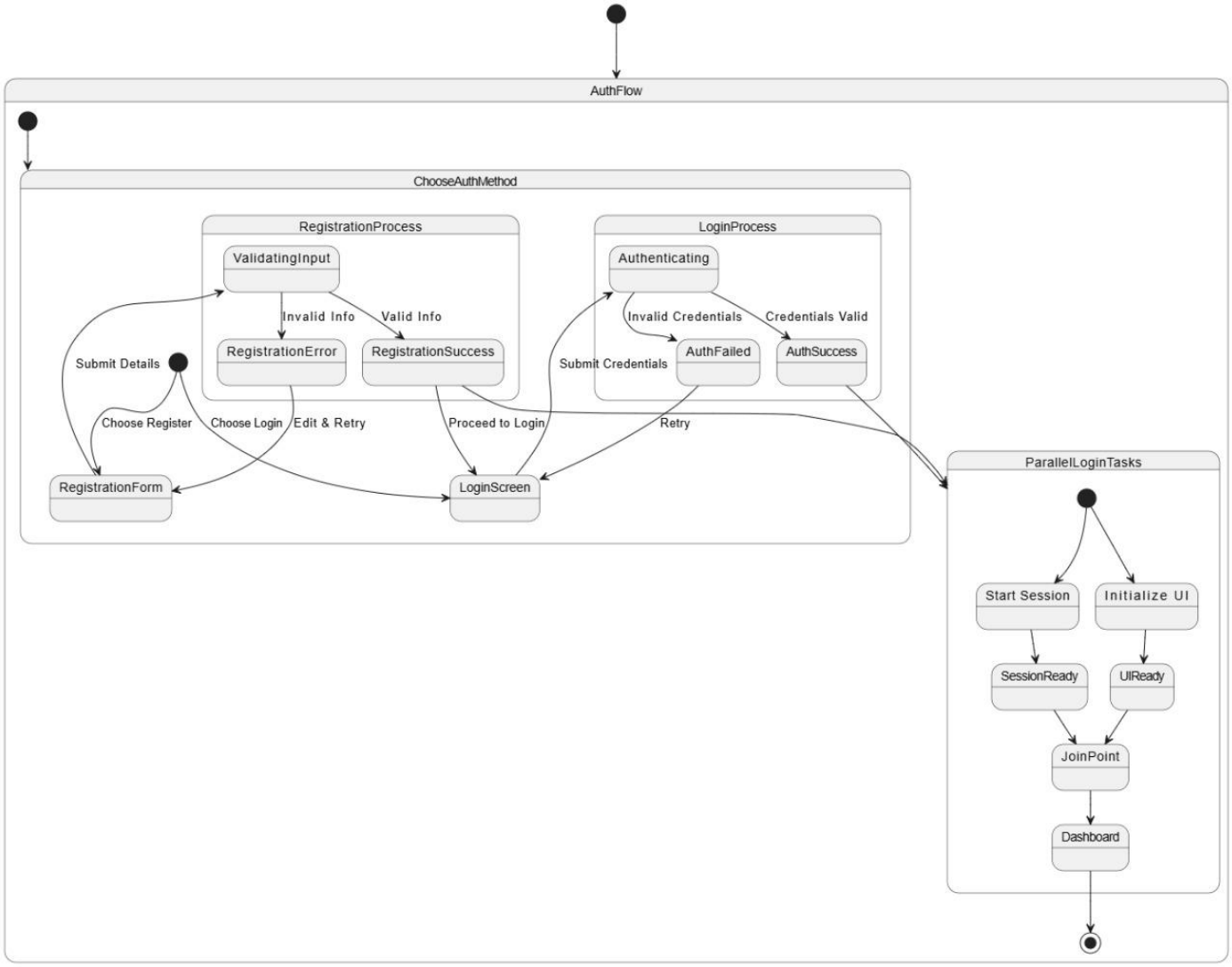
Rent


RentVehicleProcess
SelectVehicleToRent
Submit Selection
CheckAvailability
Choose Another
Vehicle Available
Unavailable
Available
NotAvailable
Enter Rental Details
ConfirmRental
ParallelProcessing
ProcessPayment
UpdateRentalHistory
Payment Confirmed
History Logged
PaymentDone
HistoryUpdated
SyncPoint
RentalSuccess
UpdateVehicleStatus

Update Vehicle Quantity



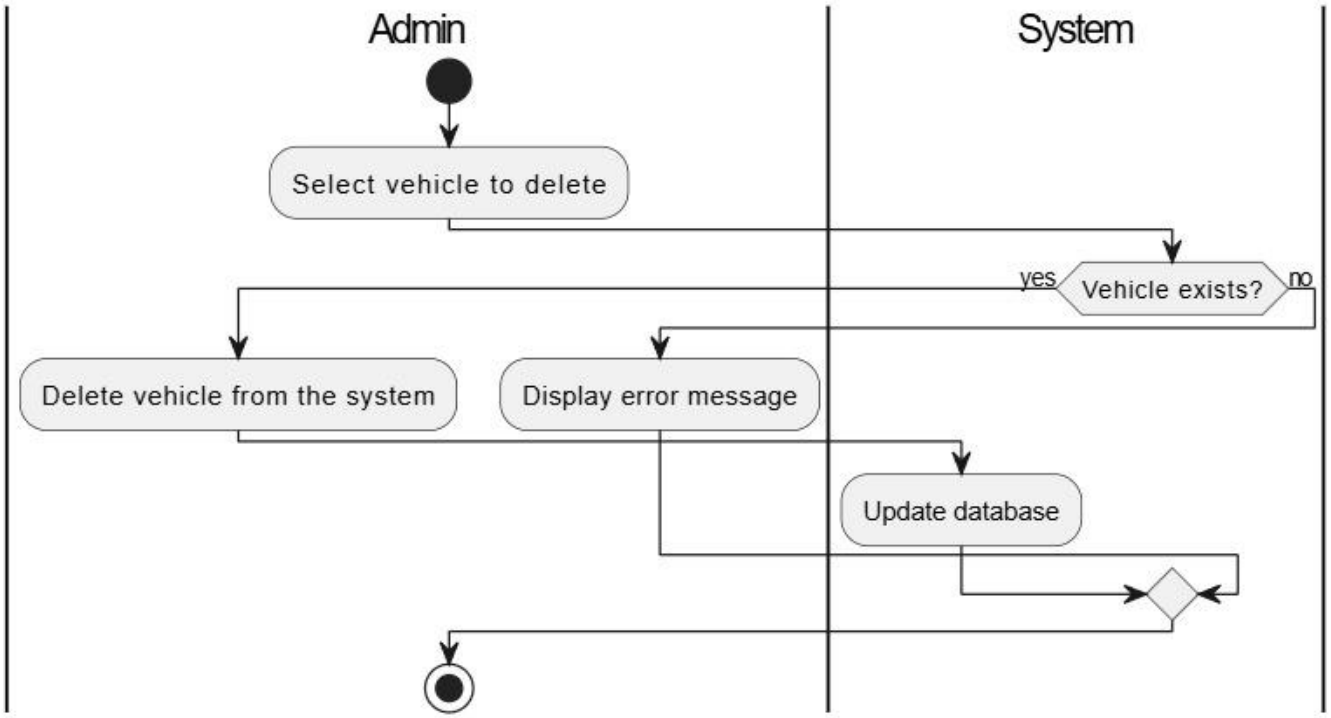
Authentication



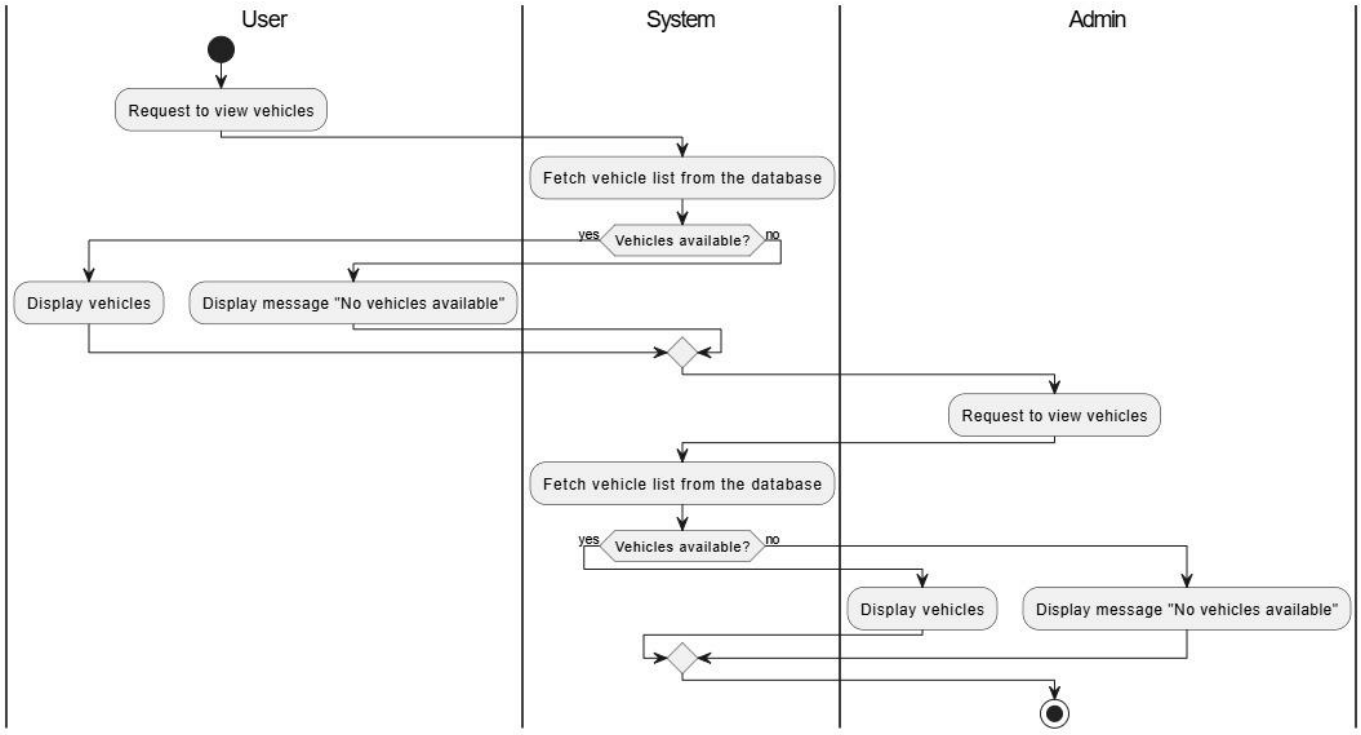
Activity Diagrams:

1. Major Usecase

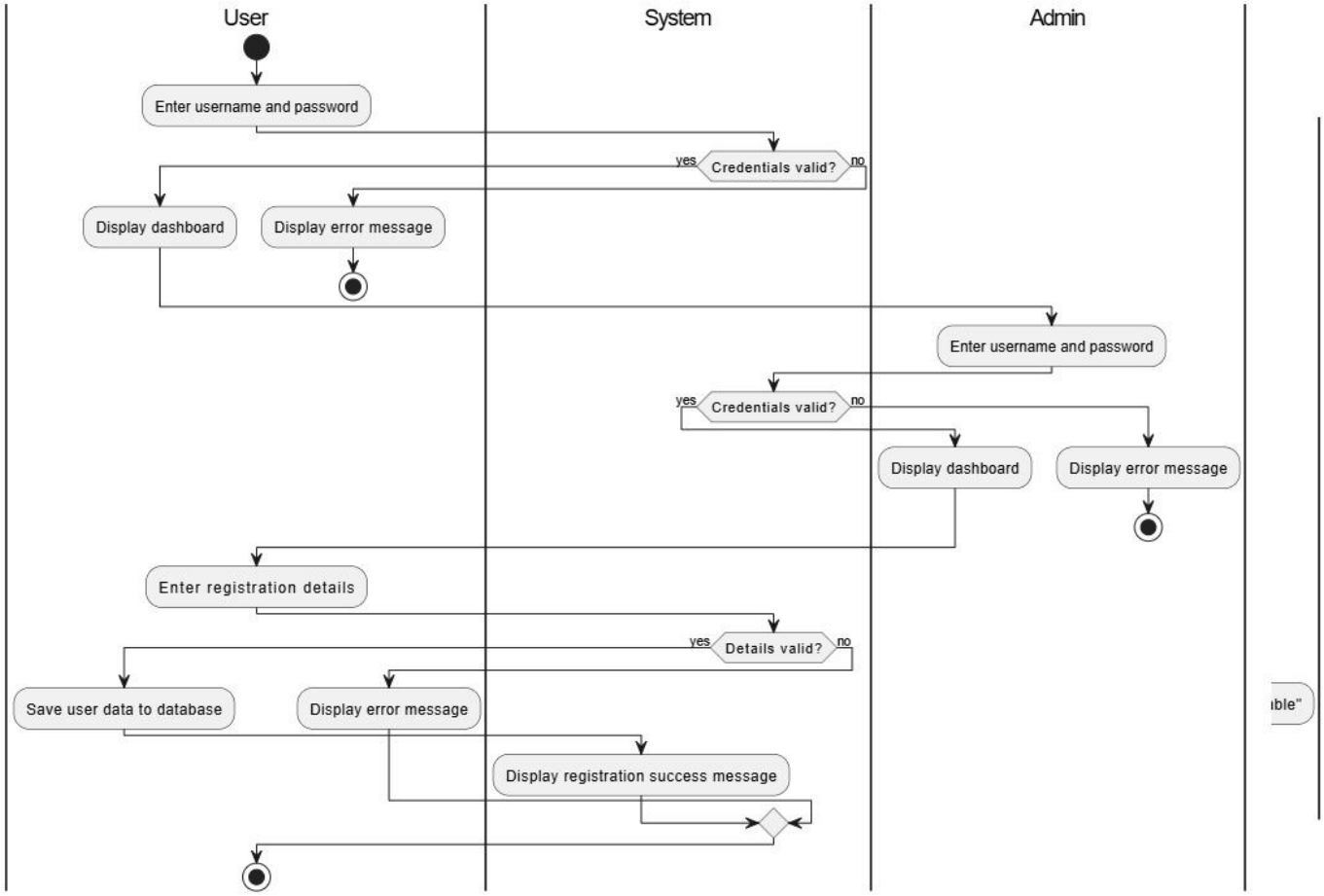
Delete



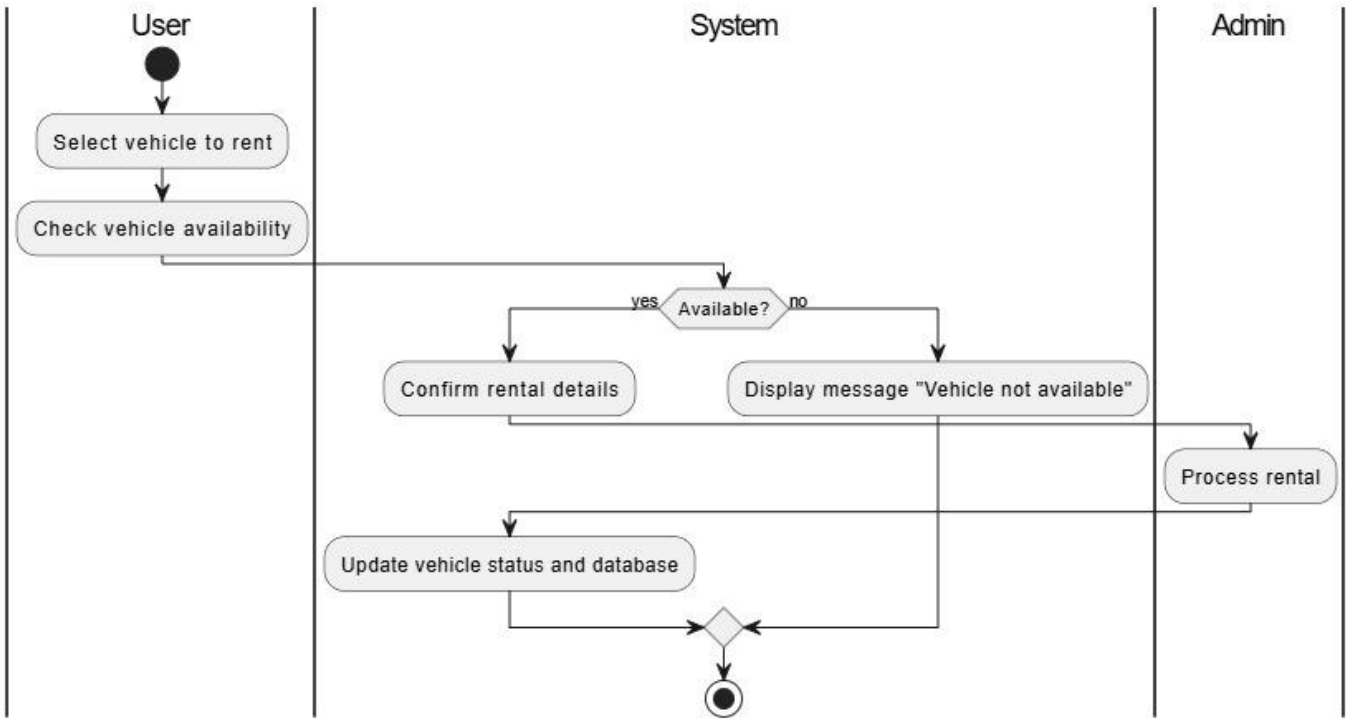
View



Authentication



Rent



Architectural Patterns:

Model -- View -- Controller Pattern (MVC)

The Vehicle Rental Management System implements the Model-View-Controller (MVC) architectural pattern to separate concerns and improve maintainability:

- Model: Represents the data and business logic of the application
  - Car.java: Represents vehicle information
  - CarModel.java: Handles database operations for vehicles
  - UserModel.java: Manages user authentication and account operations
  - RentalModel.java: Processes rental transactions
  - PaymentService.java: Handles payment calculations
- View: User interface components that display data to users and collect input
  - AddCarView.java: Interface for adding new vehicles
  - AuthGUI.java: Login and registration screens
  - AdminDashboard.java: Main interface for administrators
  - CustomerDashboard.java: Main interface for customers
  - DeleteCarView.java: Interface for removing vehicles
  - RentCarView.java: Interface for processing rentals
  - UpdateVehicleCountView.java: Interface for updating vehicle inventory
  - ViewCarsView.java: Display of available vehicles
- Controller: Intermediaries that process user input from the View and update the Model
  - AddCarController.java: Handles adding vehicles to inventory
  - AuthController.java: Manages authentication operations
  - DeleteCarController.java: Controls vehicle removal
  - RentalController.java: Processes rental requests
  - ViewCarsController.java: Retrieves vehicle information for display

Design Principles:

Single Responsibility Principle (SRP)

Each class in the system has a single responsibility. For example:

- Car.java focuses only on representing vehicle data
- RentalModel.java handles only rental-related operations
- PaymentService.java is dedicated to payment processing

Open/Closed Principle (OCP)

The system is designed to be open for extension but closed for modification:

- The vehicle system could easily be extended to support new vehicle types beyond cars and two-wheelers without modifying existing code
- New rental features could be added without changing core rental functionality

Dependency Inversion Principle (DIP)

Higher-level modules depend on abstractions rather than concrete implementations:

- Controllers depend on Model interfaces rather than concrete implementations
- Views communicate with Controllers rather than directly with Models

Interface Segregation Principle (ISP)

The system avoids forcing clients to depend on methods they don't use by creating specific controllers for different functionalities:

- AddCarController for adding vehicles
- DeleteCarController for removing vehicles

ViewCarsController for viewing vehicles

Design Patterns:

Factory Method Pattern

The system implements the Factory Method pattern in the controller initialization within Main.java:

- Creates instances of controllers with appropriate dependencies
- Centralizes object creation logic

Singleton Pattern (Implied)

Database connections in the Model classes follow a singleton-like pattern:

- Connection is established when needed and closed after operations
- Prevents redundant connection objects

Facade Pattern

The Controller classes act as facades that simplify complex subsystem interactions:

- RentalController provides a simplified interface for the rental process
- AuthController abstracts the authentication mechanisms