EXPERIMENT 1

AIM-Write down the problem statement for Parking Management System

PROBLEM STATEMENT

Parking Management System (PMS)

The rapid growth in vehicles has made urban parking management increasingly difficult. Traditional systems that rely on manual ticketing and cash payments often cause delays, human errors, long queues, incorrect billing, double booking and revenue loss due to fraud. They also lack real-time monitoring, transparency and flexible payment options leading to poor user experience and inefficient operations.

A digital Parking Management System (PMS) addresses these issues by automating and streamlining parking operations. It includes features like automatic number plate recognition, real-time tracking of parking space availability, secure digital payments, e-receipts and a user-friendly mobile app or web portal. For administrators, it offers a centralized dashboard, live monitoring and automated reporting. PMS reduces waiting times, eliminates manual errors, improves transparency, prevents fraud and supports future scalability. Overall, it provides a more efficient, accurate and user-friendly solution to modern parking challenges.

EXPERIMENT - 2

Software Requirements Specification

for

Parking Management System

Faculty Name :

Student Name:

Roll No:

Semester : 5th Semester

Table of Contents

Revision History

Name	Date	Reason For Changes	Version

1. Introduction

1.1 Purpose

The Parking Management System (PMS) is designed to modernize parking facility management across multiple locations by replacing manual processes with an automated system that records vehicle entries, allocates spots, issues tickets and processes payments efficiently. This SRS defines system requirements to ensure a clear understanding among developers, testers, administrators and stakeholders.

1.2 Document Conventions

This document follows the IEEE SRS template with sections explaining various system aspects. Functional requirements have unique codes (e.g., REQ-1). UML notation is used for models, and technical terms are clarified in the glossary. Descriptive paragraphs ensure clarity and avoid ambiguity.

1.3 Intended Audience and Reading Suggestions

This SRS targets project managers, clients, developers, testers, and administrators. Each group uses the document for different purposes such as understanding goals, implementing features, testing and deployment. Readers should start with the introduction then the overall description, followed by detailed requirements and constraints.

1.4 Product Scope

The PMS simplifies parking operations by managing multiple lots and parking spots. It records vehicle details, assigns spots automatically, generates tickets, and calculates fees at exit with multiple payment options. The system supports reporting and revenue tracking with objectives of efficiency, transparency, scalability, and user convenience.

1.5 References

References include IEEE Std 830-1998, Karl Wiegers' SRS template (1999), UML 2.5 specification, and relevant payment and data protection guidelines.

---

2. Overall Description

2.1 Product Perspective

PMS is an independent, extensible system with modules for lot management, vehicle registration, allocation, ticketing, and payment. It uses a centralized database and integrates with license plate recognition cameras, payment gatewaysW and notification systems. Modular design allows easy upgrades.

2.2 Product Functions

Administrators manage parking lots by specifying details like name, location, and capacity. The system records vehicle details at entry, allocates spots automatically, issues tickets, calculates fees on exit, and processes payments securely. It generates reports on occupancy, revenue, and usage trends.

2.3 User Classes and Characteristics

Users include administrators who manage the system and review reports, and customers who interact via mobile or web interfaces to access parking services. Each user class has distinct roles and permissions to ensure smooth operation.

2.4 Operating Environment

The PMS supports cloud and on-premises deployments, compatible with Android, iOS, modern browsers, and databases like PostgreSQL and MySQL. Communications are secured with HTTPS and TLS. Stable internet is preferred, with limited offline support.

2.5 Design and Implementation Constraints

The system complies with digital payment regulations and data privacy laws (e.g., GDPR). It depends on third-party APIs and enforces encryption and role-based access control to protect data.

2.6 User Documentation

Documentation includes detailed manuals for administrators and online help and FAQs for customers, ensuring usability for all users.

2.7 Assumptions and Dependencies

PMS assumes reliable internet connectivity and depends on third-party services for payments and vehicle recognition. API or regulation changes may require updates.

---

3. External Interface Requirements

3.1 User Interfaces

PMS offers user-friendly mobile apps and web portals for customers to view availability, register vehicles, and pay. Administrators access dashboards with real-time stats. Interfaces are designed for easy use.

3.2 Hardware Interfaces

Interfaces include license plate recognition cameras, ticket printers, and POS terminals for seamless integration of hardware components.

3.3 Software Interfaces

PMS interacts with databases, authentication services, payment gateways, and external APIs like municipal systems for enhanced functionality.

3.4 Communications Interfaces

All communication uses HTTPS with TLS encryption. RESTful APIs support secure third-party integration and real-time data synchronization.

---

4. System Features

4.1 Parking Lot Management

Admins can add, edit, or remove parking lots with details such as name, location, and capacity. The system maintains real-time records of spot availability to prevent conflicts and optimize resource use.

4.2 Vehicle Registration and Ticketing

Vehicle details are recorded manually or automatically at entry. The system issues unique tickets with entry time and assigned spot, streamlining tracking and reducing errors

.

4.3 Parking Spot Allocation

The system automatically assigns spots based on vehicle type and occupancy, updating availability instantly to maximize space utilization.

4.4 Payment Processing

On exit, charges are calculated based on stay duration and vehicle type. Multiple payment methods (UPI, cards, cash) are supported via secure gateways, ensuring accurate billing.

4.5 Reporting and Analytics

Admins generate reports on usage, revenue and occupancy for daily, weekly, or monthly periods, aiding in decision-making and resource planning.

---

5. Other Nonfunctional Requirements

5.1 Performance Requirements

The PMS must support up to 10,000 concurrent users with response times under two seconds, ensuring smooth operation during peak times.

5.2 Safety Requirements

The system guarantees reliable and safe operation to protect users and data.

5.3 Security Requirements

PMS enforces encryption, secure authentication, and complies with data privacy standards to protect user information.

5.4 Software Quality Attributes

The system is scalable, maintainable, and reliable to support large-scale operations efficiently.

5.5 Business Rules

The PMS adheres to financial and operational rules set by regulations and organizational policies.

---

6. Other Requirements

The PMS ensures high availability, efficient data handling, and compliance with legal and technical standards to support growing user demands.

Appendix A: Glossary

- PMS: Parking Management System
- Parking Lot: Location with parking spots
- Ticket: Proof of parking issued at entry

- UPI: Unified Payments Interface
- TLS: Transport Layer Security
- PCI-DSS: Payment Card Industry Data Security Standard

Appendix B: Analysis Models

Includes ER diagram, Data Flow Diagrams (Level 0 and 1), and UML Use Case Diagram.

Appendix C: To Be Determined List

Pending clarifications:

1. Final tariff structure
2. Offline support scope
3. License plate recognition hardware
4. Refund policies for failed payments

EXPERIMENT - 3

AIM - To perform the function oriented diagram: Data Flow Diagram(DFD) andStructured chart.

THEORY

DFD (Data Flow Diagram) – Level 0

Definition:

- Level 0 DFD is also called the **Context Diagram**.
- It shows the **entire system as a single process** with its **external entities** and **main data inflows/outflows**.
  No internal details or sub-processes are shown.

Purpose:

- To give a **high-level overview** of the system.
- Helps understand **who interacts** with the system and **what data** they exchange.

Key Points:

- Single process (Process 0).
- External entities (users, devices, other systems).
- Data flows between entities and the main process.
- No data stores shown usually.

DFD Level 1

Definition:

- Level 1 DFD is a **decomposition** of Level 0.
- The single main process is broken into **multiple sub-processes** (e.g., 1.0, 2.0, 3.0…).
- Shows **data stores**, **detailed data flow**, and **process interactions**.

Purpose:

- To describe **internal functions of the system** in detail.
- Shows how data moves between processes and data stores.

Key Points:

- Breaks Process 0 into smaller processes.
- Shows data stores (D1, D2…).
- More detailed but still avoids technical implementation details.
- Shows internal logical flow clearly.

Structured Chart

Definition:

- A Structured Chart is a **hierarchical diagram** showing the **modular structure** of a system.
- It represents how the system is **divided into modules**, their **calling relationships**, and **data communication**.

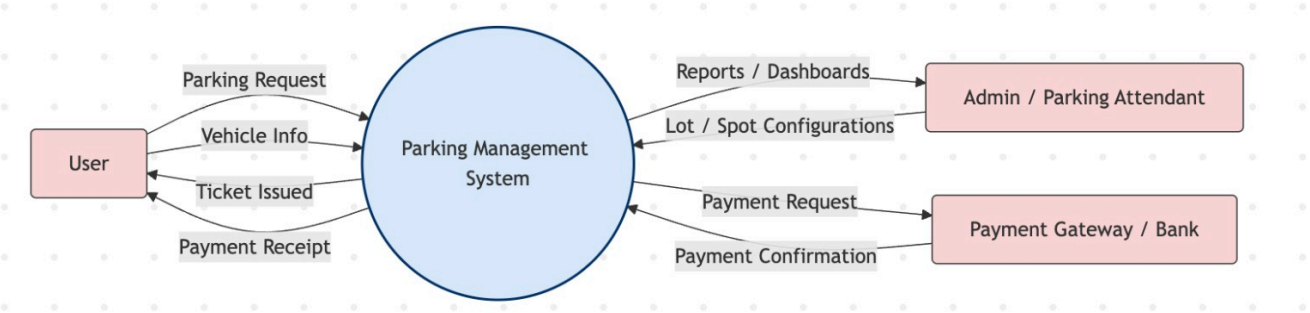
Purpose:

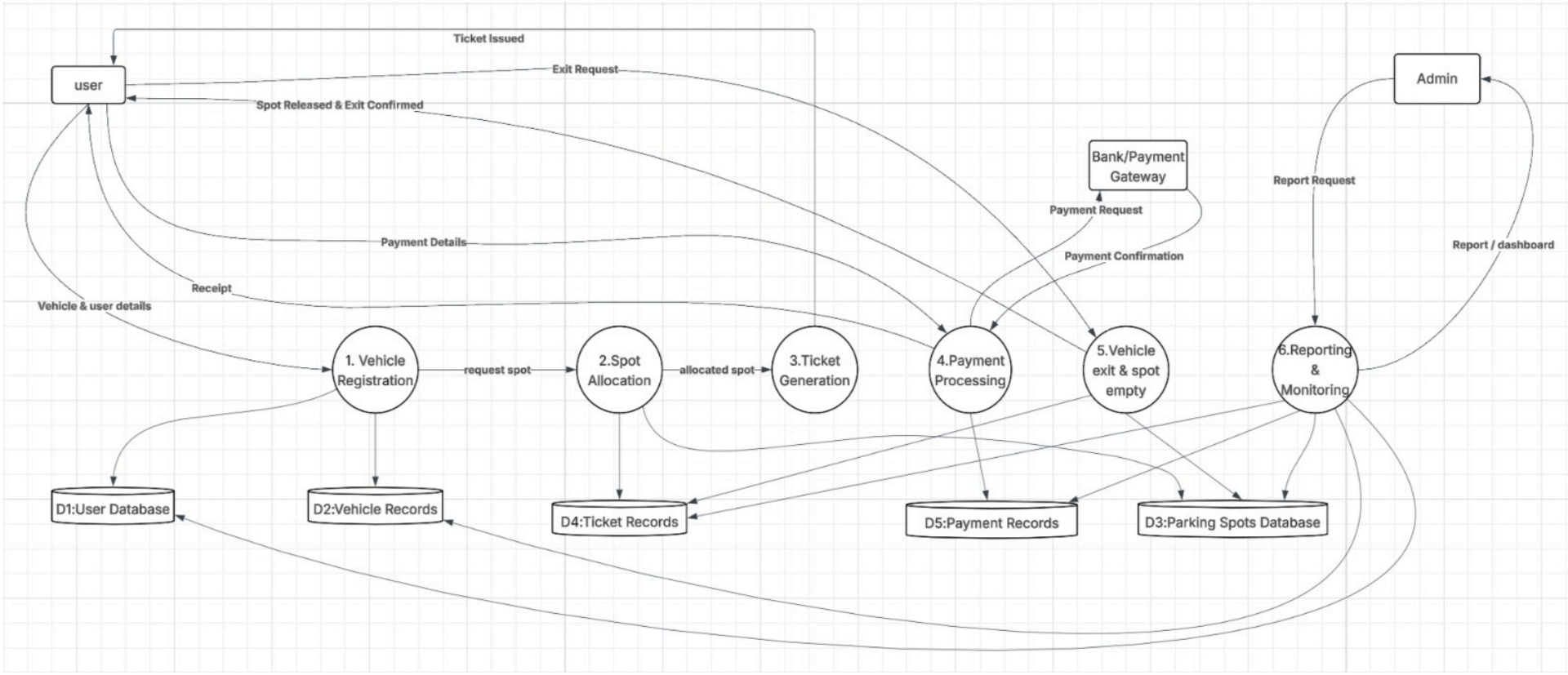
- To show the **overall program design** in a top-down manner.
- Used in **structured programming** to break the system into smaller, independent modules.

Key Characteristics:

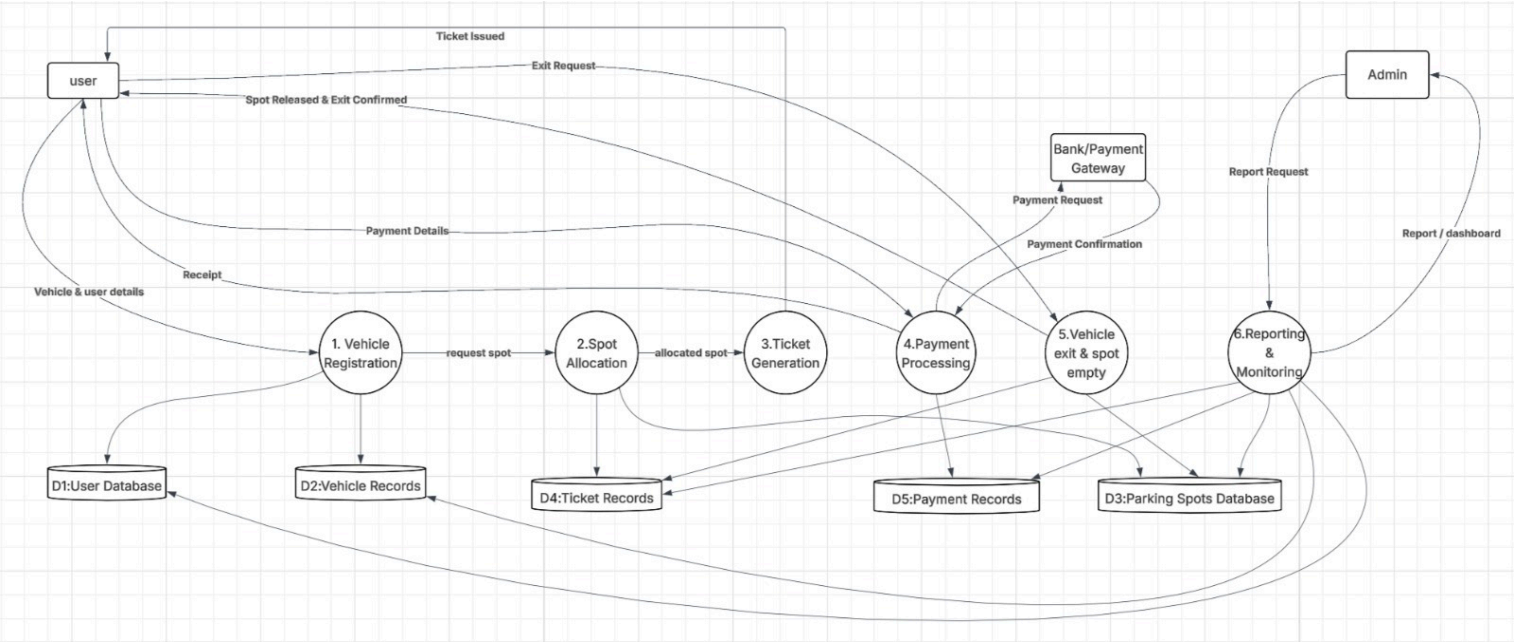
- Top module represents the **main program**.
- Lower modules show **subtasks or functions**.
- Uses arrows to show **control/data communication**.
- Helps understand **module hierarchy**, **coupling**, and **cohesion**.

LEVEL 0 Data Flow Diagram

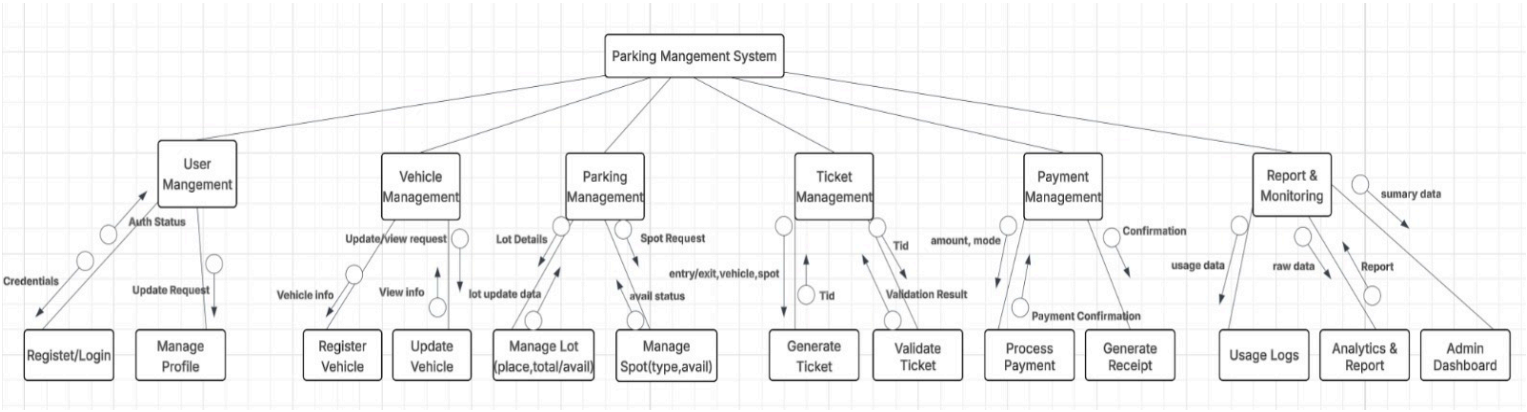



Ticket Issued
user
Exit Request
Spot Released & Exit Confirmed
Admin
Bank/Payment Gateway
Report Request
Payment Request
Payment Details
Payment Confirmation
Report / dashboard
Receipt
Vehicle & user details
1. Vehicle Registration
request spot
2.Spot Allocation
allocated spot
3.Ticket Generation
4.Payment Processing
5.Vehicle exit & spot empty
6.Reporting & Monitoring
D1:User Database
D2:Vehicle Records
D4:Ticket Records
D5:Payment Records
D3:Parking Spots Database

LEVEL 1 DFD


Ticket Issued
user
Exit Request
Spot Released & Exit Confirmed
Admin
Bank/Payment Gateway
Report Request
Payment Request
Payment Details
Payment Confirmation
Report / dashboard
Receipt
Vehicle & user details
1. Vehicle Registration
request spot
2.Spot Allocation
allocated spot
3.Ticket Generation
4.Payment Processing
5.Vehicle exit & spot empty
6.Reporting & Monitoring
D1:User Database
D2:Vehicle Records
D4:Ticket Records
D5:Payment Records
D3:Parking Spots Database

Structured Chart



EXPERIMENT - 4

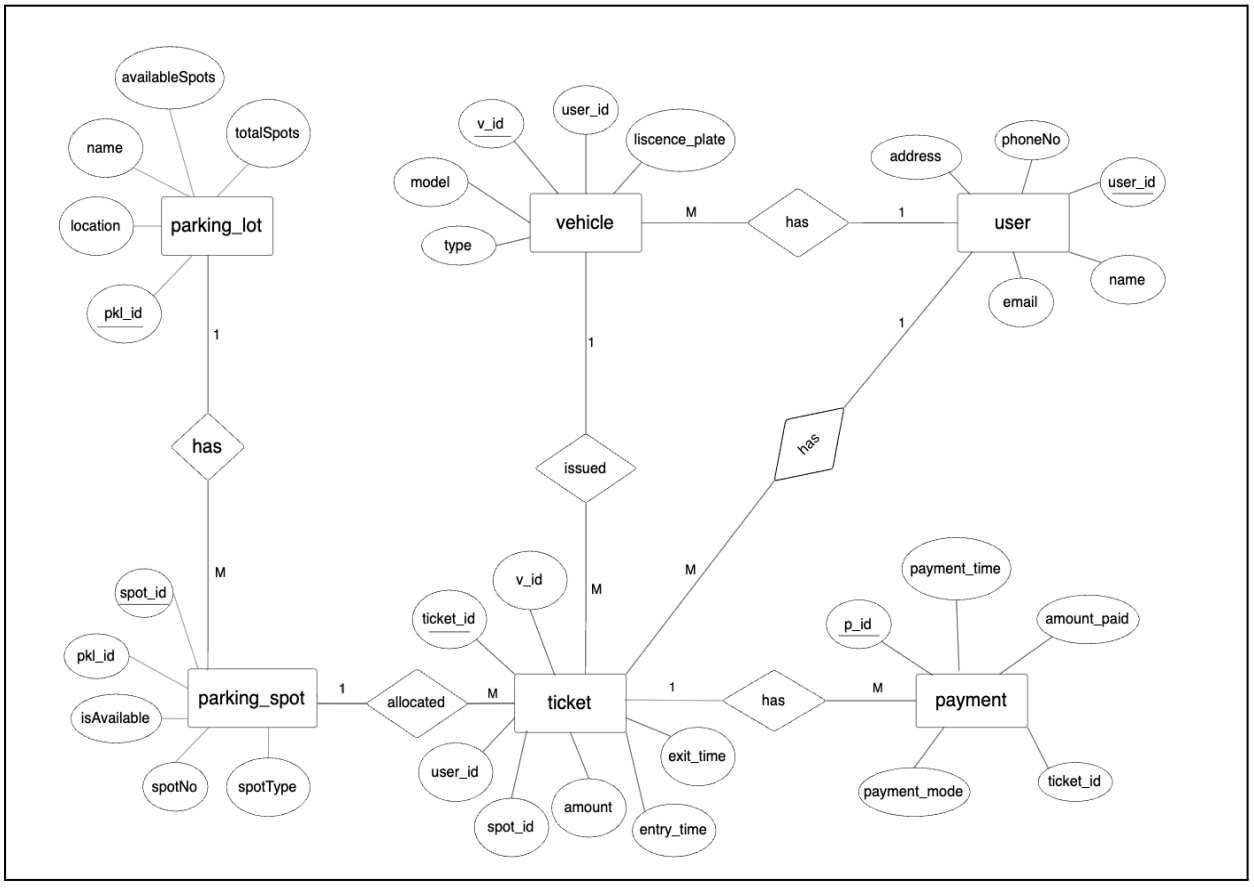
AIM - Draw the entity relationship diagram for the suggested system (Parking Management System).

THEORY

An ER (Entity–Relationship) diagram is a simple visual way to show how data is organized in a database. It uses **entities** (like Student, Course, Employee), **attributes** (details of each entity, like name, age, ID) and **relationships** (how two entities are connected, like Student *enrolls in* Course). ER diagrams help you understand the structure of a database before actually creating it. They make it easier to see what data is needed, how different parts are linked and how information will flow.

In an ER diagram, entities are drawn as rectangles, attributes as ovals and relationships as diamonds. It also shows important concepts like **primary keys**, **cardinality** (one-to-one, one-to-many, many-to-many) and constraints. This clear representation helps in proper database design, reduces redundancy and ensures data consistency.

Entity Relationship Diagram



EXPERIMENT - 5

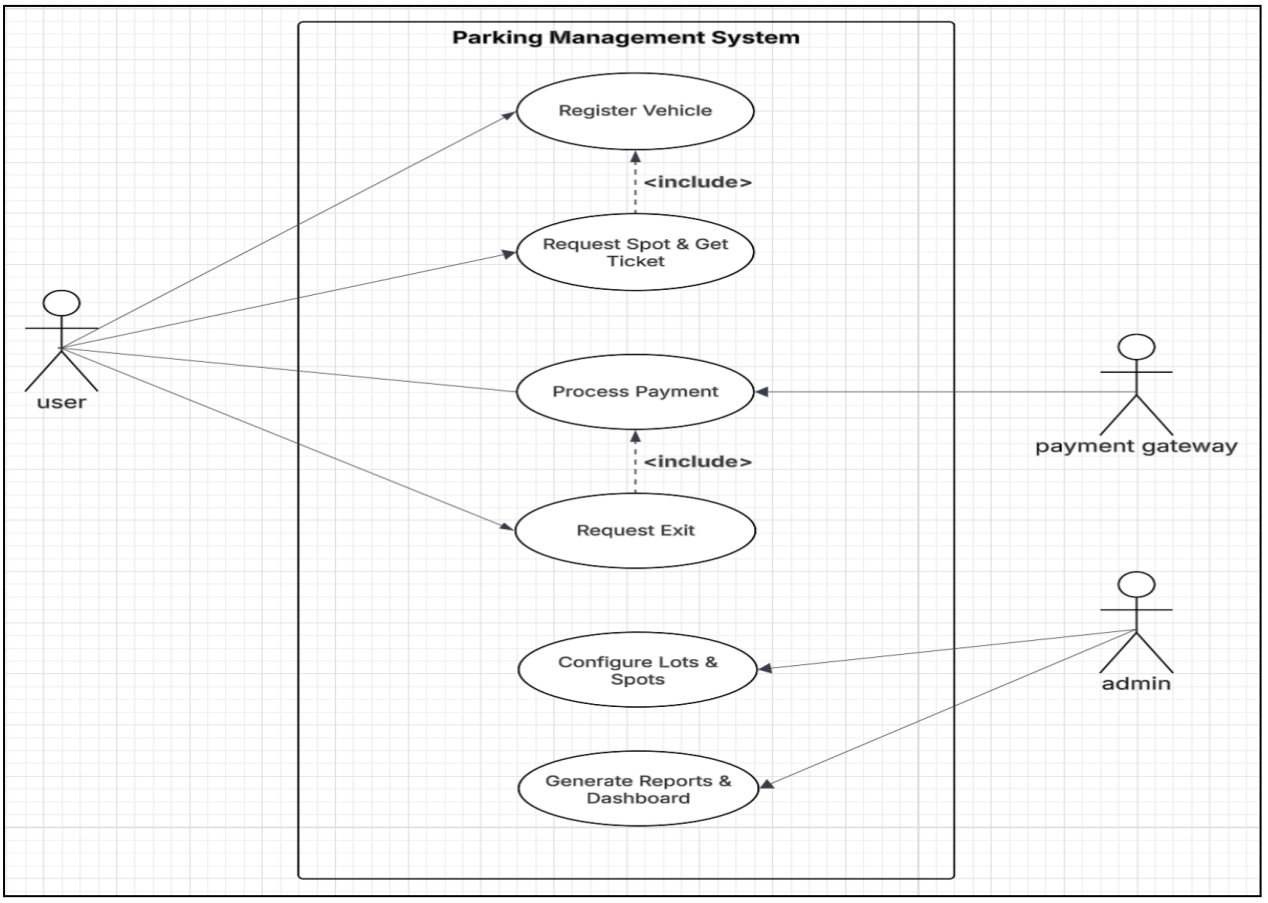
AIM - To perform the user's view analysis for the suggested system (Parking Management System) i.e Use case diagram.

THEORY

A **use case diagram** is a simple visual tool in software engineering that shows how different users interact with a system. It focuses on **what the system should do**, not how it works internally. The diagram contains **actors** (people or external systems using the application) and **use cases** (the actions or functions the system provides, like "Login," "Register," "Place Order"). Lines connect actors to the use cases they can perform, making the overall workflow easy to understand.

Use case diagrams help in gathering requirements during the early stages of a project. They give a clear picture of the system's functionality from the user's viewpoint. This makes communication easy between developers, stakeholders and clients, ensuring everyone understands what the system must deliver.

Use case diagram



EXPERIMENT - 6

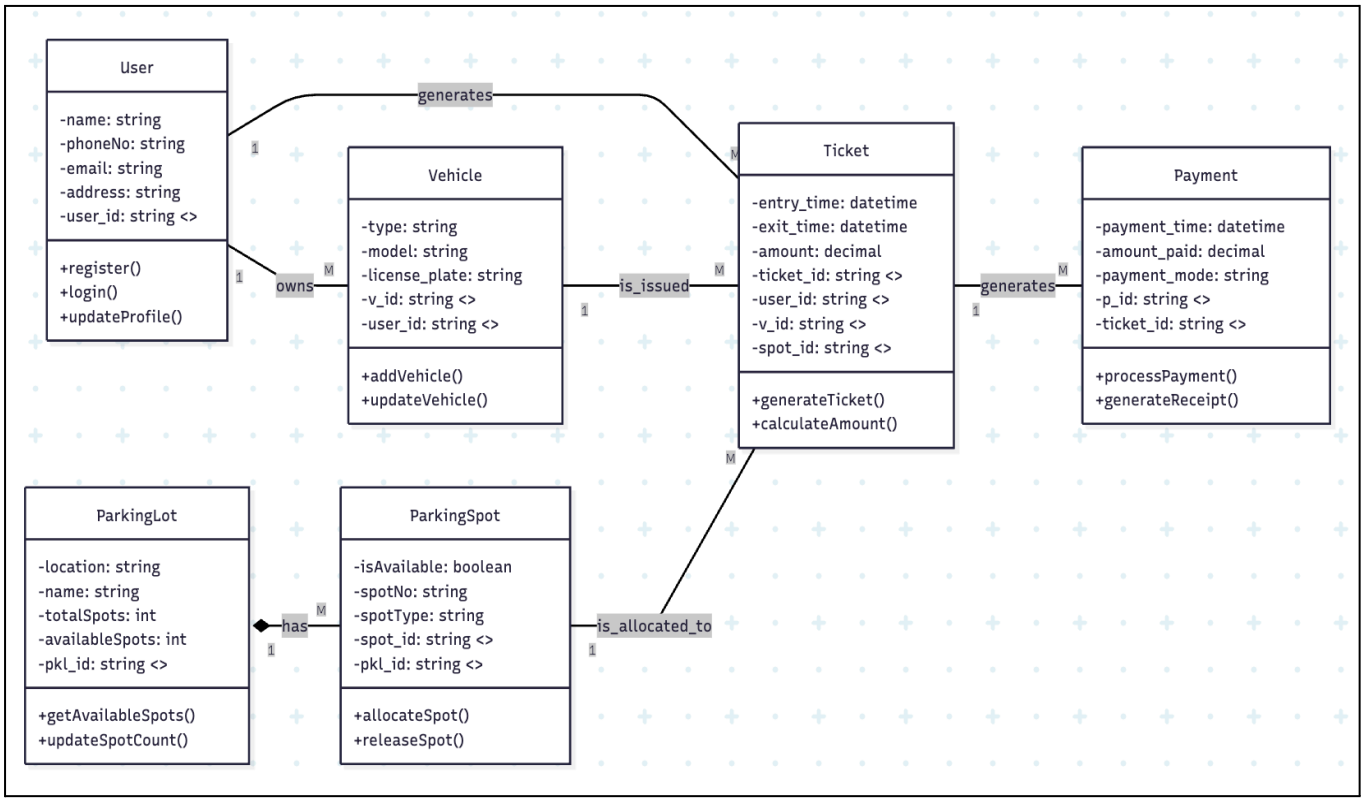
AIM - To draw the structural view diagram for the system: Class diagram, Object diagram.

THEORY

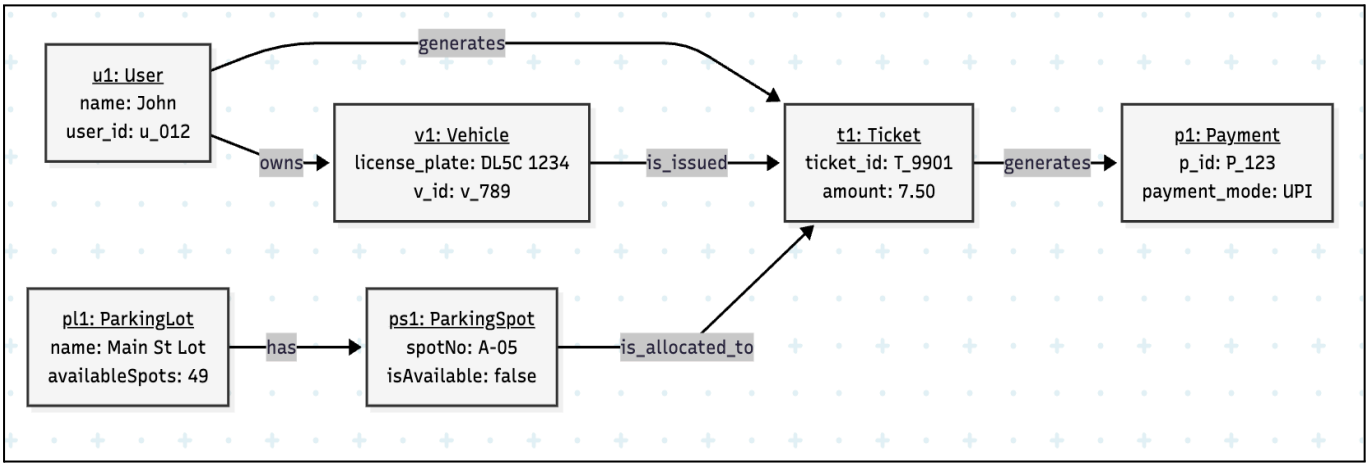
A **class diagram** is a core part of object-oriented design. It shows the **classes** in a system, their **attributes**, **methods** and the **relationships** between them. Each class is drawn as a box with three sections: the class name, its data members (attributes) and its functions (methods). Class diagrams also show connections like inheritance, association, aggregation and composition. They help developers understand the structure of the software and how different parts interact.

An **object diagram** is like a snapshot of a class diagram at a particular moment. Instead of showing classes, it shows **objects** (real examples of classes) with specific values assigned to their attributes. It helps visualize how objects actually look and relate at runtime. Object diagrams are useful for understanding concrete examples, testing scenarios and explaining complex relationships in a simple way.

Class diagram



Object diagram



EXPERIMENT - 7

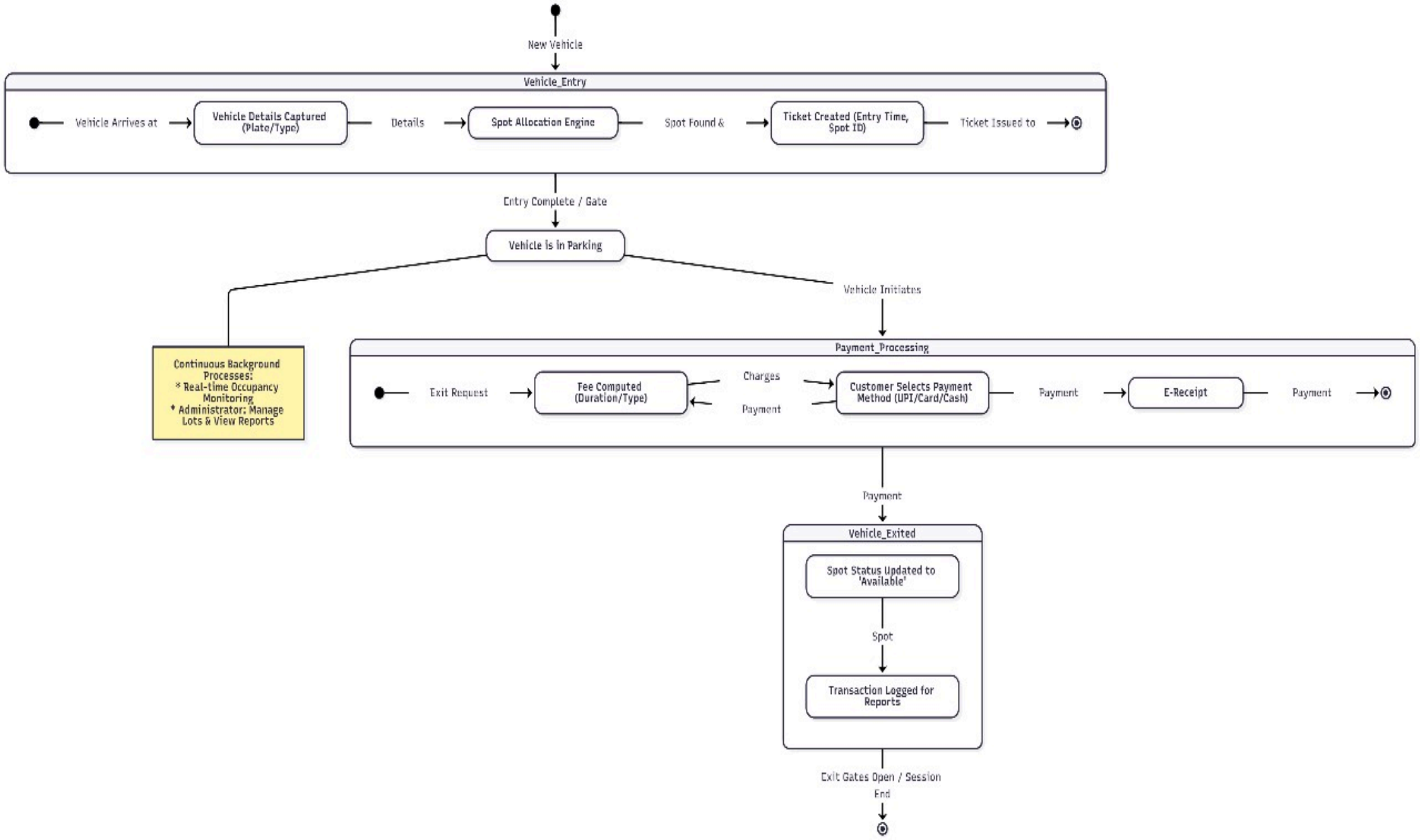
AIM - To draw the behavioural view diagram: State-chart diagram, Activity diagram.

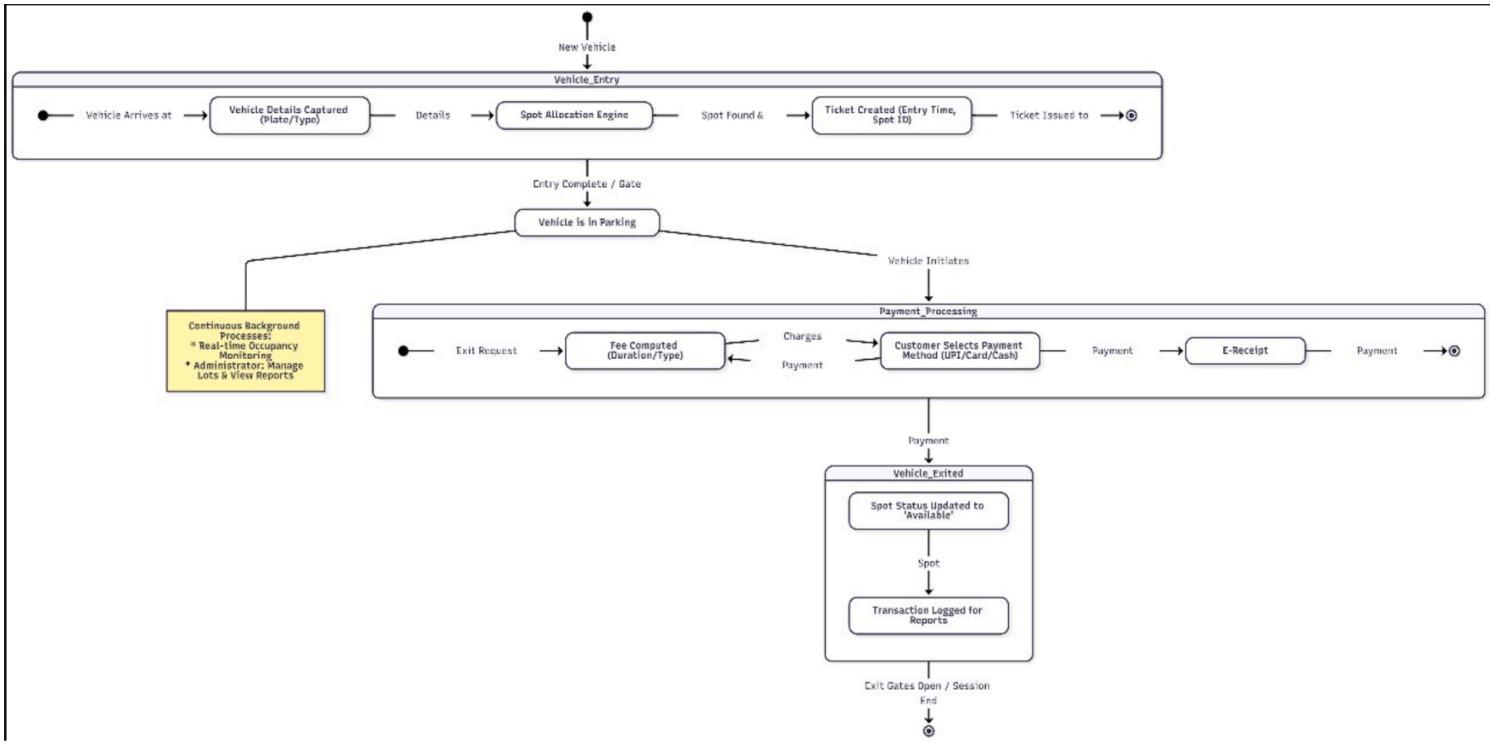
THEORY

State-Chart Diagram:
A state-chart diagram shows how an object behaves based on different **states** and **events**. It focuses on the life cycle of a single object—how it moves from one state to another when something happens. Each state represents a condition (like *Idle*, *Processing*, *Completed*) and arrows show transitions triggered by events. This diagram is useful when modeling systems where the behavior changes over time, such as ATM transactions or order processing.

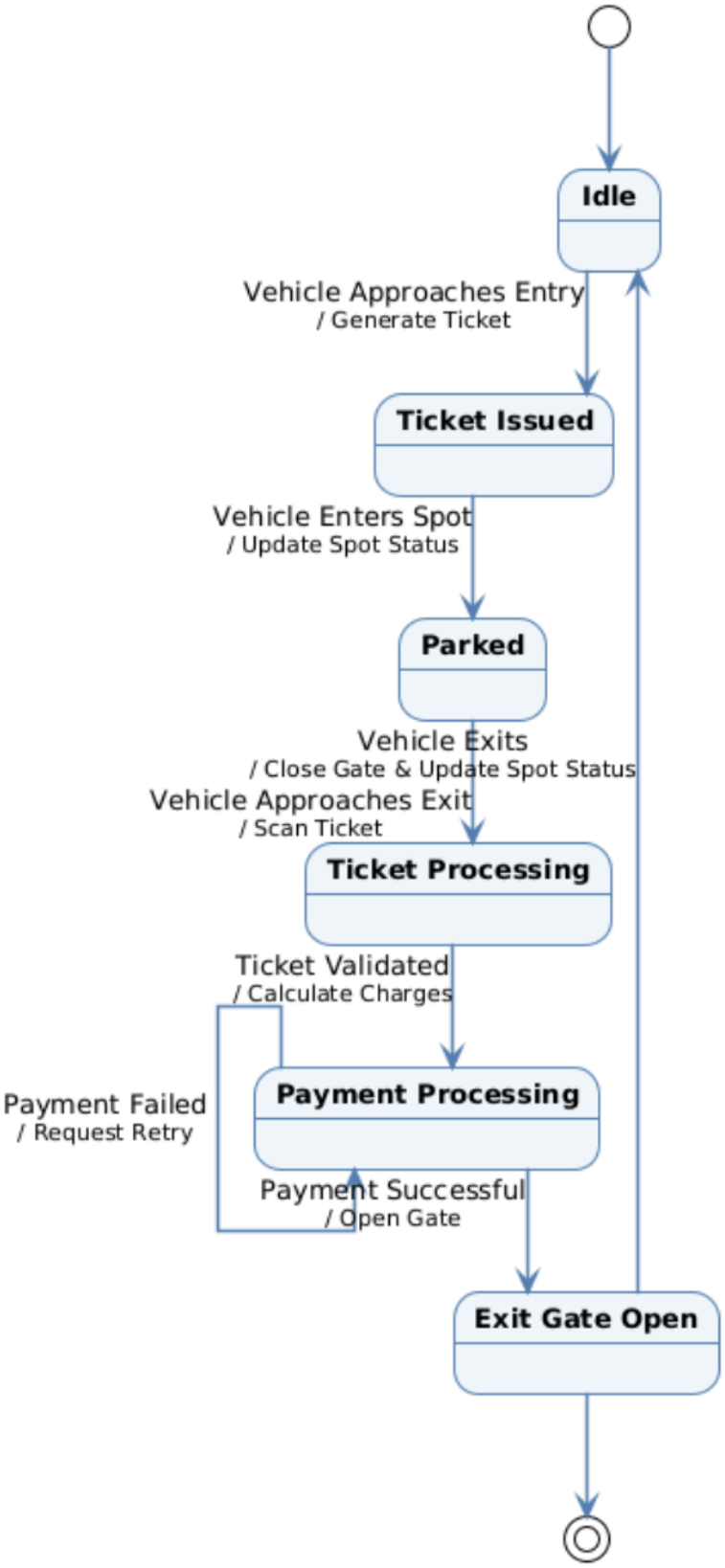
Activity Diagram:
An activity diagram represents the **flow of actions or activities** in a process. It looks similar to a flowchart and shows how tasks start, branch, merge and end. Activities are written inside rounded rectangles and arrows show the flow. It is used to model workflows, business processes, or complex operations like login flow, payment processing, or form submission. Activity diagrams help understand how tasks progress step by step in a system.

State-chart diagram

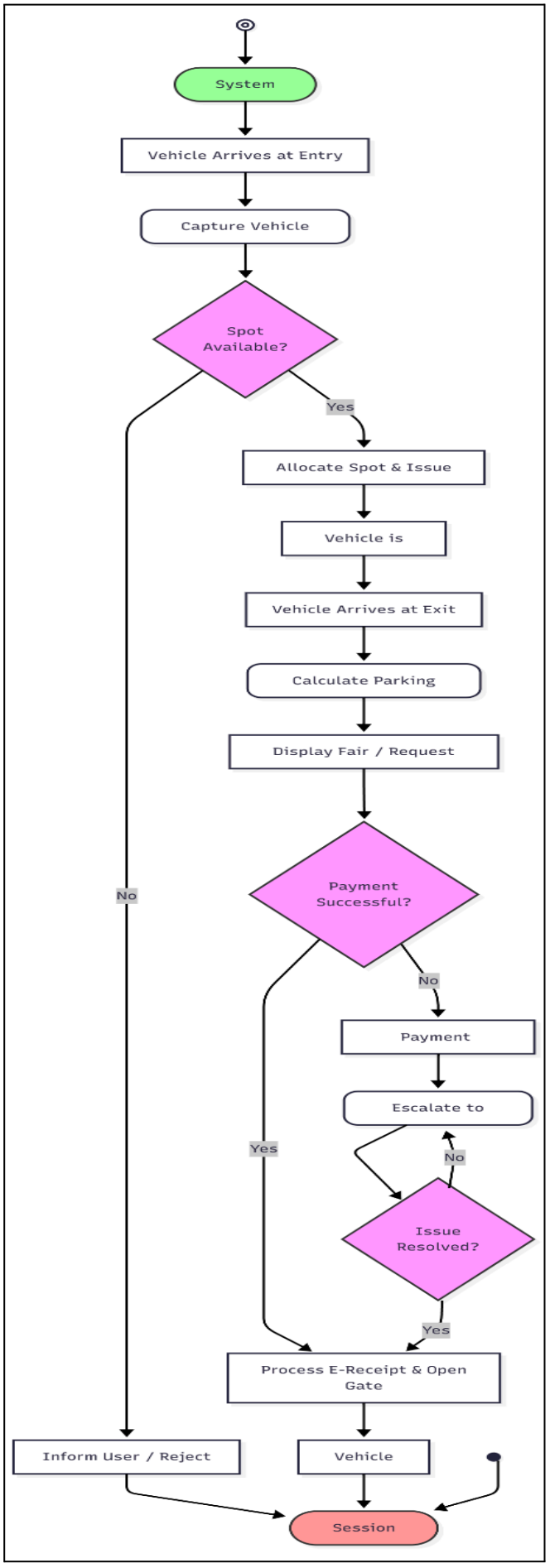

New Vehicle
Vehicle_Entry
Vehicle Arrives at
Vehicle Details Captured (Plate/Type)
Details
Spot Allocation Engine
Spot Found &
Ticket Created (Entry Time, Spot ID)
Ticket Issued to
Entry Complete / Gate
Vehicle is in Parking
Vehicle Initiates
Continuous Background Processes:
* Real-time Occupancy Monitoring
* Administrator: Manage Lots & View Reports
Payment_Processing
Exit Request
Fee Computed (Duration/Type)
Charges
Payment
Customer Selects Payment Method (UPI/Card/Cash)
Payment
E-Receipt
Payment
Payment
Vehicle_Exited
Spot Status Updated to 'Available'
Spot
Transaction Logged for Reports
Exit Gates Open / Session End


New Vehicle
Vehicle_Entry
Vehicle Arrives at
Vehicle Details Captured (Plate/Type)
Details
Spot Allocation Engine
Spot Found &
Ticket Created (Entry Time, Spot ID)
Ticket Issued to
Entry Complete / Gate
Vehicle is in Parking
Vehicle Initiates
Continuous Background Processes:
* Real-time Occupancy Monitoring
* Administrator: Manage Lots & View Reports
Payment_Processing
Exit Request
Fee Computed (Duration/Type)
Charges
Payment
Customer Selects Payment Method (UPI/Card/Cash)
Payment
E-Receipt
Payment
Payment
Vehicle_Exited
Spot Status Updated to 'Available'
Spot
Transaction Logged for Reports
Exit Gates Open / Session End

Parking Management System State Chart



Activity diagram



EXPERIMENT – 8

AIM –

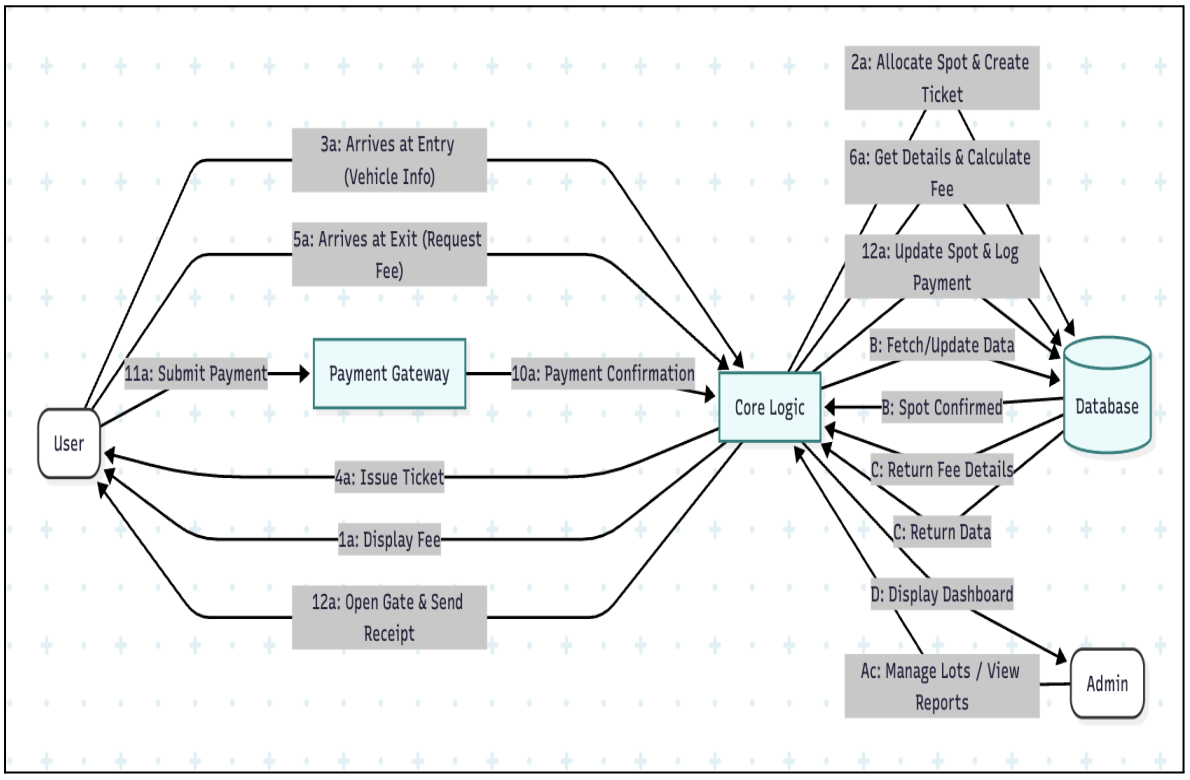
To perform the behavioural view diagram for the suggested system (Parking Management System) : Sequence diagram, Collaboration diagram.

THEORY

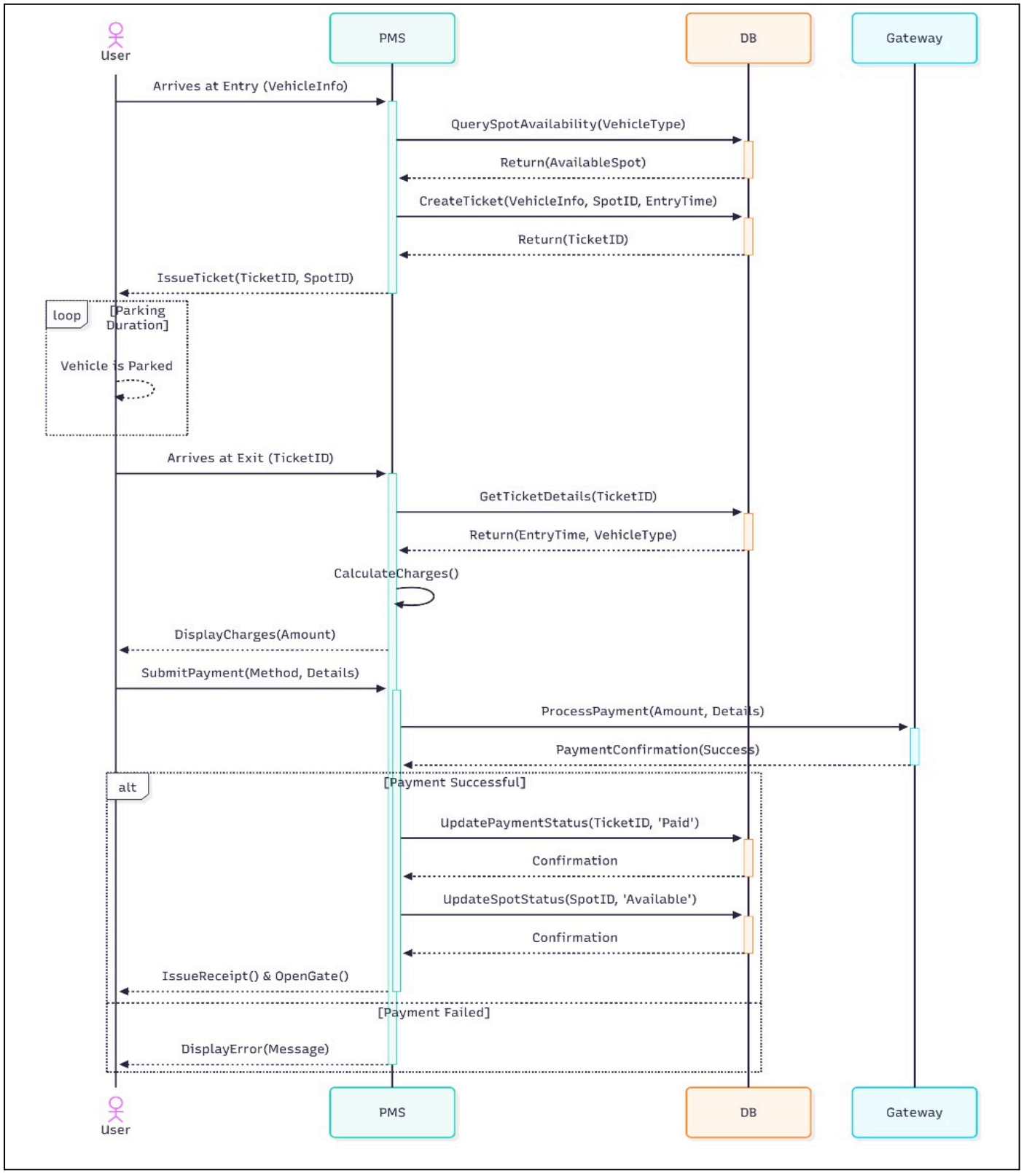
Sequence Diagram:
A sequence diagram shows how objects interact with each other **over time**. It focuses on the order of messages passed between objects. Each object is shown with a lifeline and arrows represent messages or method calls. The diagram reads from top to bottom, making it easy to see which action happens first and what follows next. It is useful for understanding the exact flow of a feature, like how a login request moves through different components.

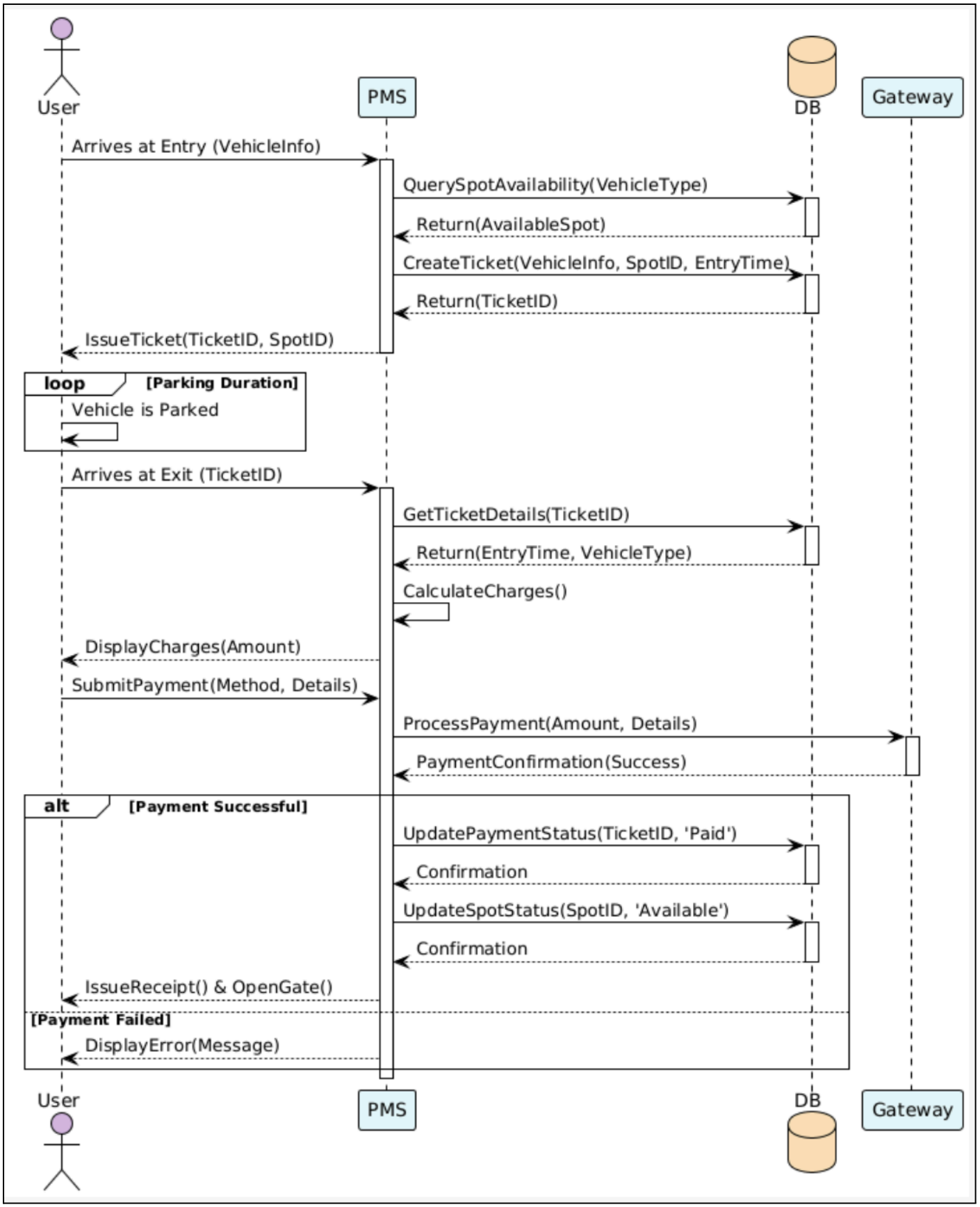
Collaboration Diagram:
A collaboration diagram (also called a communication diagram) shows how objects interact, but it focuses more on the **relationships between objects** rather than the time order. Objects are shown as nodes and messages between them are labeled with sequence numbers to indicate order. This diagram helps visualize how different parts of a system work together and how responsibilities are shared among objects.

Collaboration diagram



Sequence diagram




User
PMS
DB
Gateway
Arrives at Entry (VehicleInfo)
QuerySpotAvailability(VehicleType)
Return(AvailableSpot)
CreateTicket(VehicleInfo, SpotID, EntryTime)
Return(TicketID)
IssueTicket(TicketID, SpotID)
loop
[Parking Duration]
Vehicle is Parked
Arrives at Exit (TicketID)
GetTicketDetails(TicketID)
Return(EntryTime, VehicleType)
CalculateCharges()
DisplayCharges(Amount)
SubmitPayment(Method, Details)
ProcessPayment(Amount, Details)
PaymentConfirmation(Success)
alt
[Payment Successful]
UpdatePaymentStatus(TicketID, 'Paid')
Confirmation
UpdateSpotStatus(SpotID, 'Available')
Confirmation
IssueReceipt() & OpenGate()
[Payment Failed]
DisplayError(Message)
User
PMS
DB
Gateway

EXPERIMENT - 9

AIM

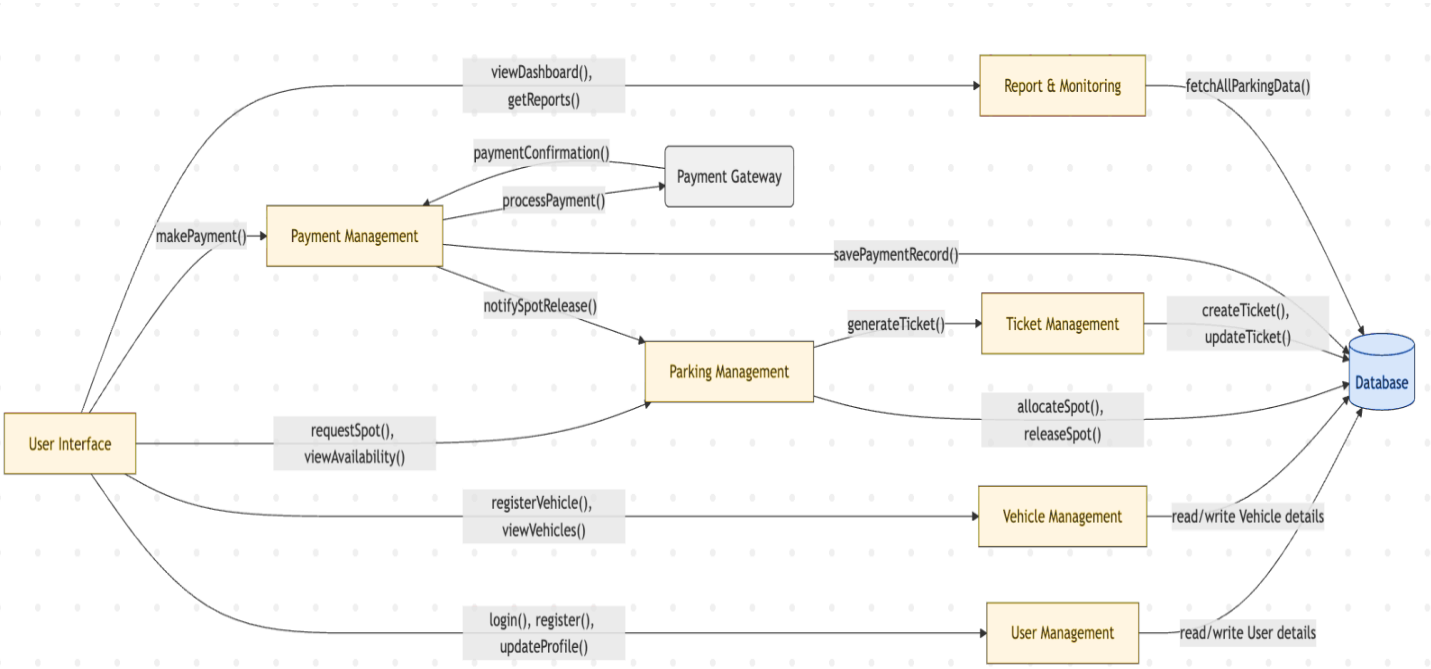
To perform the implementation view diagram: Component diagram for the system.

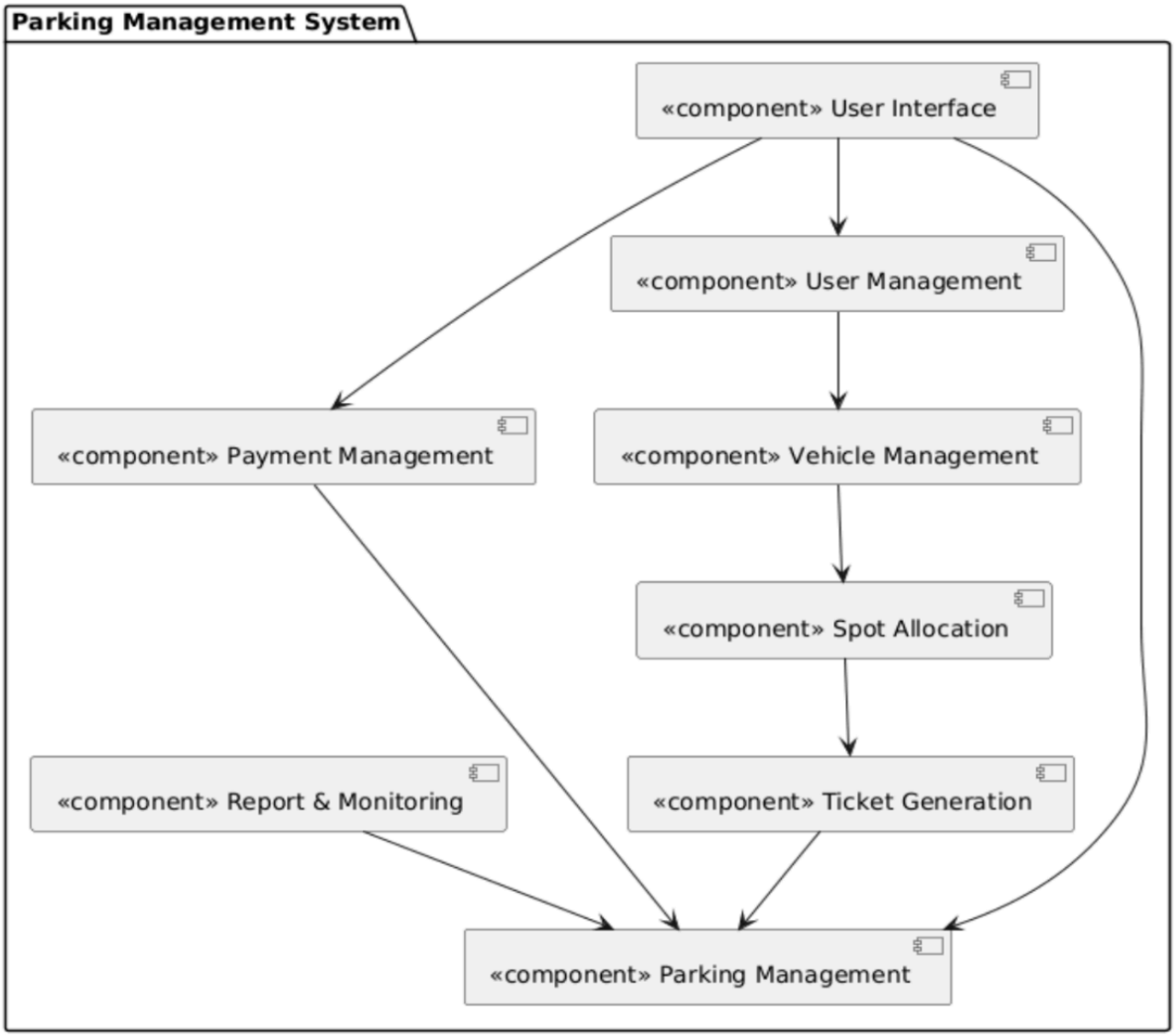
THEORY

A **component diagram** shows how the software system is divided into separate, reusable parts called **components**. Each component represents a module, library, or subsystem that performs a specific function, such as a login module, database module, or payment service. The diagram highlights how these components connect, depend on each other and work together to form the complete system.

Component diagrams are useful in the design and architecture phase because they help visualize the system's structure at a higher level. They make it clear which parts can be developed independently, reused, or replaced. This helps in maintaining, scaling and organizing large software projects more efficiently.

Component diagram




Parking Management System
«component» User Interface
«component» User Management
«component» Payment Management
«component» Vehicle Management
«component» Spot Allocation
«component» Report & Monitoring
«component» Ticket Generation
«component» Parking Management

EXPERIMENT - 10

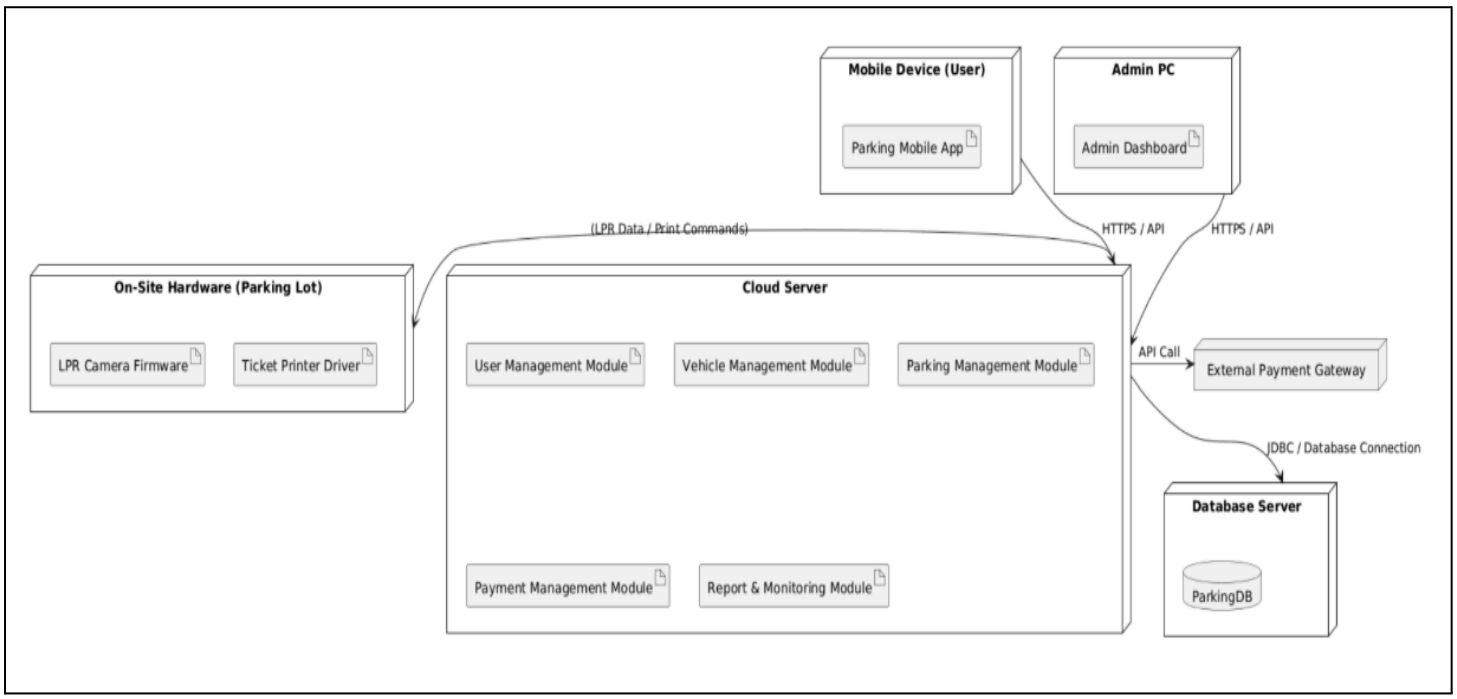
AIM - To perform the environmental view diagram: Deployment diagram for the system.

THEORY

A **deployment diagram** shows how a software system is physically placed on hardware. It focuses on the **runtime environment**, meaning where each part of the system actually runs. The diagram includes **nodes** like servers, computers, mobile devices, or cloud systems and shows which software components are deployed on each node. Connections between nodes represent communication paths, such as network links.

Deployment diagrams are useful for understanding the system's **infrastructure**, performance needs and installation setup. They help plan how the application will run in the real world—such as which server hosts the database, where the application server runs and how different machines interact in a distributed environment.

Deployment diagram



EXPERIMENT - 11

AIM

To perform various testing using unit testing, integration testing and prepare test cases for the sample code of the Parking Management System (PMS).

THEORY

Software testing ensures that the system meets functional and performance requirements. For the PMS, testing is essential because it handles real-time spot allocation, vehicle tracking and financial transactions—all of which require high accuracy and reliability.

1. Unit Testing Unit testing focuses on testing individual modules or functions of the software system. Examples of units in PMS:

- User.login()
- Vehicle.addVehicle()
- ParkingSpot.allocateSpot()
- Ticket.generateTicket()
- Payment.processPayment()

Unit testing ensures that each method works correctly in isolation. Frameworks like JUnit, PyTest, or Mocha can be used depending on technology.

2. Integration Testing Integration Testing checks the interaction between combined modules. In PMS, integration points include:

- User → Vehicle Registration Module
- Vehicle Registration → Spot Allocation Module
- Spot Allocation → Ticket Generation Module
- Payment Processing → Payment Gateway
- Payment Confirmation → Vehicle Exit & Spot Release

Integration testing ensures smooth data flow between modules.

TEST CASES FOR PMS

(A) Unit Test Cases

Test Case ID	Module	Test Description	Input	Expected Output
UT01	Login Module	Verify valid login	username=dev, password=123	Login successful
UT02	Vehicle Module	Register a new vehicle	license_plate="DL5C1234"	Vehicle registered, v_id created
UT03	Spot Module	Allocate an available spot	spotType="Car"	spot_id="A-05" allocated, isAvailable=false
UT04	Ticket Module	Generate new ticket	v_id="V-101", spot_id="A-05"	Ticket T-1001 created with entry time

(B) Integration Test Cases

Test Case ID	Modules	Test Description	Input	Expected Output
IT01	Registration → Spot Allocation	Vehicle registration triggers spot search	Register vehicle "DL5C1234"	Spot allocation process triggered to find a spot

IT02	Spot Allocation → Ticket	Successful allocation generates a ticket	Spot 'A-05' allocated	Ticket 'T-1001' generated and issued to user
IT03	Payment → Payment Gateway	User pays for the ticket	Pay 50 for Ticket 'T-1001'	Payment Request sent to Gateway, Payment Confirmation received
IT04	Payment → Spot Release	Successful payment frees the spot	Payment Confirmation for T-1001	ParkingSpot 'A-05' is updated to isAvailable=true

CONCLUSION:

Thus, unit testing and integration testing were performed for the PMS, ensuring correct functionality of all modules.

EXPERIMENT - 12

AIM

To perform estimation of effort using Function Point (FP) Estimation for the Parking Management System (PMS).

THEORY

FP Estimation is used to estimate the size and effort required to develop a software system. It is based on the number of external inputs, outputs, files and user interactions.

1. Identifying Function Types for PMS

Function Type	Count	Description
External Inputs (EI)	5	Login, Register Vehicle, Submit Payment Details, Admin Configures Lot, Admin Requests Report
External Outputs (EO)	4	Issue Ticket, Show Receipt, Display Spot Availability, Generate Admin Report
External Inquiries (EQ)	3	View Parking History, View User Profile, Check Spot Status
Internal Logical Files (ILF)	6	User, Vehicle, ParkingLot, ParkingSpot, Ticket, Payment tables
External Interface Files (EIF)	2	Payment Gateway API, License Plate Recognition (LPR) System

---

2. Assign Complexity Weights

Using standard FP weights (assuming medium complexity for all):

Function Type	Count	Weight	Total
EI	5	4	20
EO	4	5	20
EQ	3	4	12

ILF	6	7	42
EIF	2	5	10
Unadjusted Function Point (UFP)			**104**

UFP = 20 + 20 + 12 + 42 + 10 = **104 FP**

---

3. Value Adjustment Factor (VAF)

Based on 14 general system characteristics.

Assume total score = 30

VAF = 0.65 + (0.01 x 30) = **0.95**

---

4. Final FP Calculation

FP = UFP x VAF

FP = 104 x 0.95 = **98.8 ≈ 99 FP**

RESULT:

Function Point estimation for PMS is 99 FP, which helps in effort and cost estimation of the project.

EXPERIMENT - 13

AIM - To prepare a time-line chart/Gantt Chart/PERT Chart for the Parking Management System (PMS).

THEORY

Project scheduling helps visualize tasks, duration, dependencies and the critical path.

Gantt Chart for PMS

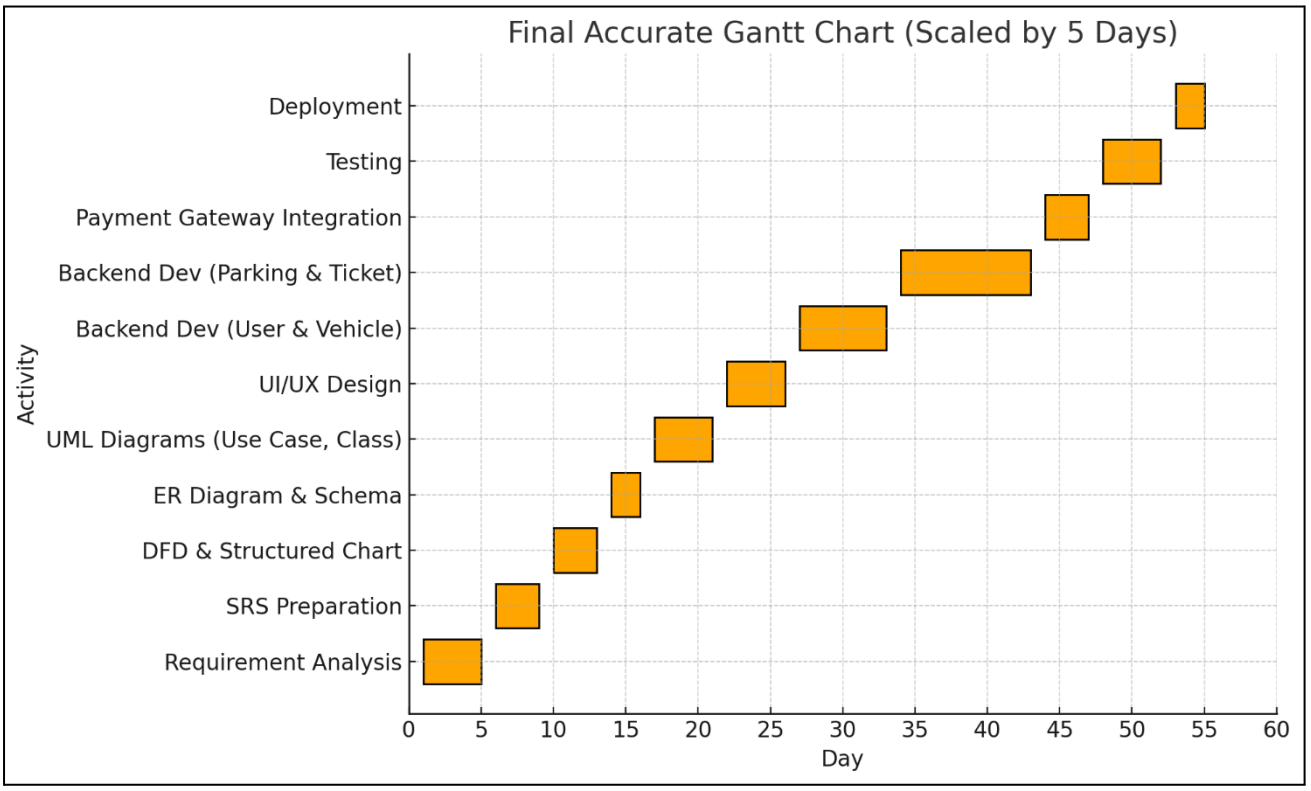
Activity	Duration	Start	End
Requirement Analysis	5 days	Day 1	Day 5
SRS Preparation	4 days	Day 6	Day 9
DFD & Structured Chart	4 days	Day 10	Day 13
ER Diagram & Schema	3 days	Day 14	Day 16
UML Diagrams (Use Case, Class)	5 days	Day 17	Day 21
UI/UX Design	5 days	Day 22	Day 26
Backend Dev (User & Vehicle)	7 days	Day 27	Day 33
Backend Dev (Parking & Ticket)	10 days	Day 34	Day 43
Payment Gateway Integration	4 days	Day 44	Day 47
Testing	5 days	Day 48	Day 52
Deployment	3 days	Day 53	Day 55

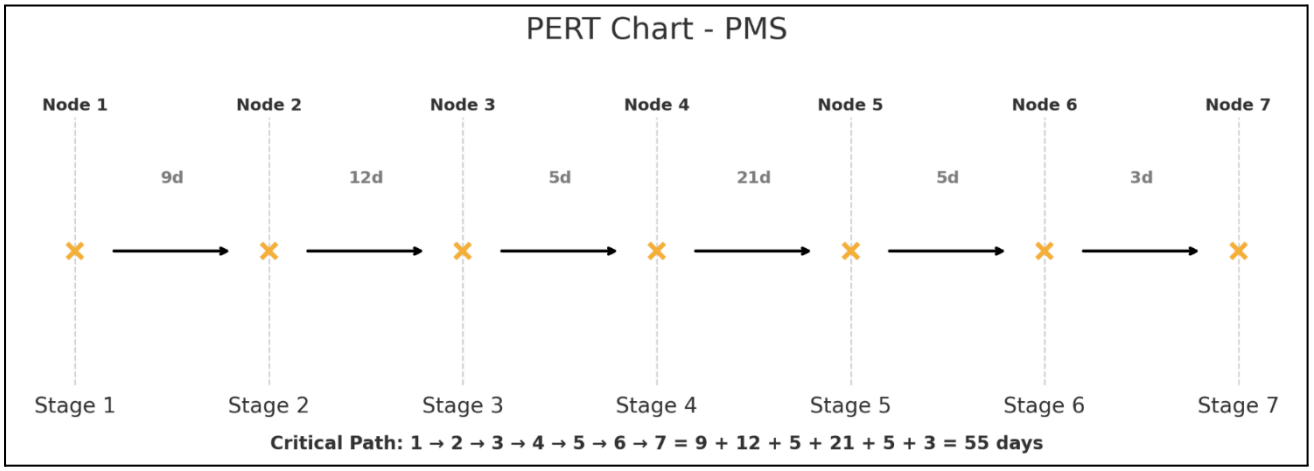
PERT Chart (Major Stages)

1. Stage 1 → 2: Analysis & SRS (9 days)
2. Stage 2 → 3: System Design (DFD, ER, UML) (12 days)
3. Stage 3 → 4: UI/UX Design (5 days)
4. Stage 4 → 5: Backend Development & Integration (21 days)
5. Stage 5 → 6: Testing (5 days)
6. Stage 6 → 7: Deployment (3 days)

Critical Path:

1 → 2 → 3 → 4 → 5 → 6 → 7 = 9 + 12 + 5 +21 + 5 + 3 = **55 days**





CONCLUSION:

A Gantt Chart and PERT Chart were prepared for the PMS project, estimating a total project duration of 55 days and mapping out task dependencies.

EXPERIMENT - 14

AIM - To give the comparison between the starUML and the Rational rose.

1. Introduction to UML Modeling Tools

UML (Unified Modeling Language) is a standard for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system. UML modeling tools help software engineers create and manage UML diagrams, thereby aiding in the design and analysis phases of software development.

2. StarUML

StarUML is an open-source UML modeling tool. It is designed to be a comprehensive solution for creating various UML diagrams and supports a wide range of modeling functionalities.

Key Features of StarUML:

- **Open Source:** Free to use and modify.
- **Cross-Platform:** Available on Windows, macOS and Linux.
- **Extensible:** Supports a plugin architecture allowing users to extend its functionality.
- **Standard UML Support:** Supports all standard UML 2.x diagrams.
- **Code Generation:** Can generate code from UML diagrams for various programming languages (e.g., Java, C#).
- **Reverse Engineering:** Can reverse engineer code into UML diagrams.

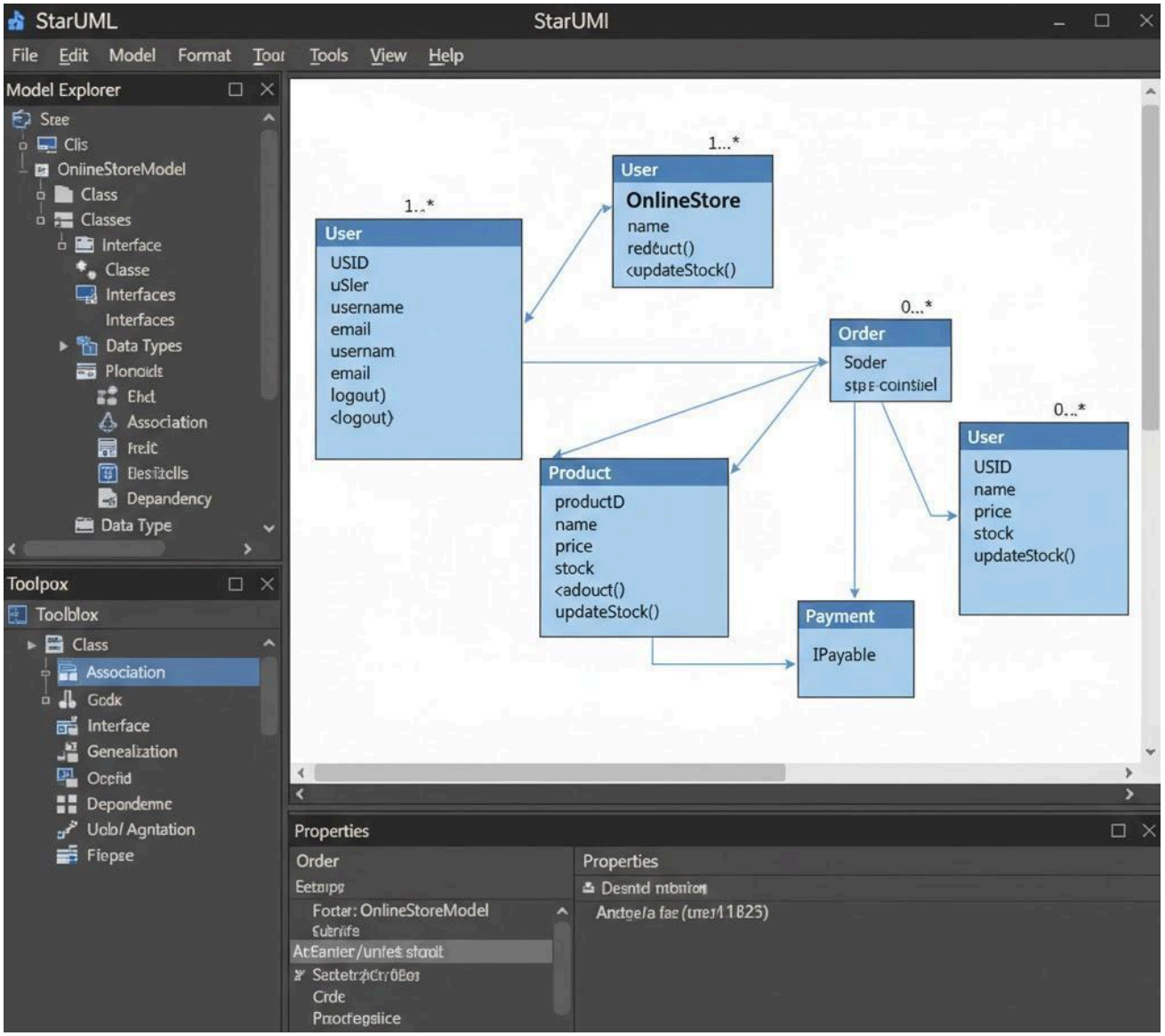
Advantages of StarUML:

- Cost-effective due to its open-source nature.
- Lightweight and fast performance.
- Good for individual developers and small teams.
- Active community support.

Disadvantages of StarUML:

- May have fewer advanced features compared to commercial tools.
- Documentation can sometimes be less comprehensive.

Here's an example of a class diagram created in StarUML:



3. IBM Rational Rose

IBM Rational Rose is a commercial, object-oriented software engineering tool used for visual modeling and component construction. It was one of the earliest and most widely adopted UML tools, especially in large enterprises.

Key Features of IBM Rational Rose:

- **Comprehensive Toolset:** Offers a wide range of features for all stages of software development.
- **Integration:** Integrates well with other IBM Rational products and development environments.
- **Robust Code Generation:** Strong capabilities for generating code in multiple languages.
- **Configuration Management:** Includes features for version control and configuration management.
- **Requirements Management:** Supports linking models to requirements.

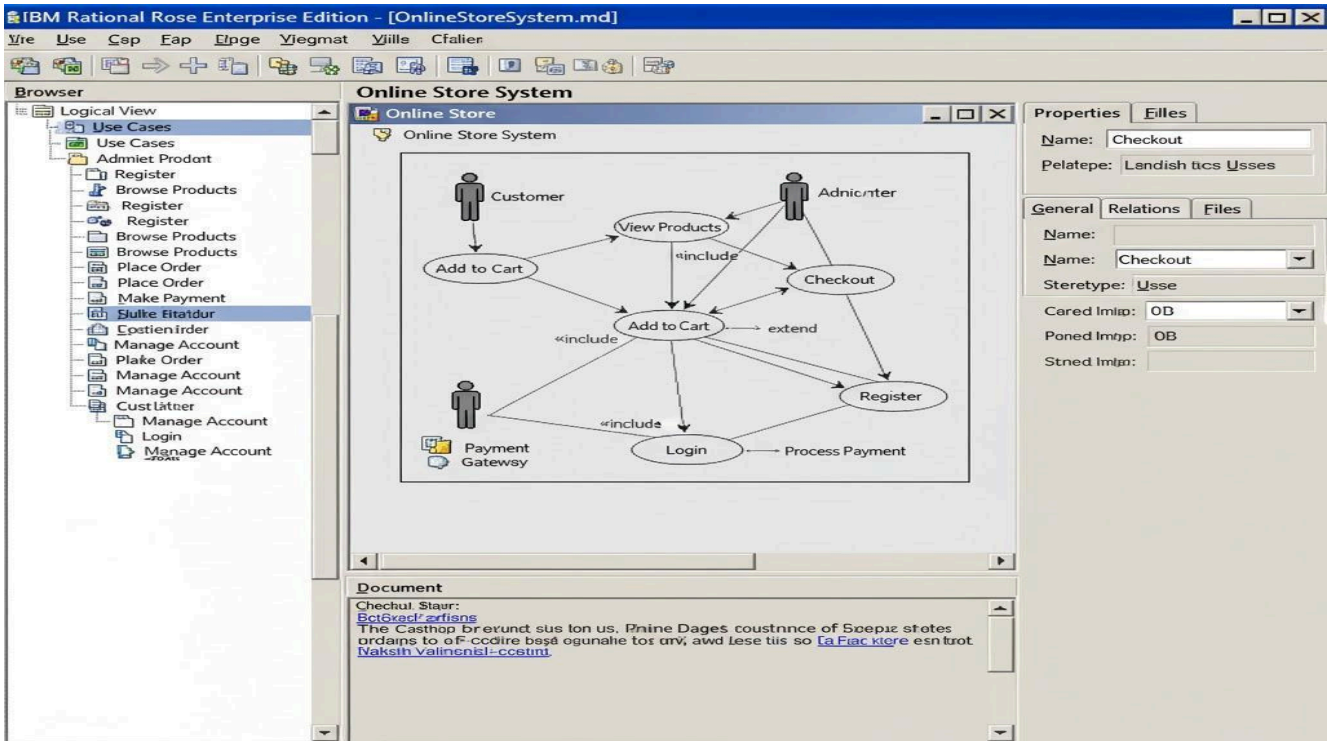
Advantages of IBM Rational Rose:

- Mature and stable product with extensive features.
- Strong support from IBM, suitable for enterprise-level projects.
- Robust collaboration features for large teams.
- Good for complex systems and rigorous development processes.

Disadvantages of IBM Rational Rose:

- High licensing cost.
- Can be resource-intensive and have a steeper learning curve.
- May be considered overkill for smaller projects.

Here's an example of a use case diagram created in IBM Rational Rose:



EXPERIMENT - 15

AIM - To understand why Entity Relationship Diagrams (ERDs) are used and to demonstrate their connectivity with a database using an example.

1. Introduction to Entity Relationship Diagrams (ERD)

An **Entity Relationship Diagram (ERD)**, or ER Diagram, is a graphical representation of a database's structure. It is a conceptual blueprint that illustrates the *entities* (data objects), their *attributes* (properties) and the *relationships* between them.

Think of it as the architectural plan for a house. Before you start building (writing code or creating tables), you use a blueprint (the ERD) to plan what rooms (entities) you need, what features each room will have (attributes) and how the rooms are connected (relationships).

---

2. Why Are ERDs Used?

ERDs are a fundamental tool in database design and software engineering for several key reasons:

- **Clear Visualization:** They provide a simple, graphical way to visualize a complex database structure, making it easy to understand at a glance.
- **Effective Communication:** ERDs act as a common language between different stakeholders, including database designers, developers, business analysts and clients. Everyone can look at the diagram and agree on what data is being stored and how it's connected.
- **Database Design & Planning:** They are the first step in designing a logical database. They help designers identify all the data needed, how to group it and how to link it, ensuring a well-organized and efficient database.
- **Reduces Redundancy and Errors:** By planning the relationships, designers can avoid data duplication (data redundancy) and prevent inconsistencies. For example, a customer's address can be stored once in a Customer table instead of being repeated for every order they make.
- **Documentation:** An ERD serves as essential documentation for the database. Any new developer joining the project can refer to the ERD to quickly understand the database schema.

---

3. Connectivity: From ERD to Database

An ERD is a conceptual model. The *connectivity* lies in how this conceptual model is **mapped** or **translated** into a physical database (like one in SQL).

Here is the direct mapping:

ERD Component	Database Component
Entity (e.g., Student, Course)	**Table** (e.g., Students table, Courses table)
Attribute (e.g., StudentName, CourseTitle)	**Column** (e.g., StudentName column, CourseTitle column)
Primary Key (A unique attribute)	**Primary Key (PK)** constraint on a column
Relationship (e.g., "Student enrolls in Course")	**Foreign Key (FK)** constraint (or a **Junction Table**)

How Relationships are Connected:

- **One-to-Many (1:M):** (e.g., One Customer can have many Orders). This is implemented by placing the **primary key** of the "one" side (e.g., CustomerID) into the table of the "many" side (e.g., the Orders table) as a **foreign key**.
- **Many-to-Many (M:N):** (e.g., One Student can enroll in many Courses and one Course can have many Students). This is the most complex. You **cannot** connect them directly. Instead, you create a third table, known as a **junction table** or **associative entity** (e.g., Enrollment). This table holds the primary keys from *both* tables, linking them together.

---

4. Example: A University Enrollment System

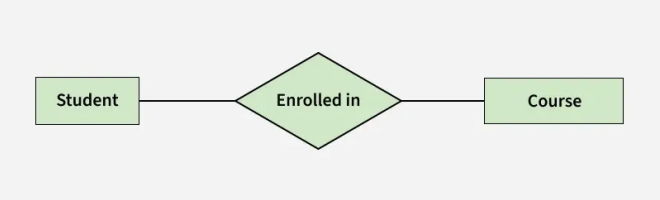
Let's illustrate the entire process with a simple example of students enrolling in courses.

Step 1: The Conceptual ERD

First, we identify the entities, their attributes and the relationship.

1. **Entities:**
    - Student
    - Course
2. **Attributes:**
    - Student: **StudentID** (Primary Key), StudentName, Email
    - Course: **CourseID** (Primary Key), CourseName, Credits
3. **Relationship:**
    - A student can enroll in *many* courses.
    - A course can have *many* students.
    - This is a **Many-to-Many (M:N)** relationship.

This would be drawn as follows (using Crow's Foot notation, where the "crow's foot" symbol means "many"):



Step 2: The Physical Database Connectivity

We now translate this ERD into a physical database schema.

- The Student entity becomes the Students table.
- The Course entity becomes the Courses table.
- The **"Enrolls In" M:N relationship** cannot be directly implemented. We *must* create a **junction table** called Enrollment to connect them

Students Table

Column	**Type**	**Constraints**
StudentID	INT	**Primary Key (PK)**

StudentName	VARCHAR(100)	
Email	VARCHAR(100)	

Courses Table

Column	**Type**	**Constraints**
CourseID	INT	**Primary Key (PK)**
CourseName	VARCHAR(100)	
Credits	INT	

Enrollment Table (The Junction Table)

Column	**Type**	**Constraints**
EnrollmentID	INT	**Primary Key (PK)**
StudentID	INT	**Foreign Key (FK)** -> Students(StudentID)
CourseID	INT	**Foreign Key (FK)** -> Courses(CourseID)
Grade	CHAR(2)	

Explanation of the Connectivity:

- The M:N relationship from the ERD is now *resolved* by the Enrollment table.
- If you want to find all courses for "Student 101", you query the Enrollment table for StudentID = 101.
- If you want to find all students in "Course 500", you query the Enrollment table for CourseID = 500.
- The **foreign keys** (Enrollment.StudentID and Enrollment.CourseID) are the physical connection that enforces this relationship at the database level, ensuring data integrity.

5. Conclusion

Entity Relationship Diagrams are an essential tool in software engineering. They serve as the conceptual blueprint for a database, facilitating clear communication and effective design. The *connectivity* between an ERD and a database is the direct mapping of entities to tables, attributes to columns and relationships to foreign keys or junction tables. This process translates the logical design into a physical, functional database.