Software Architecture Document

Version 2.0

for

Star Car Rentals

Prepared by

Name	Student ID	Email
Vaibhav Malhotra (Team Lead)	40079373	malhotra.vaibhav0304@gmail.com
Adeyinka Areje	40088288	adeyinka.areje@icloud.com

Instructor: ***Dr. C. Constantinides***

Course: SOEN 6461

Date: 24th November 2019

Document history

Date	Version	Description	Author
8th Nov	1.0	Iteration 3	Team 3
24th Nov	2.0	Iteration 4	Team 3

Table of contents

1. Introduction

This document is a detailed architectural documentation for a Car Rental System. It includes the purpose, scope definitions acronyms/abbreviations, references, overview and constraints of the system.

Purpose

This document provides a comprehensive architecture overview of the system, with an intent to capture and convey the significant architectural decisions which have been made on the system. The main goal of this document is to capture the high level design of the Car Rental System which the development team or stakeholders, require for a smooth implementation and future maintenance.

Scope

This Software Architecture Document provides and an architectural overview of a car Rental System. It describes the functional and nonfunctional requirements as well as the constraints of the system. The intent is to support the members of Star Car Rental Service in their day to day business activities.

Definitions, acronyms, and abbreviations

RUP:Rational Unified Process
UML:Unified Modeling Language
SAD:Software Architecture Document
TDG:Table Data Gateway
OS:Operating SYSTEM
UOW:Unit Of Work

2. Architectural representation

In this document, the architecture is presented as a series of views modeled using the 4+1 view which is illustrated below:

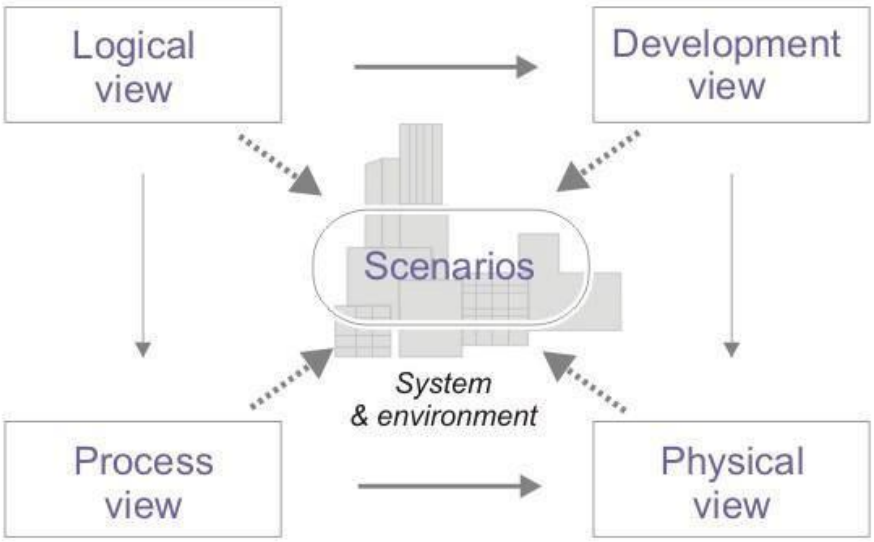

Figure 1: The 4+1 view model.

1. Logical view:

Audience: Designers

In this view, the System sequence diagrams highlight the critical use cases in the system, and the interaction diagrams represent how each use case actually interacts with the system:

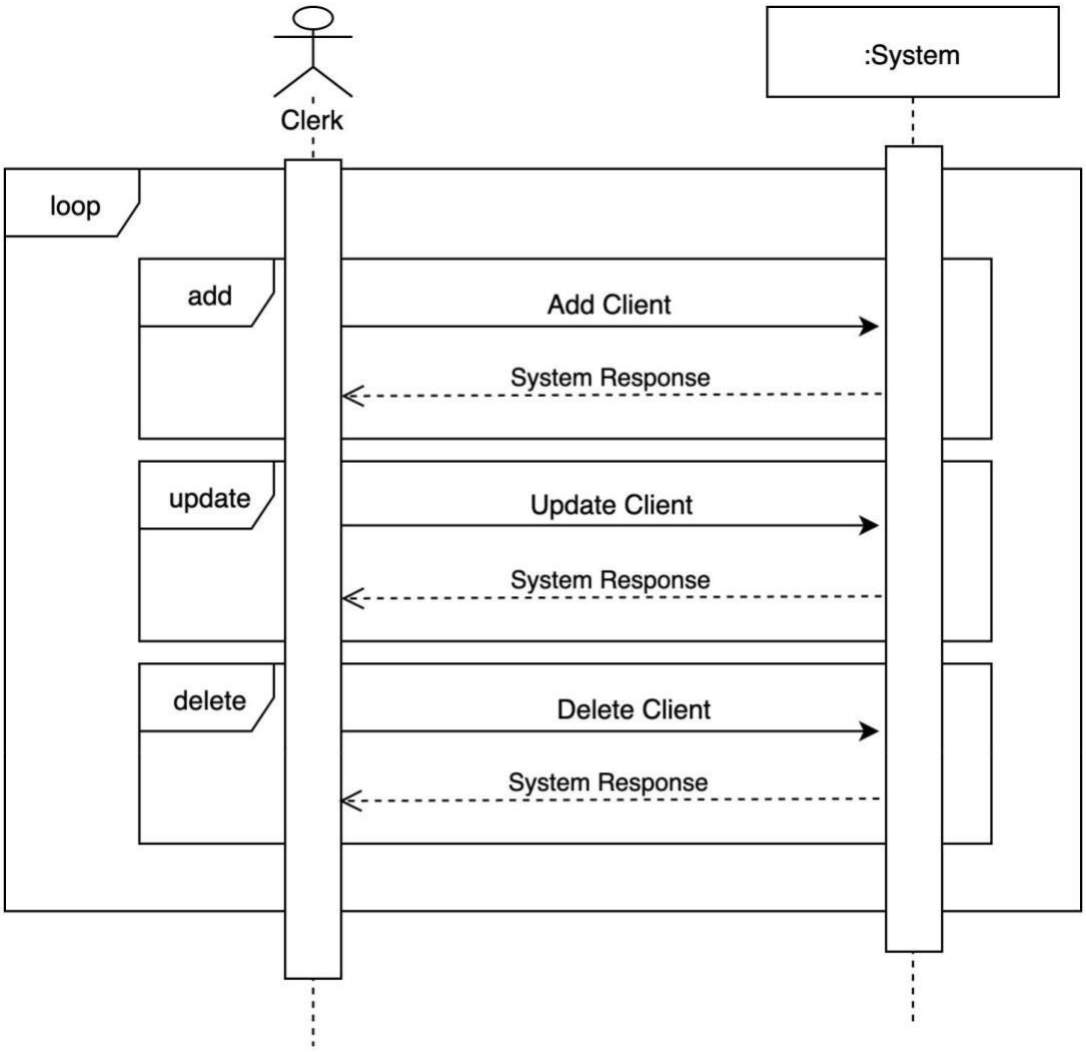

Fig 1.1 System Sequence Diagram - Manage Client

Contract CO1: CreateClient

Operation: createClient()

CrossReferences: **Use Case: Manage Client**

Preconditions:
- There is a new client being created.
- A record with same clientLicenseNumber does not exist in database.

Postconditions:
- A Client instance was created.
- The instance is verified and registered to clientUoW.
- A new client row successfully added to client the database.

Contract CO2: UpdateClient

Operation: updateClient()

CrossReferences: **Use Case: Manage Client**

Preconditions:
- A client with the requested record exists in the client UOW or database.
- Information modified is valid.

Postconditions:
- The Client instance fetched was updated successfully.
- The instance is verified and registered to client UOW.
- The respective client row successfully updated in the client database.

Contract CO3: DeleteClient

Operation: deleteClient()

CrossReferences: **Use Case: Manage Client**

Preconditions:
- A client with the requested record exists in the database.
-The client instance is not a part of clientUOW for addition or updation.

Postconditions:
- A Client instance fetched and added to client UOW for deletion.
- The respective client row deleted from the database.

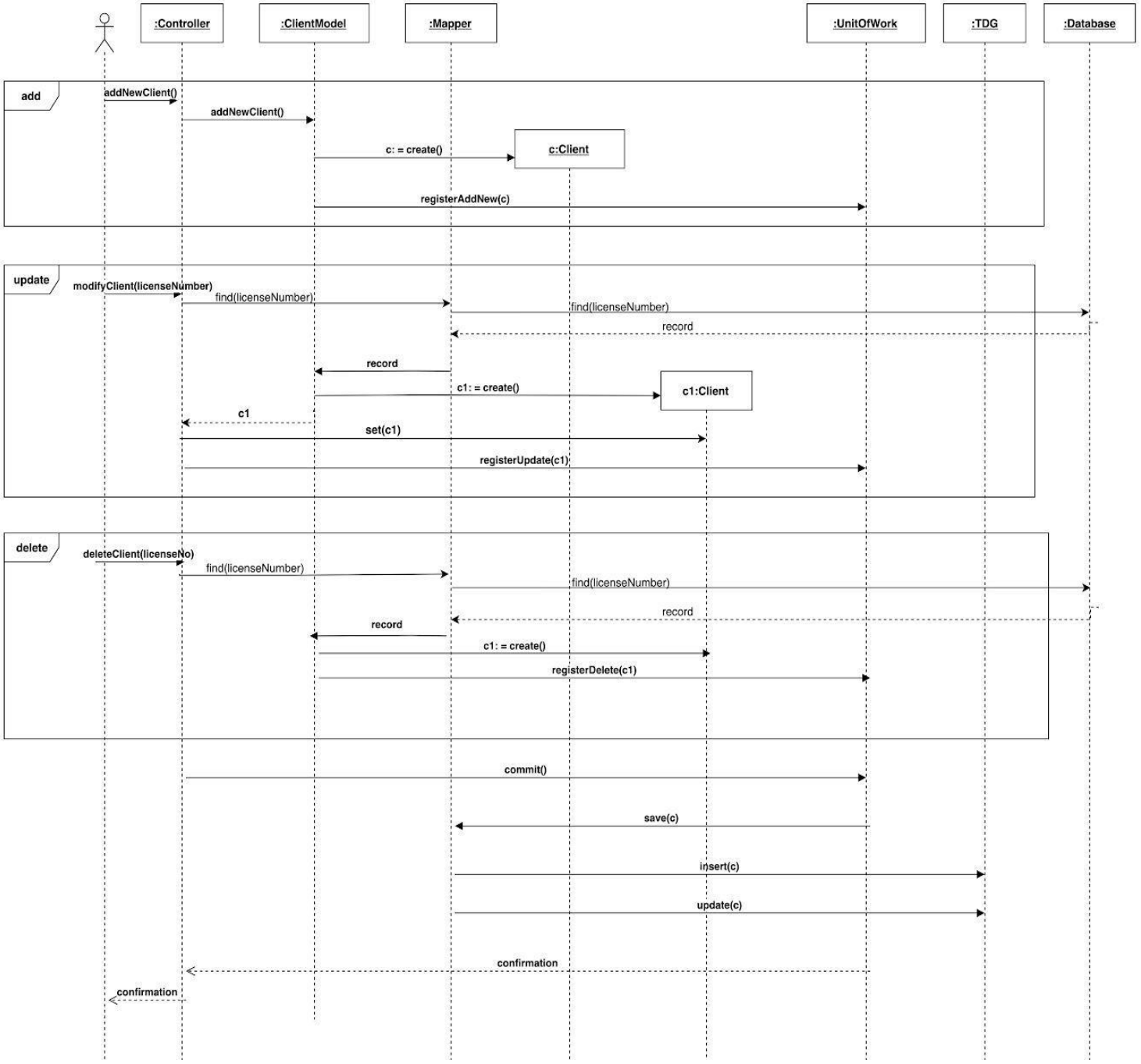

Fig 1.2 Sequence Diagram for Manage Client

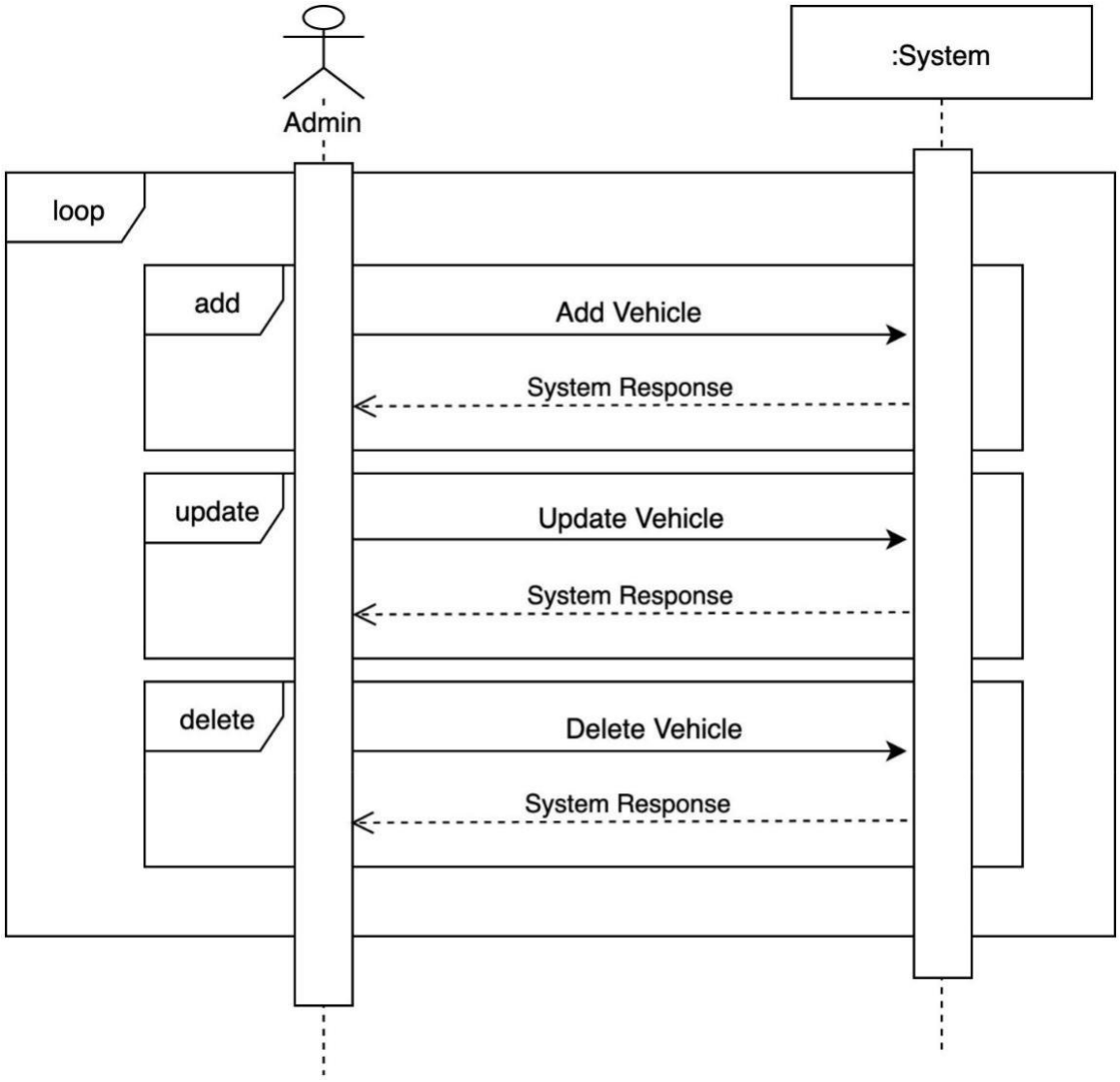

Fig 1.3 System Sequence Diagram - Manage Vehicle Catalog

Contract CO8: **CreateVehicle**

Operation: createVehicle()

CrossReferences: **Use Case: Manage Vehicle**

Preconditions: - There is a new vehicle being created.

- A record with same vehicleLicenseNumber does not exist in database.

Postconditions: - A Vehicle instance v is created.

- The new instance v, is registered with the vehicle UOW for creation.
- The newly created vehicle v is inserted into the database.

Contract CO9: **UpdateVehicle**

Operation: createVehicle()

CrossReferences: **Use Case: Manage Vehicle**

Preconditions: - The requested vehicle record exists in the database or vehicle UOW.

- Information modified is valid.

Postconditions: - The Vehicle instance v, to be modified is updated successfully.

- The instance v, is registered with the vehicle UOWork for an update.
- The instance v in respective vehicle row in the database is updated.

Contract CO10: **DeleteVehicle**

Operation: deleteVehicle()

CrossReferences: **Use Case: Manage Vehicle**

Preconditions: - A Vehicle with the requested record exists in the database.

Postconditions: - A Vehicle instance v to be deleted is fetched.

- The instance v is registered with the vehicle UOW for a delete.
- The instance v is successfully removed from the database.

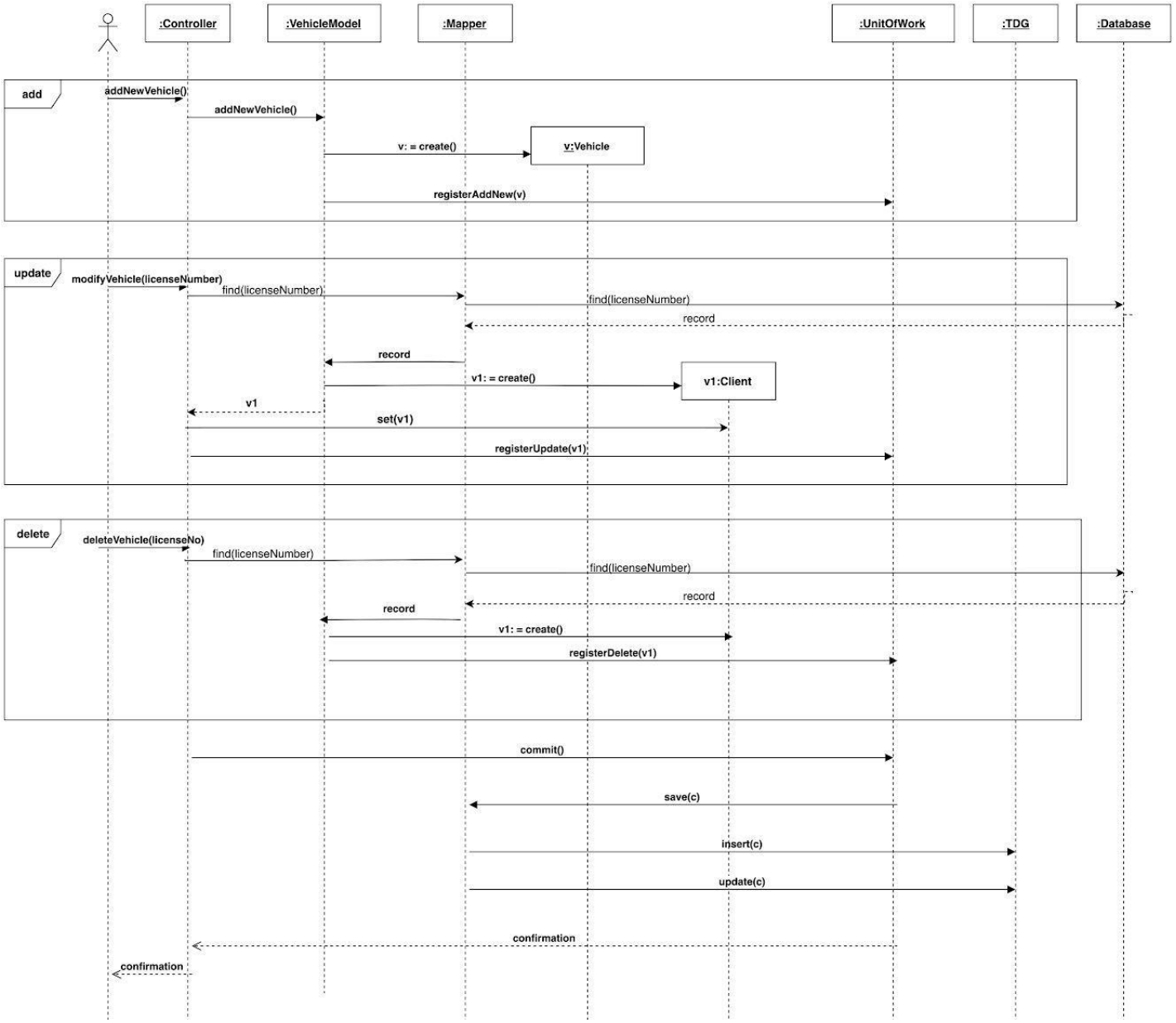

Fig 1.4 Sequence Diagram - Manage Vehicle Catalog

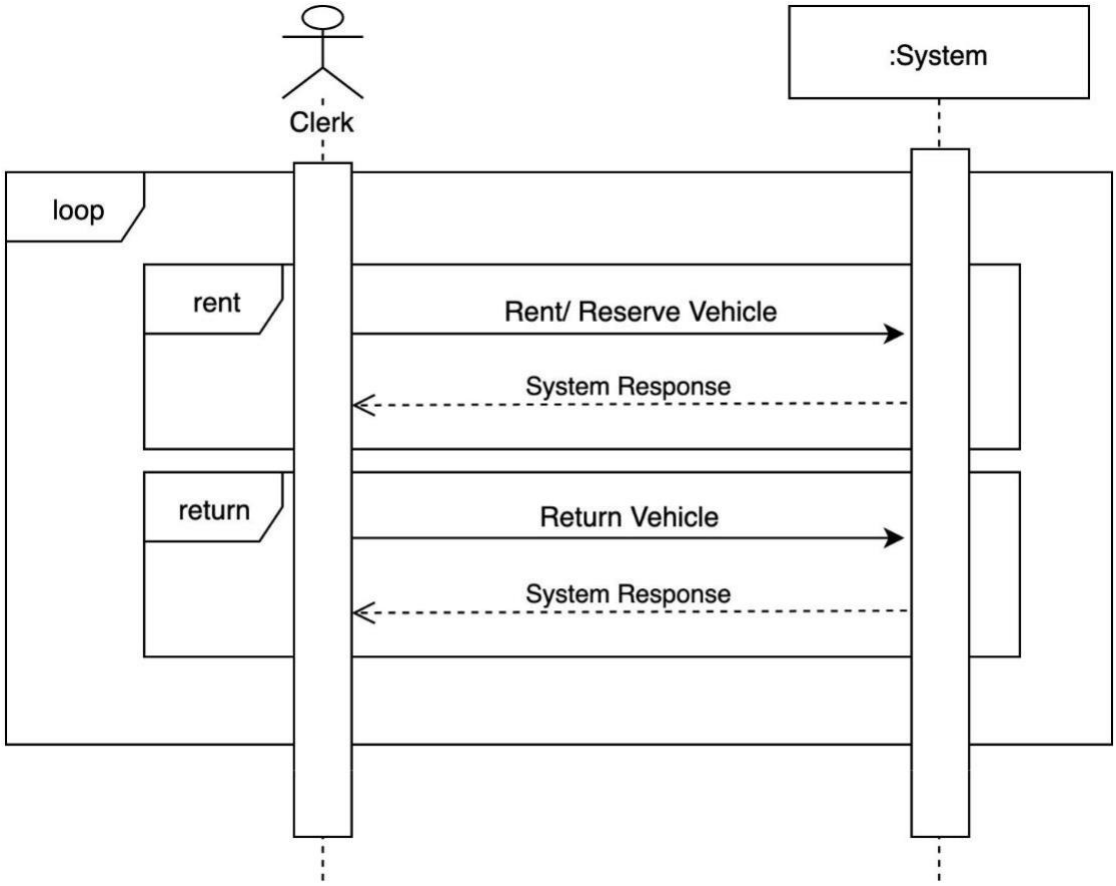

Fig 1.5 System Sequence Diagram - Manage Rentals/Reservations

Contract CO4: **CreateRental**

Operation: createRental()

CrossReferences: **Use Case: Create Rental**

Preconditions: - A record with clientLicenseNumber exists in database.

- A record with vehicleLicenseNumber exists in database and is available.

Postconditions: - A vehicle instance *newV* fetched and transaction instance *newT* created.

- *newV* is updated with Rented status.
- newT is associated with clientLicenseNumber, vehicleLicenceNumber and timeStamp.

- newV is verified and registered to vehicleUOW for update.

- The respective vehicle row updated in vehicle database.

- newR is verified and registered to transactionUOW for addition.

- A new transaction row for *newT*added to transaction database.

Contract CO5: **ReturnRental**

Operation: returnRental()

CrossReferences: **Use Case: Return rental**

Preconditions: - A Transaction with matching rental with the requested record exists in the database.

Postconditions: - A vehicle instance *newV* fetched and transaction instance *newT* created.

- *newV* is updated with Rented status.
- newT is associated with clientLicenseNumber, vehicleLicenceNumber and timeStamp.

- newV is verified and registered to vehicleUOW for update.

- The respective vehicle row updated in vehicle database.

- newR is verified and registered to transactionUOW for addition.

- A new transaction row for *newT*added to transaction database.

Contract CO6: **CreateReservation**

Operation: createReservation()

CrossReferences: **Use Case: Create Reservation**

Preconditions: - A record with clientLicenseNumber exists in database.

- A record with vehicleLicenseNumber exists in database and is available.

Postconditions: - A vehicle instance *newV* fetched and transaction instance *newT* created.

- *newV* is updated with Reserved status.
- newT is associated with clientLicenseNumber, vehicleLicenceNumber and timeStamp.

- newV is verified and registered to vehicleUOW for update.

- The respective vehicle row updated in vehicle database.

- newR is verified and registered to transactionUOW for addition.

- A new transaction row for *newT*added to transaction database.

Contract CO7: **ReturnReservation**

Operation: returnReservation()

CrossReferences: **Use Case: Return Reservation**

Preconditions: - A Transaction with matching reservation with the requested record exists in the database.

Postconditions: - A vehicle instance *newV* fetched and transaction instance *newT* created.

- *newV* is updated with UnReserved status.
- newT is associated with clientLicenseNumber, vehicleLicenceNumber and timeStamp.

- newV is verified and registered to vehicleUOW for update.
- The respective vehicle row updated in vehicle database.
- newR is verified and registered to transactionUOW for addition.
- A new transaction row for *newT*added to transaction database.

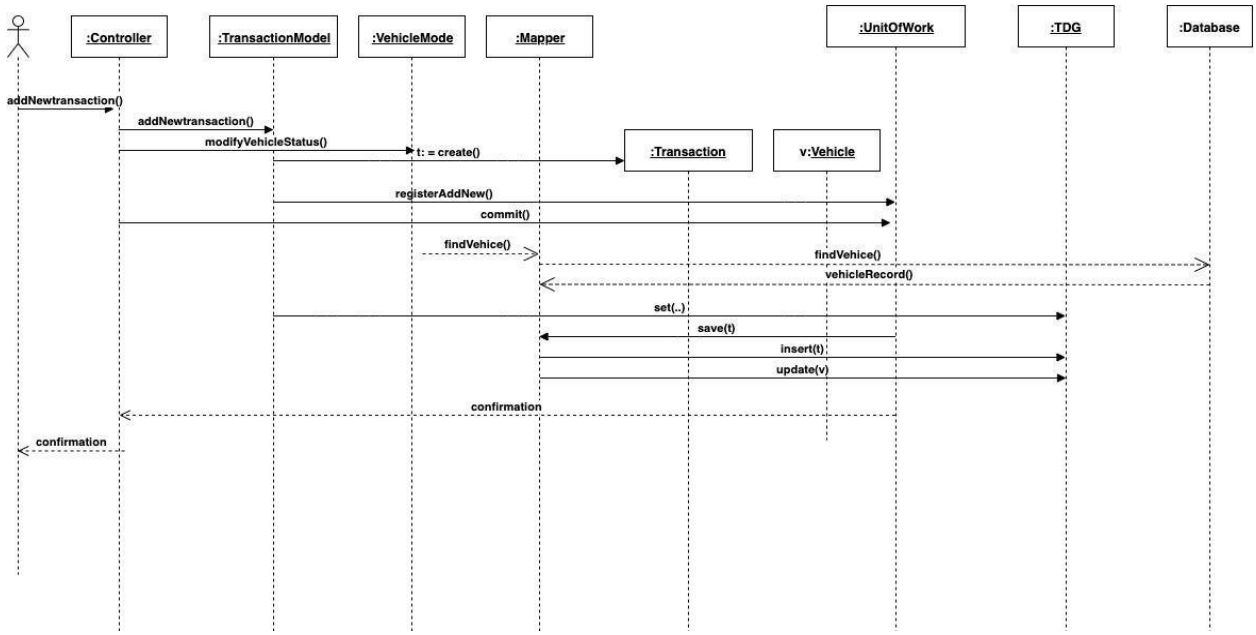

Fig 1.6 Sequence Diagram - Manage Rentals/Reservations

2. **Development view**(also known as Implementation view):
   Audience: Programmers.

   The development view illustrates a system from a programmer's perspective and is concerned with software management. It uses the UML Component diagram to describe system components.

3. **Process view**:
   Audience: Integrators.

   The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML Diagrams to represent process view include the **Activity diagram**.

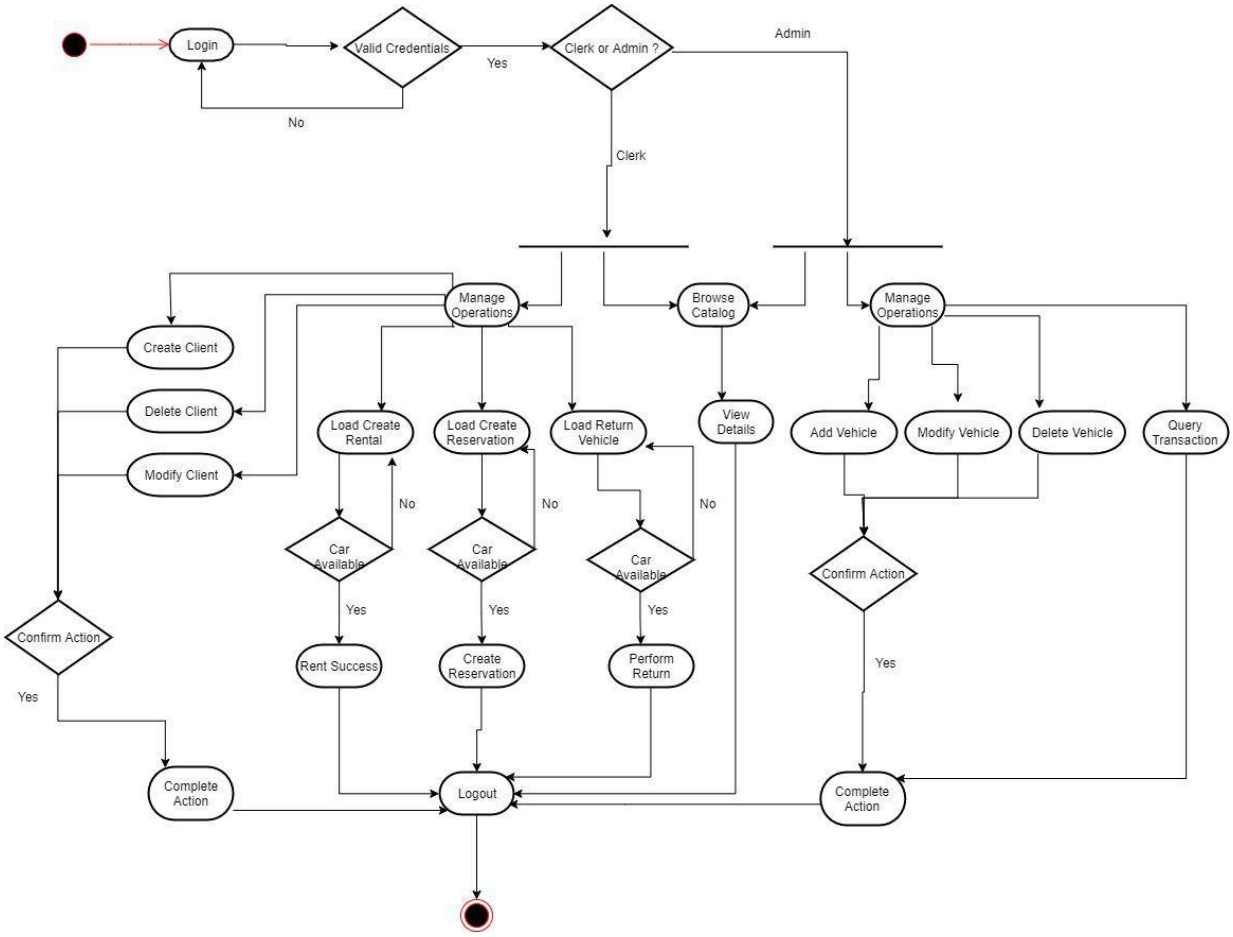

Figure 3.1 Activity Diagram

4. **Physical view**(also known as deployment view) :

Audience: Deployment managers.

The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. UML Diagrams used to represent physical view include the **Deployment diagram**.

5. Use case view(also known as Scenarios) :

Audience: all the stakeholders of the system, including the end-users.

The description of the architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. Related Artifacts : **Use-Case Model**.

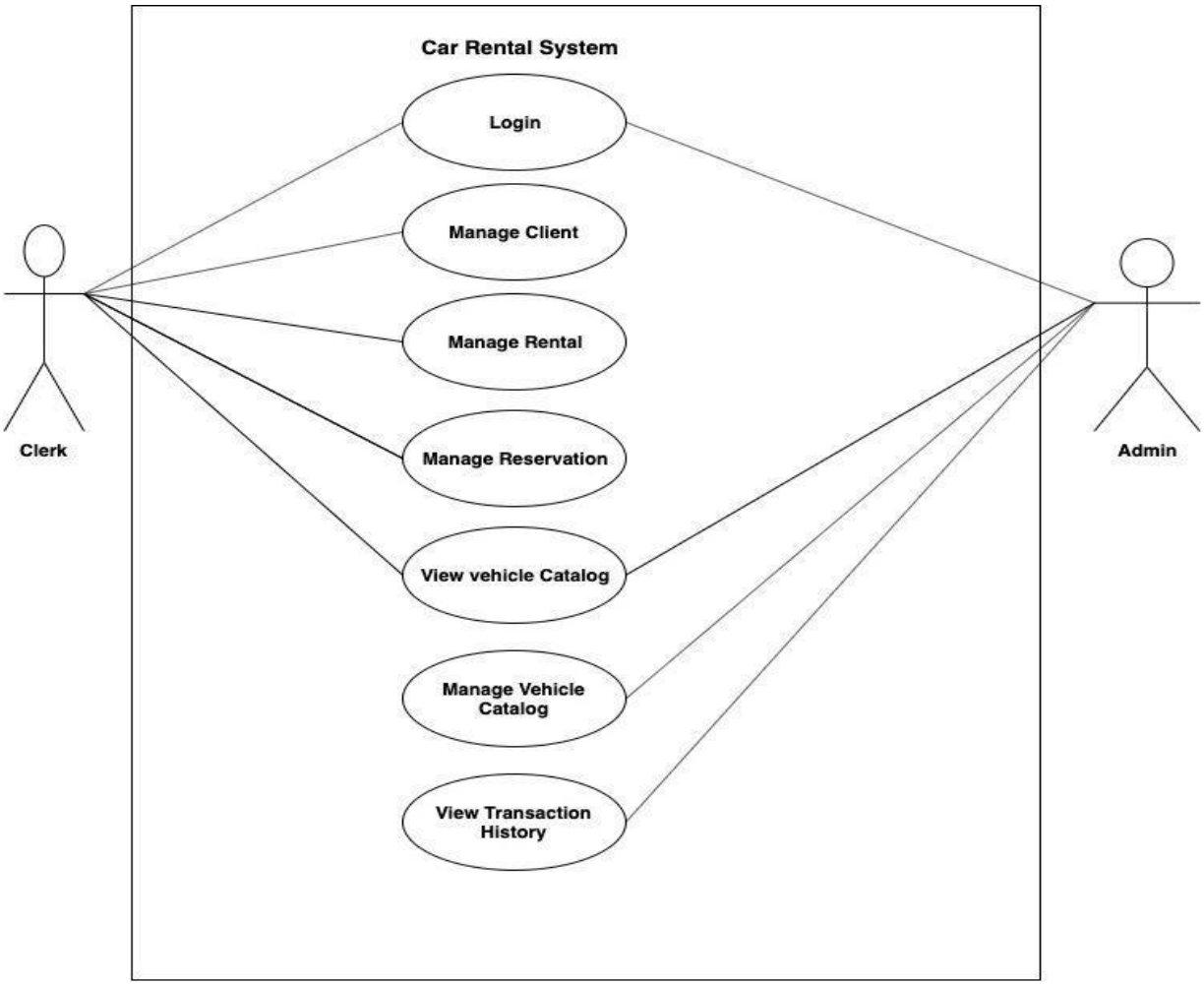

Figure 5.1 Use-Case Model

6. **Data view**(optional): Audience: Data specialists, Database administrators. Describes the architecturally significant persistent elements in the data model . Related Artifacts: **Data model**.

3. Architectural requirements: goals and constraints

Requirements are already described in SRS. In this section describe *key*requirements and constraints that have a significant impact on the architecture.

Functional requirements (Use case view)

Refer to Use Cases or Use Case scenarios which are relevant with respect to the software architecture. The Use Cases referred to should contain central functionality, many architectural elements or specific delicate parts of the architecture.

The overview below refers to architecturally relevant Use Cases from the Use Case Model (see references).

Source	Name		Architectural relevance	Addressed in:
System Login	login		If a user(Clerk) does not log in, no functionality in the car rental system can	SRS- SECTION 3

		be achieved for example, a car cannot be rented out.	
Client record Management	Manage Client(CRUD)	Since this is responsible for creating a client's record in to the system and modifying the records, without this a client will be unable to make any car reservation or rental.	SRS- SECTION 3
Vehicle Catalog	View catalog	This is required to show all the available cars and the cars that can be rented out or have already been rented out, without this, a clerk is unable to access the cars available that a client can rent out.	SRS- SECTION 3
Create Rental	Create rental	The main motive of the system is to create a rental for a client.	SRS- SECTION 3

Create Reservation	Create Reservation	The main motive of the system is to allow a clerk to create a rental for a client.	SRS- SECTION 3
Car Returns	Return Vehicle	This allows a clerk to return the vehicle the client rented out.	SRS- SECTION 3
Reservation Cancellation	Cancel Reservation	This allows a clerk to cancel a reservation made for a client.	SRS- SECTION 3
Vehicle record management	Manage Vehicle Record (CRUD)	This allows the Admin to create and modify vehicle records in the catalog, without this a clerk will be unable to make any car reservation for a client. Also, the clerks and the admins will not be able to view the catalog.	SRS- SECTION 3
View Transaction History	View Transactions	This use case addressed the functionality that allows the admin to run queries on the transaction log in	SRS- SECTION 3

		order to view all the transactions were carried out in the system namely car rentals or reservations.	

Non-functional requirements

Describe the architecturally relevant non-functional requirements, i.e. those which are important for developing the software architecture. Think of security, privacy, third-party products, system dependencies, distribution and reuse. Also environmental factors such as context, design, implementation strategy, team composition, development tools, time to market, use of legacy code may be addressed.

Usually, the non-functional requirements are already in place and can be referenced here. This document is not meant to be the source of non-functional requirements, but to address them. Provide a reference per requirement, and where the requirement is addressed.

Source	Name	Architectural relevance	Addressed in:
Start Rental SRS	Design Constraint	This is required for the successful development and implementation of the system.	Section 3.3

Start Rental SRS	Compatibility	This allows the system to launch the application on any browser or on any OS.	Section 3.3
Start Rental SRS	Usability	provides ease of Use for the stakeholders namely the clerk or Admin.	Section 3.3

4. Logical view

The logical view captures the functionality provided by the system; it illustrates the collaborations between system components in order to realize the system's use cases.

Layers, tiers etc.

Describe the top-level architecture style. Deploy a *UML class diagram*.

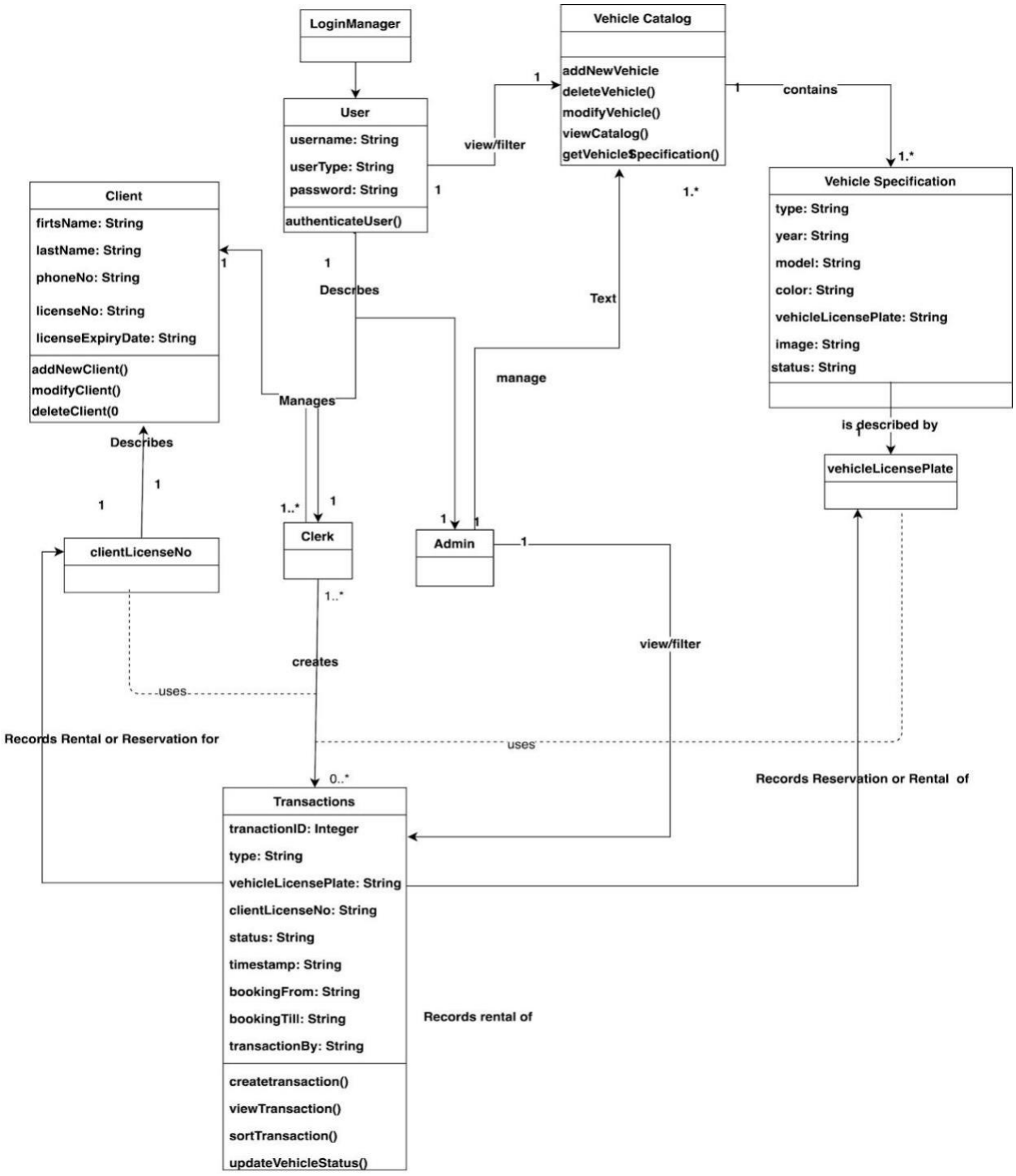

Figure 9.1 Class Diagram

Architecturally significant design packages

Describe packages of individual subsystems that are architecturally significant. For each package includes a subsection with its name, its brief description, and a diagram with all significant classes and packages contained within the package.

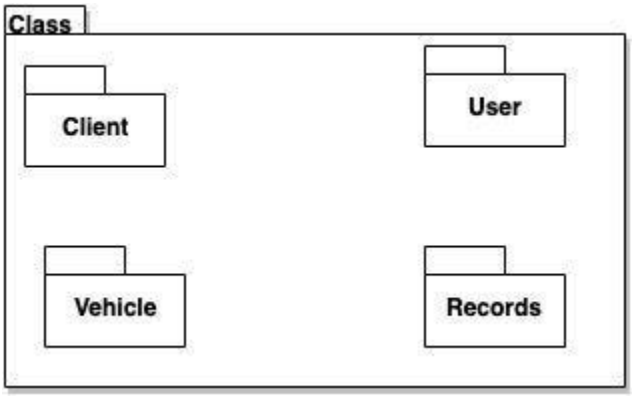

Figure 9.2 Class Diagram Package Diagram

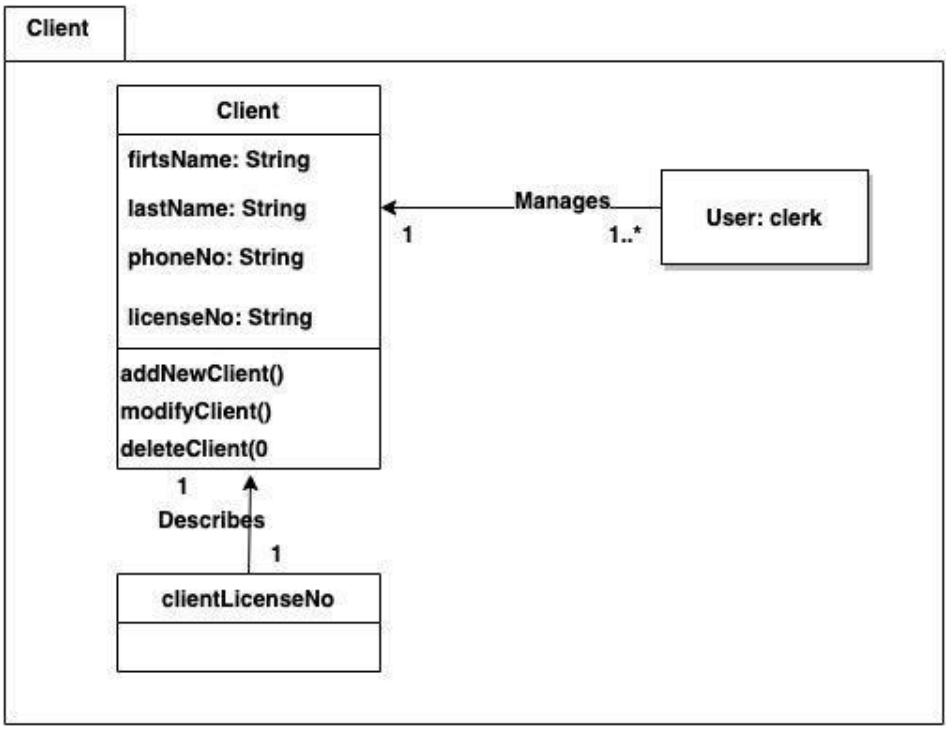

Figure 9.3 Client Package Diagram

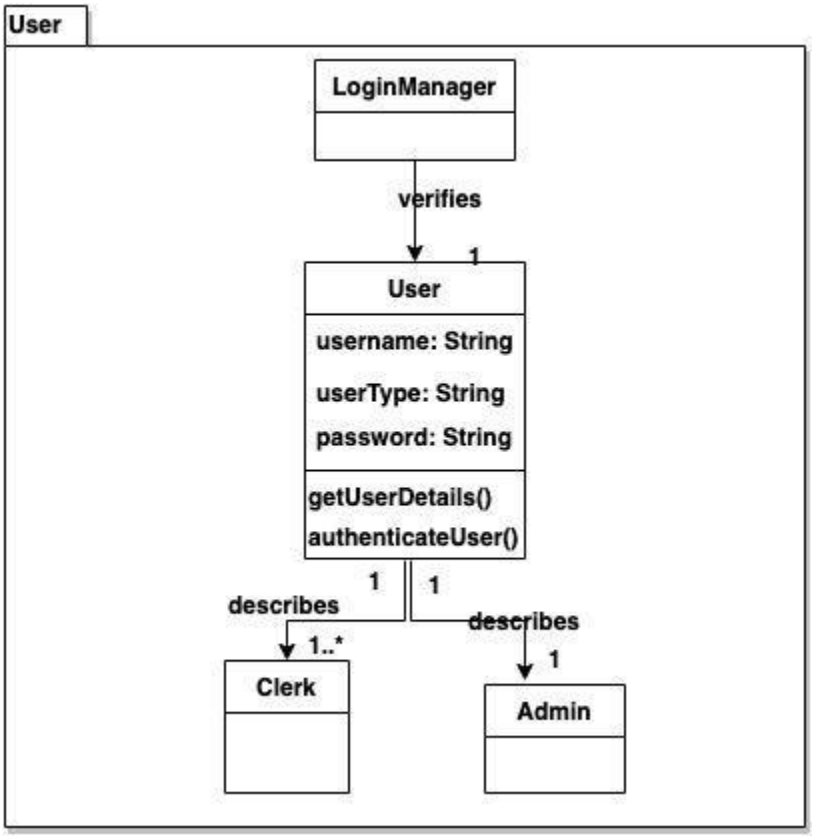

Figure 9.4 User Package Diagram

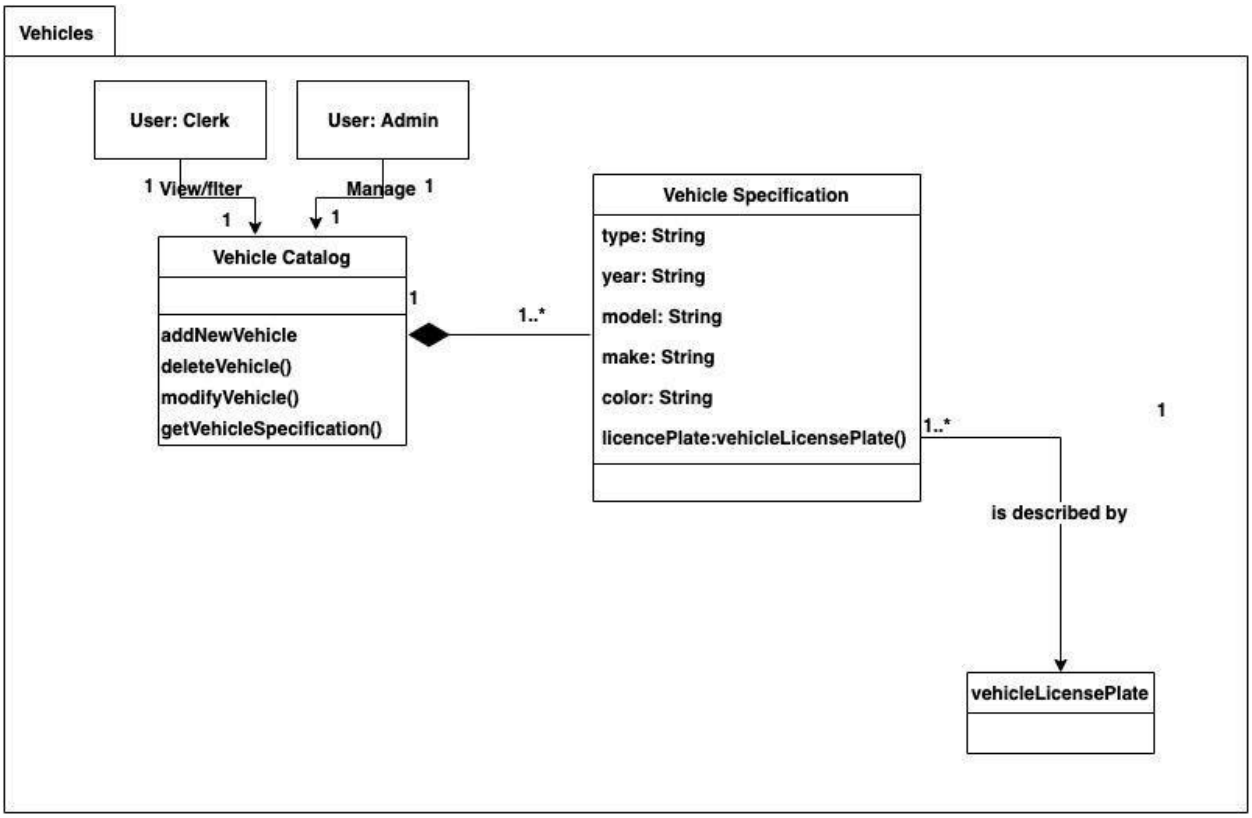

Figure 9.5 Vehicle Package Diagram

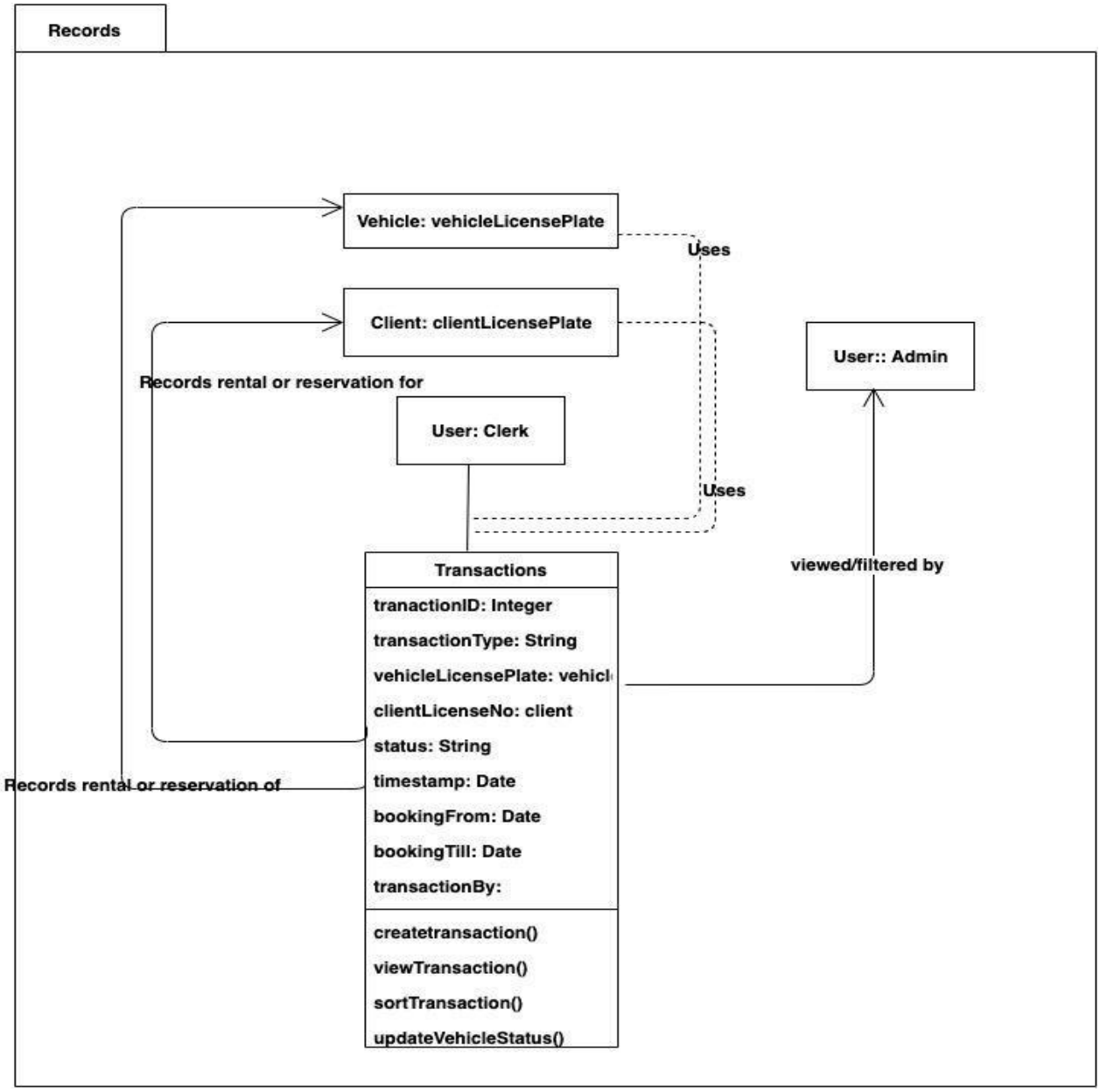

Figure 9.6 Records Package Diagram

Subsystems

The system can be further seen as a collection of many subsystems. The subsystem in the application are as follows:

- Model
- View
- Controller
- DataMapper
- Unit Of Work

Use case realizations

In this section you have to illustrate how use cases are translated into *UML interaction diagrams*.Give examples of the way in which the Use Case Specifications are technically translated into Use Case Realizations, for example, by providing a sequence-diagram. Explain how the tiers communicate and clarify how the components or objects used realize the functionality.

For each critical use case, a system sequence diagram has been created, depicting the communication between the actor and the system. **They are listed in section 4.**

4. Development (Implementation) view

The development (or implementation) view describes the components used to assemble the system. Use a *UML component diagram*to capture this view.

Reuse of components and frameworks

The third party applications used in this project are listed below:

- Spring Boot Framework - Spring Boot is an open source Java-based framework used to create a micro Service. It is an architecture that allows developers to develop and deploy services independently.
  https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm
- Bootstrap Themes - is a framework to help you design websites faster and easier. It includes HTML and CSS based design templates for typography, forms, buttons, tables, navigation, modals, image carousels, etc.
  https://www.htmlgoodies.com/html5/markup/10-common-uses-of-bootstrap.html

5. Deployment (Physical) view

The deployment (or physical) view illustrates the physical components of the architecture, their connectors and their topology. Describe the physical network and hardware configurations on which the software will be deployed. This includes at least the various physical nodes (computers, CPUs), the interaction between (sub)systems and the connections between these nodes (bus, LAN, point-to-point, messaging, SOAP, http, http).

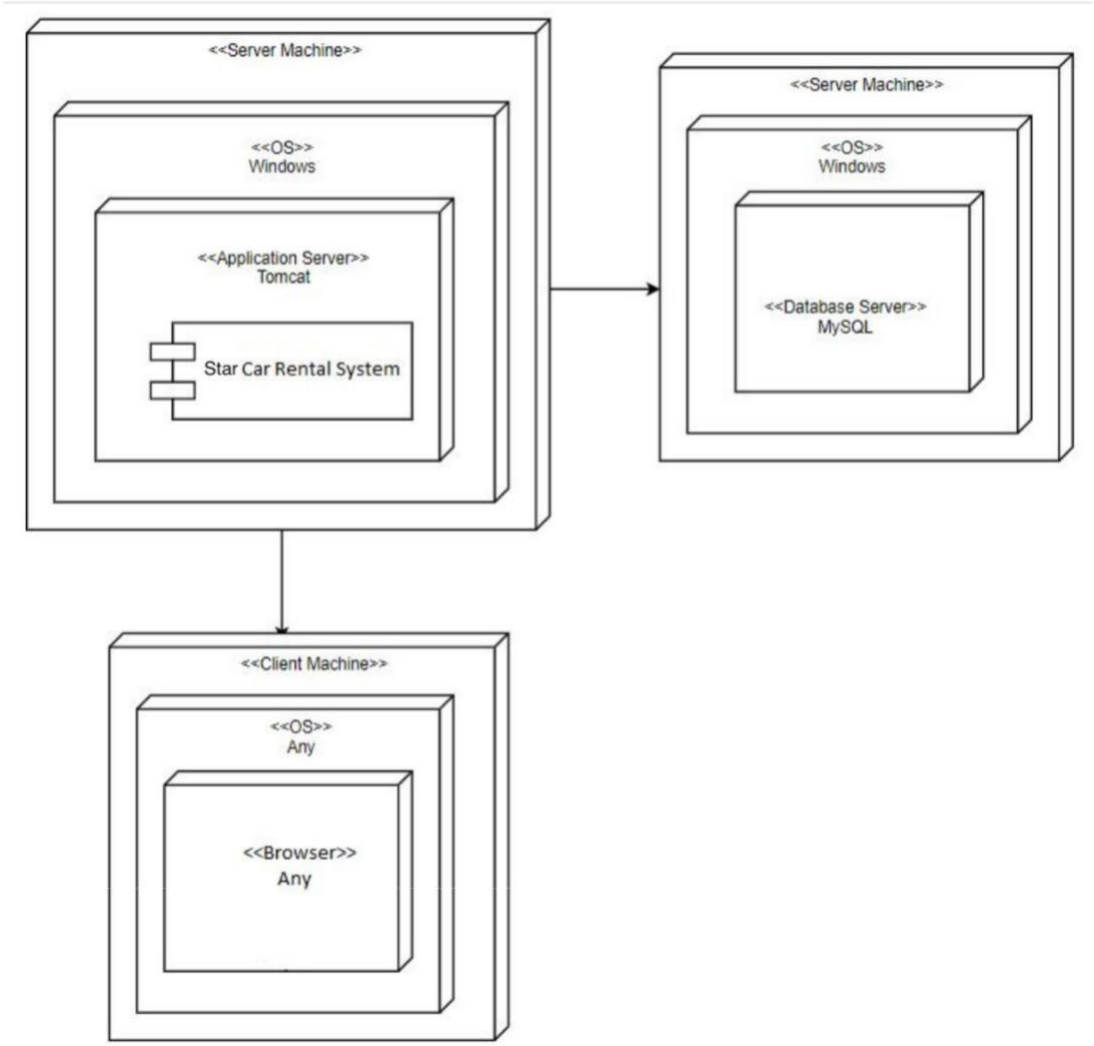

Figure 12.1 Deployment Diagram

Name	Type	Description
Name of the node.	Node type.	Technical specifications.
Client	AnyOS	Need a web browser
Server	AnyOS	Machine that supports Java 8

6. Data view (optional)

An enterprise software system would additionally require a data view. The data view describes the data entities and their relationships. Deploy an *Entity-Relationship* (ER) *Model*to represent this view. Note that the ER model is not part of the UML specification. Additionally you can deploy a UML class diagram to represent the data view where classes would correspond to data entities.

7. Quality

A description of how the software architecture contributes to the quality attributes of the system as described in the ISO-9126 (I) standard. **For example**:The following quality goals have been identified:

Scalability:

- Description : System's reaction when user demands increase
- Solution : J2EE application servers support several workload management techniques

Reliability, Availability:

- Description : Transparent failover mechanism, mean-time-between-failure
- Solution : : J2EE application server supports load balancing through

clusters Portability:

- Description : Ability to be reused in another environment
- Solution : The system me be fully J2EE compliant and thus can be deploy onto any J2EE application server

- Description : Authentication and authorization mechanisms
- Solution : J2EE native security mechanisms will be reused

8. Design Patterns Implemented

Design Pattern	Description	Rationale
Architectural Design Pattern	MVC(Model View Controller)	Provides clear division between data, logic and UI. Model is independent from View and independent of controller. **Model**houses all the objects in the system namely, vehicle, user, client and transaction. These objects contain data that are stored in the database. **View**contains all the jsp pages that are rendered as HTML output that the client's browser can interpret example viewTransaction.jsp shows all transaction objects. **Controller**Java Classes that provide behaviour by that handling UI inputs and ,user-requests. and passing data between model and view.
Creational Design Pattern	Singleton pattern	Implementation Of Unit of work for Vehicle, Client and Transaction Objects to ensure single instantiation.
Behavioural Design Pattern	Strategy pattern	VehicleUnitOfWork, ClientUnitOfWork and TransactionUnitOfWork implement a common UnitOfWork interface. The behavior of the object can change dynamically based on their specific implementation.