PARKING GARAGE SYSTEM

KURT DELACRUZ
VISHAL VASANTHAKUMAR
RAYMOND SANGALANG
JEFFERSON LY

PHASE 2 - DESIGN

DESIGN

1 INTRODUCTION

- Purpose
- Scope
- System Overview
  - Vehicle Entry/Exit Management
  - Payment Processing
  - Logging & Garage Reports
  - Administration & Maintenance

2 USE CASES / SEQUENCE DIAGRAMS

- Login & Logout
- Ticket Handling
- Handling Parking Data
- Payment Processing
- Auto vs Manual Payment
- Add Parking Levels

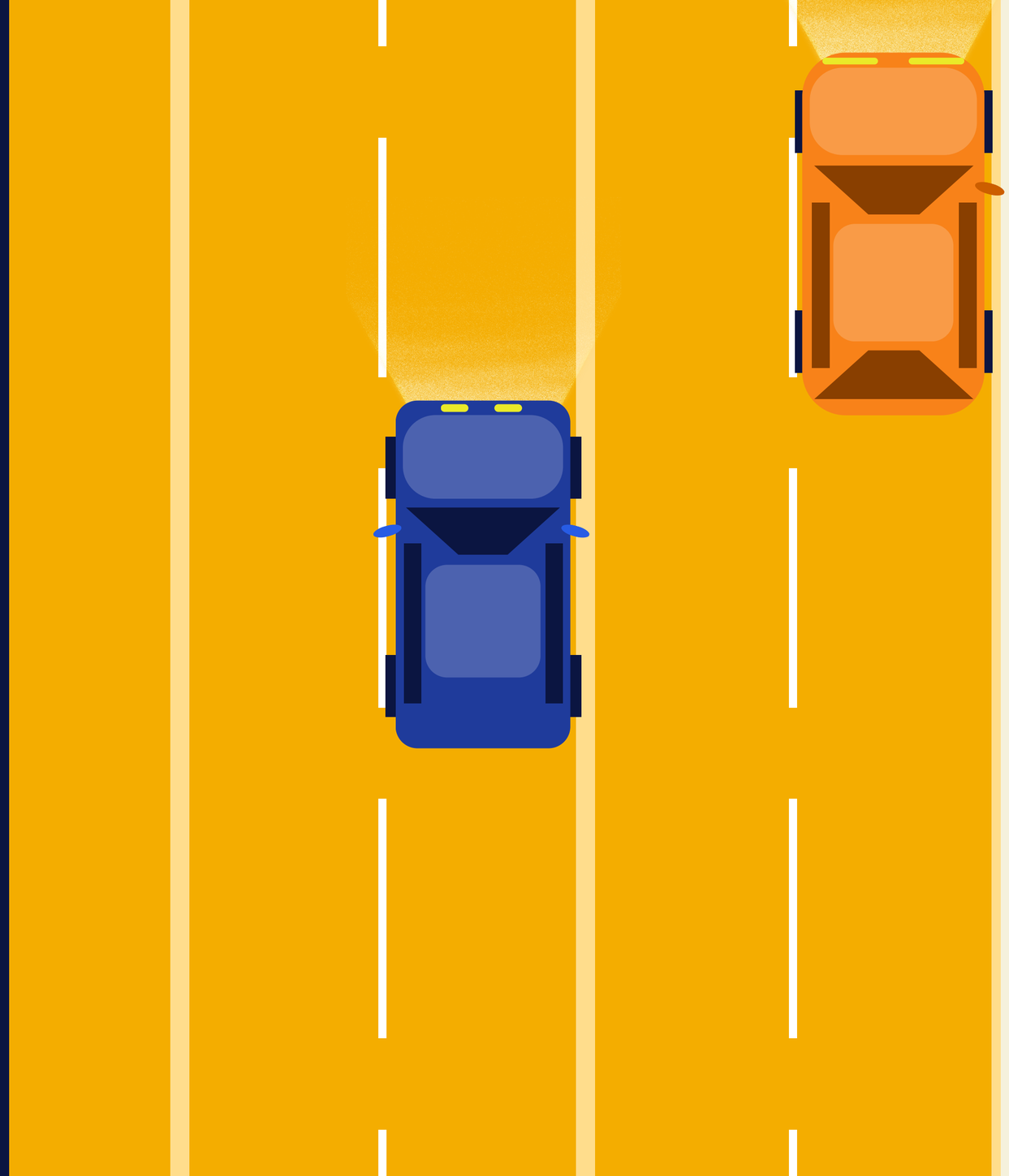
3 CLASSES

- Parking Garage
- Parking Space
- Vehicle
- Ticket
- Payment
- Employee
- Customer
- Parking Attendant
- SystemLog
- Parking Level
- Entrance Display Board
- Entry Kiosk
- Exit Kiosk
- Admin
- Hardware

INTRODUCTION:

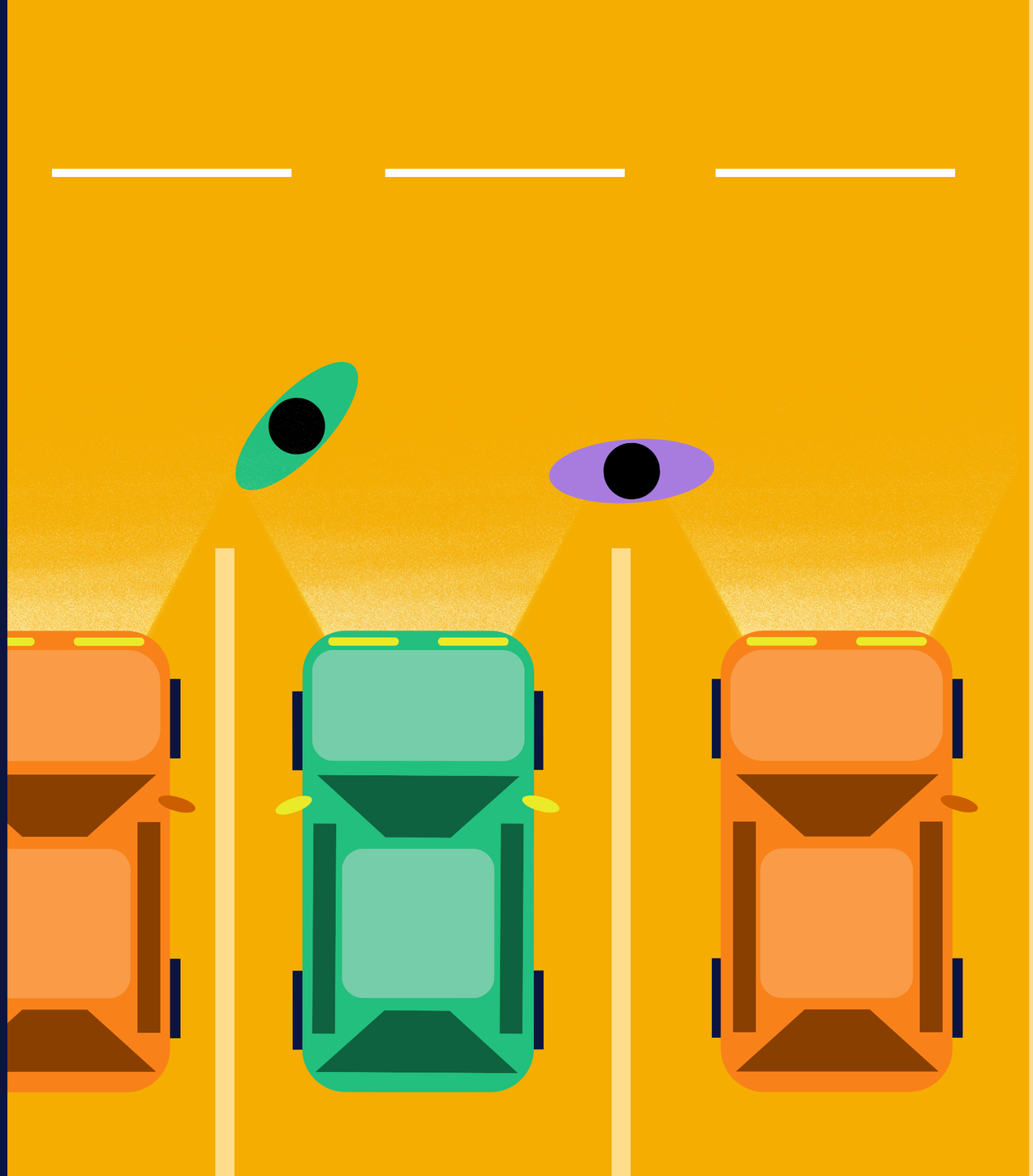
PURPOSE

- The parking garage software is to systematically manage and streamline an overhead for parking garage operations by tracking all vehicles entering and exiting, calculating cost, monitoring available parking spaces, and logging facility-related reports.



SCOPE

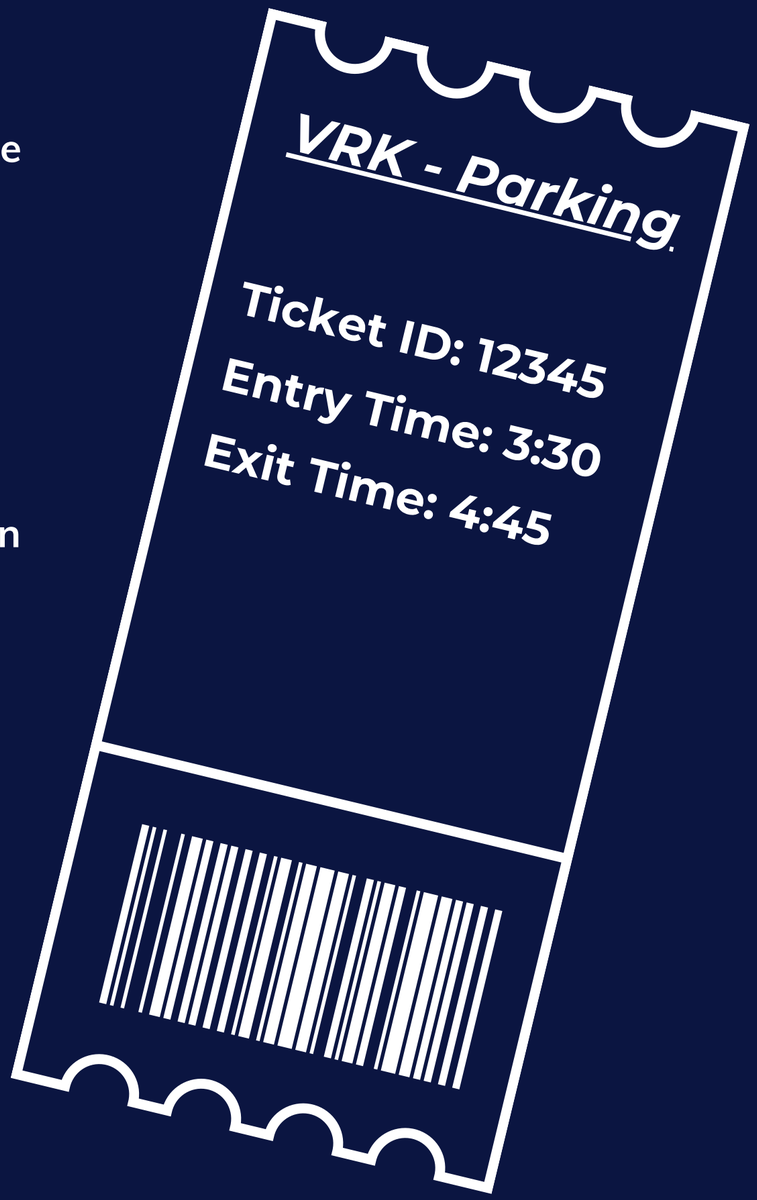
- The system allows customers to enter and exit the parking garage, track the available spaces, process payments, and generate usage reports. The system will also include features such as automated ticketing, real-time space monitoring, and integration through our payment system.



SYSTEM MODULES:

ENTRY MODULE-

- Upon arrival, the system detects an incoming vehicle using scanners and will utilize a ticket dispenser.
- If the garage has available space, the system generates a unique ticket with the following details:
  - Ticket ID (Unique Identifier)
  - Entry Time
  - Exit Time (Blank)
- The barrier gate automatically opens for the vehicle to enter after a ticket has been issued.



EXIT MODULE-

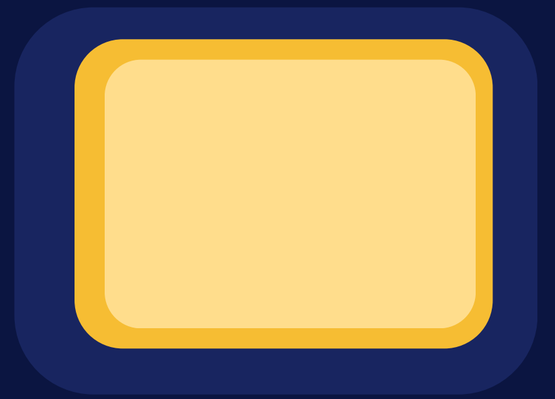
- The system calculates the total parking duration and fee based on a predefined rate.
- Payment is requested via automated kiosks or manual employee processing.
- Upon successful payment, the system:
  - Updates the garage database to free up the occupied parking space.
  - Generates a printed receipt/updated ticket.
  - Opens the exit barrier for the vehicle to leave.
- If a vehicle attempts to exit without a valid ticket, an employee override option is available to manually process the exit once validated.

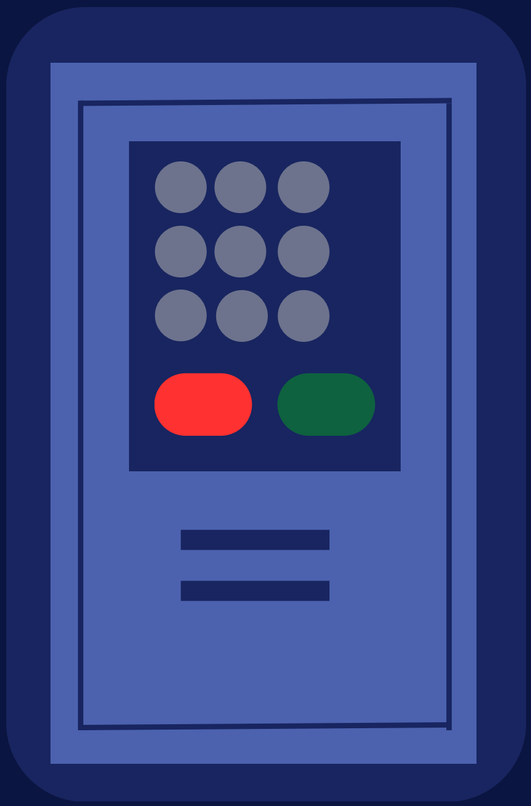
LOGGING AND GARAGE REPORTS

- **Types of Logged Data:**
  - **Occupancy Reports:**
    - Daily, weekly, and monthly reports showing parking utilization trends.
  - **Revenue Reports:**
    - Total revenue generated per time period, categorized by payment method.
    - Breakdown of penalties and manual adjustments.
  - **Employee Activity Reports:**
    - Login/logout timestamps for employees handling transactions.
    - Employee overrides (e.g., manually processed exits).
  - **Security Logs:**
    - Unauthorized access attempts.
    - Sensor malfunctions and system errors.
  - **Data Accessibility:**
    - Admins and managers can generate reports through a secured web interface.
    - Reports can be exported as a text file.
    - System logs are archived periodically for auditing and compliance.

PAYMENT PROCESSING-

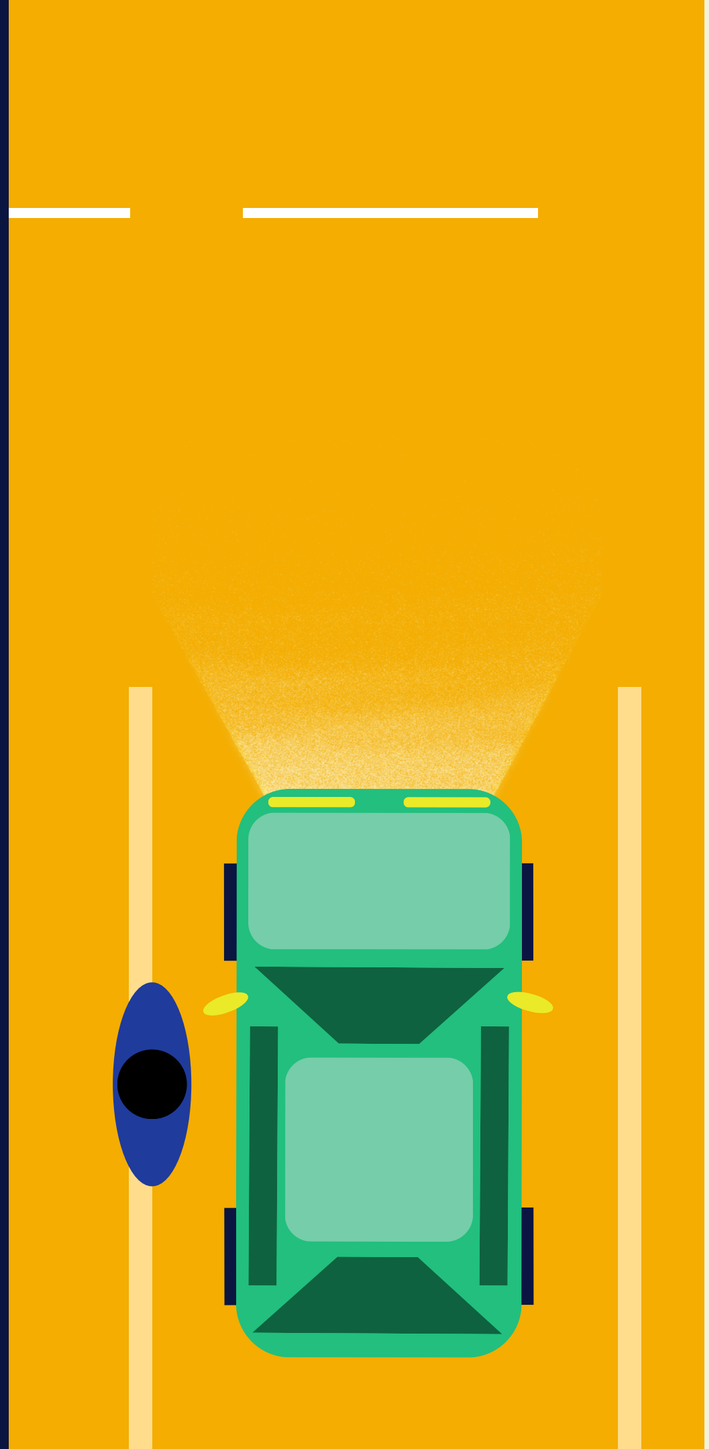
- Automated Fee Calculation:
  - The parking fee is determined based on:
    - Duration of stay (time-in vs. time-out).
    - Parking zone (regular, premium, reserved).
  - The system automatically computes and displays the payable amount upon ticket scanning.
- Enables Multiple Payment Methods:
  - Automated Payments:
    - Credit/Debit Cards (via self-service kiosks or online portal).
    - Mobile Payments (Google Pay, Apple Pay, etc.).
  - Manual Payments:
    - Cash payments at the employee checkout counter.
    - Employee-assisted card transactions.
- Issuing Penalties:
  - Lost Ticket Fee: If a customer loses their ticket, an employee must manually retrieve entry details and charge a default fee.
  - Overstay Charges: Vehicles parked beyond permitted hours will be charged extra, as per garage policy.
  - Unauthorized Parking Violations: If a vehicle is parked in a restricted zone, penalties are recorded and linked to the ticket.



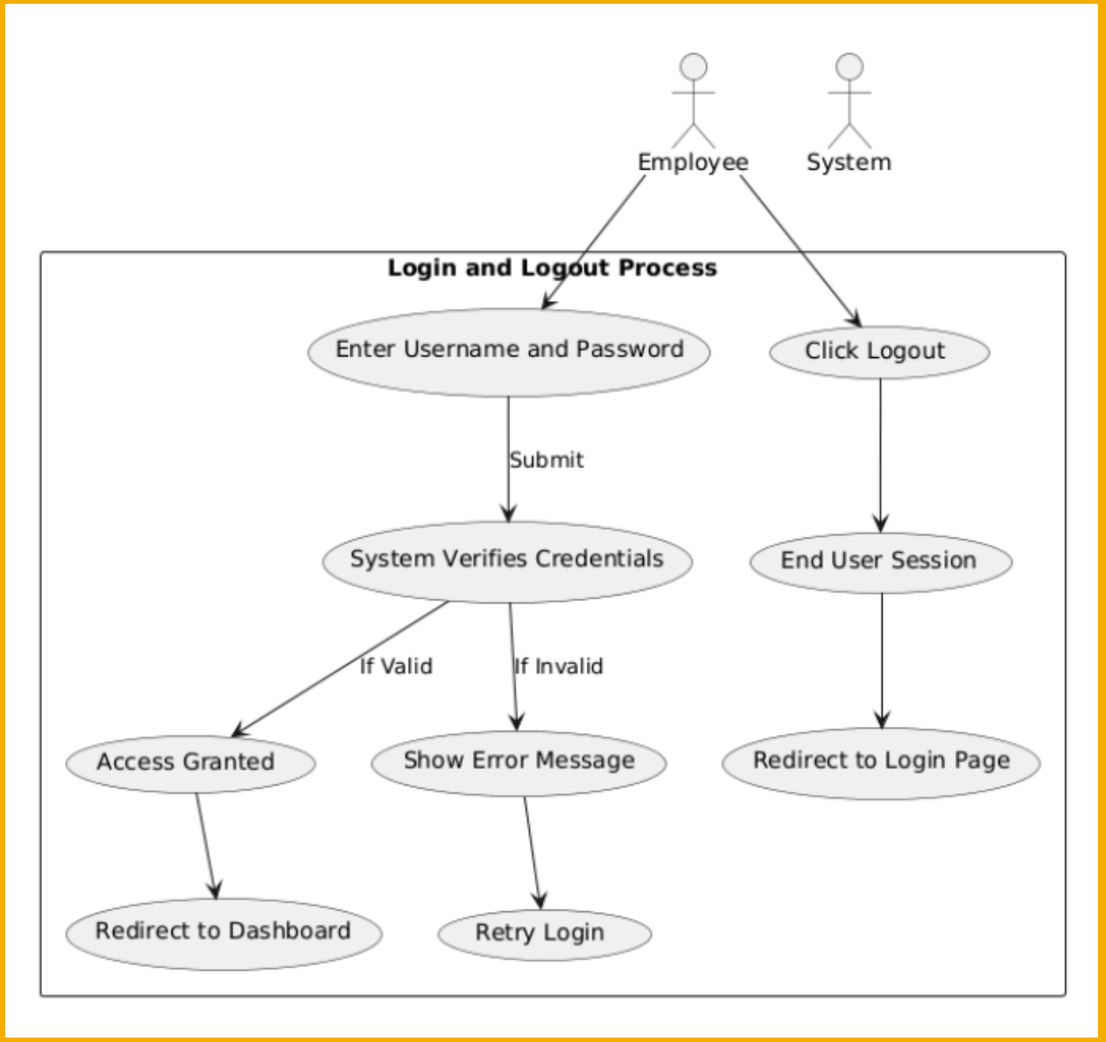


ADMINISTRATION & MAINTENANCE:

- **User Management:**
  - Employee access is role-based, granting permissions based on job responsibilities where managers can override employees and gain more access for logs and etc.
  - Credentials are required for login.
- **System Configuration:**
  - Parking rates, penalties, and policies can be configured through the admin dashboard.
  - Hardware components (sensors, gates, kiosks) can be monitored remotely for malfunctions.
- **Maintenance Alerts:**
  - The system detects and logs any software malfunctions and bugs by handling errors.
  - Admins receive automatic notifications when maintenance is required.
  - A maintenance tracking system logs completed repairs and scheduled servicing.
- **Multi-Garage Management:**
  - If multiple parking locations are managed under the same system, administrators can monitor all garages from a central dashboard.
  - Reports can be generated per garage or aggregated across multiple locations.



UML USE CASE: LOGIN & LOGOUT



Primary Actor: Employee (Attendant, Manager)

Pre-conditions: The system must be powered on; the user must have valid login credentials.

Post-conditions: The user is successfully logged in or out of the system.

Basic Flow:

1. The user navigates to the login page.
2. The user enters their username and password.
3. The system verifies the credentials.
4. If valid, the system grants access based on the user's role.
5. The user is redirected to the dashboard.
6. When finished, the user selects the logout option, and the session ends.

Extensions/Alternate Flows:

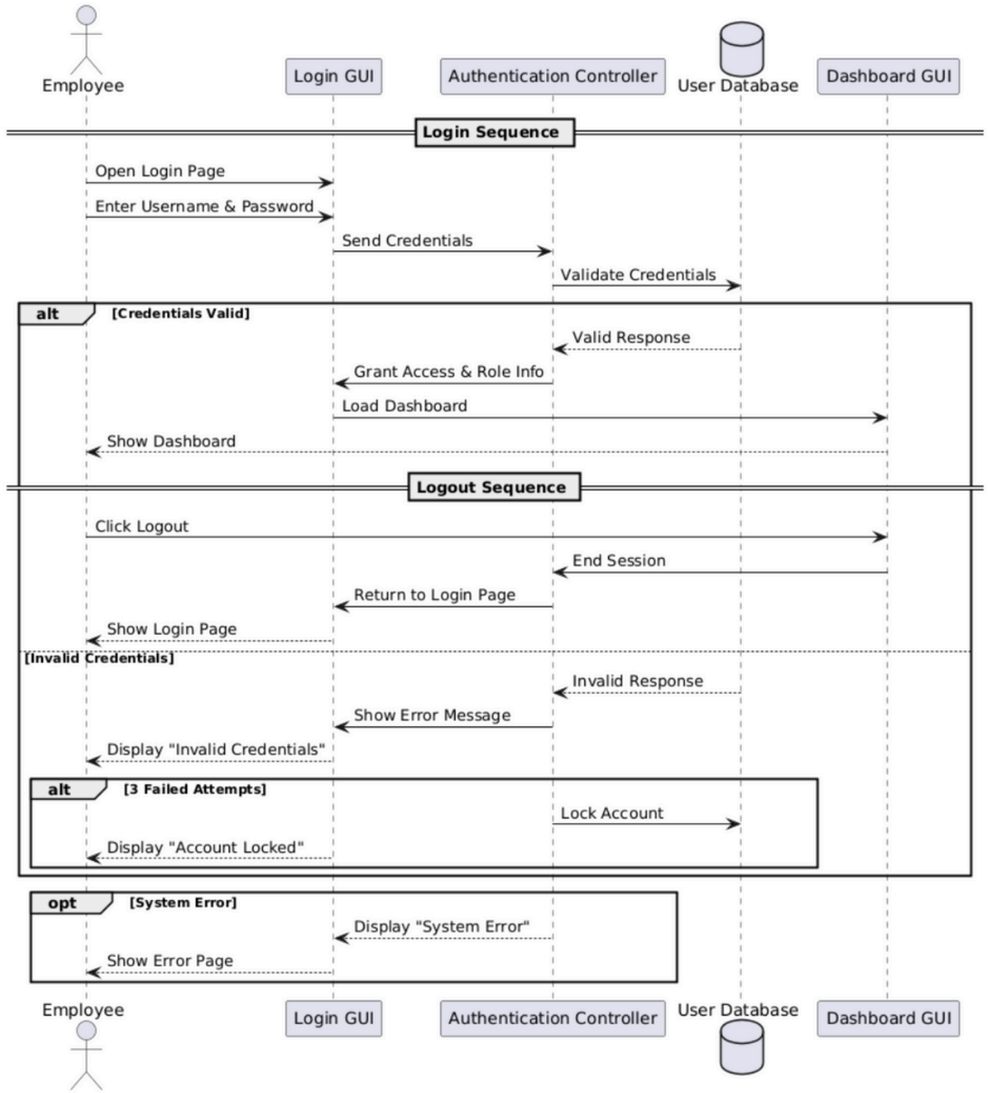
- If credentials are incorrect, the system displays an error message.
- If three failed attempts occur, the account is locked for security.

Exceptions:

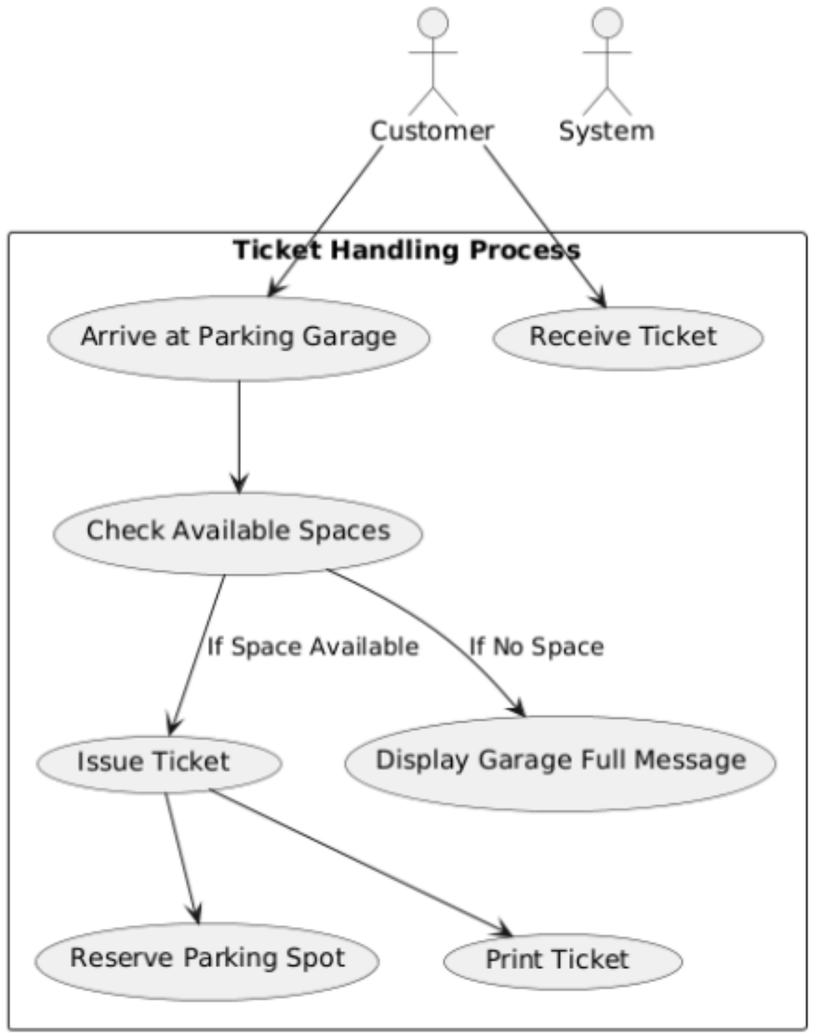
- System error preventing login.
- Unexpected session timeout during login.

Related Use Cases: Ticket Handling, Payment Processing

Sequence Diagram for Use Case 1: Login and Logout



UML USE CASE: TICKET HANDLING



Primary Actor: System, Customer

Pre-conditions: The parking garage has available space; the system is active.

Post-conditions: The customer receives a ticket, and a parking space is then occupied.

Basic Flow:

1. The customer arrives at the parking garage entry.
2. The system checks for available parking spaces.
3. If a spot is available, the system generates a ticket.
4. The system assigns a parking space and updates the total capacity.
5. The ticket is printed and given to the customer.

Extensions/Alternate Flows:

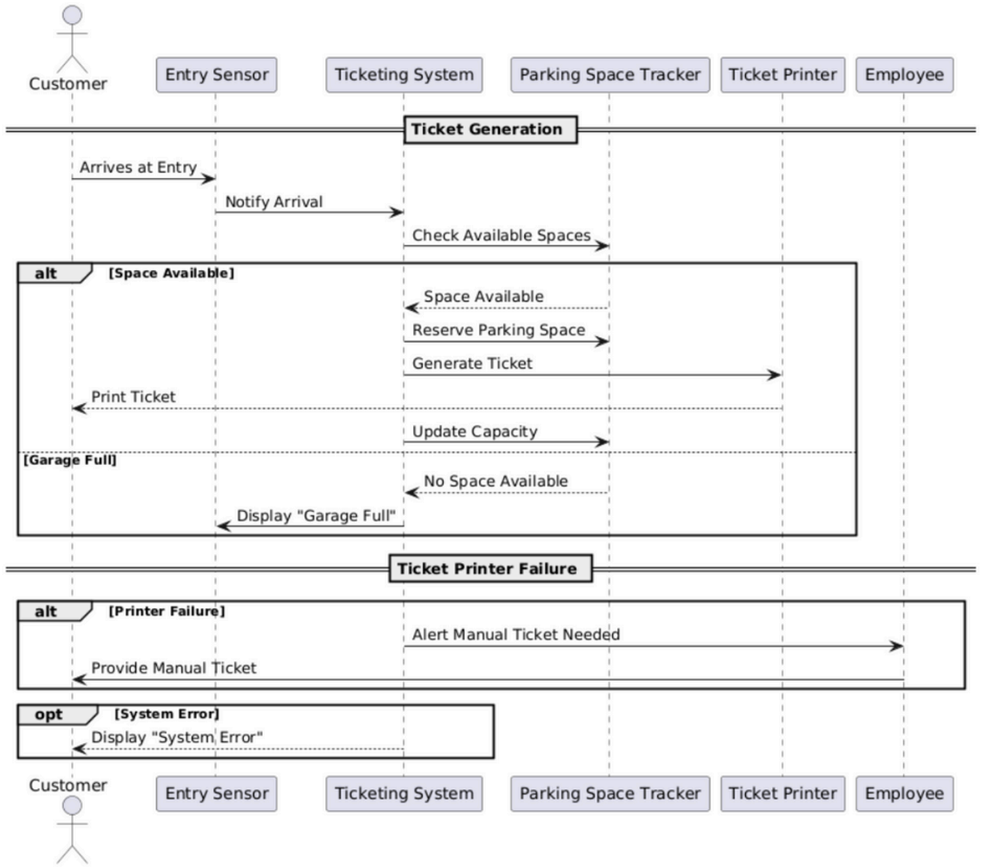
- If the garage is full, the system displays a "Garage Full" message and does not issue a ticket.
- If the ticket printer fails, an employee must manually generate a ticket.

Exceptions:

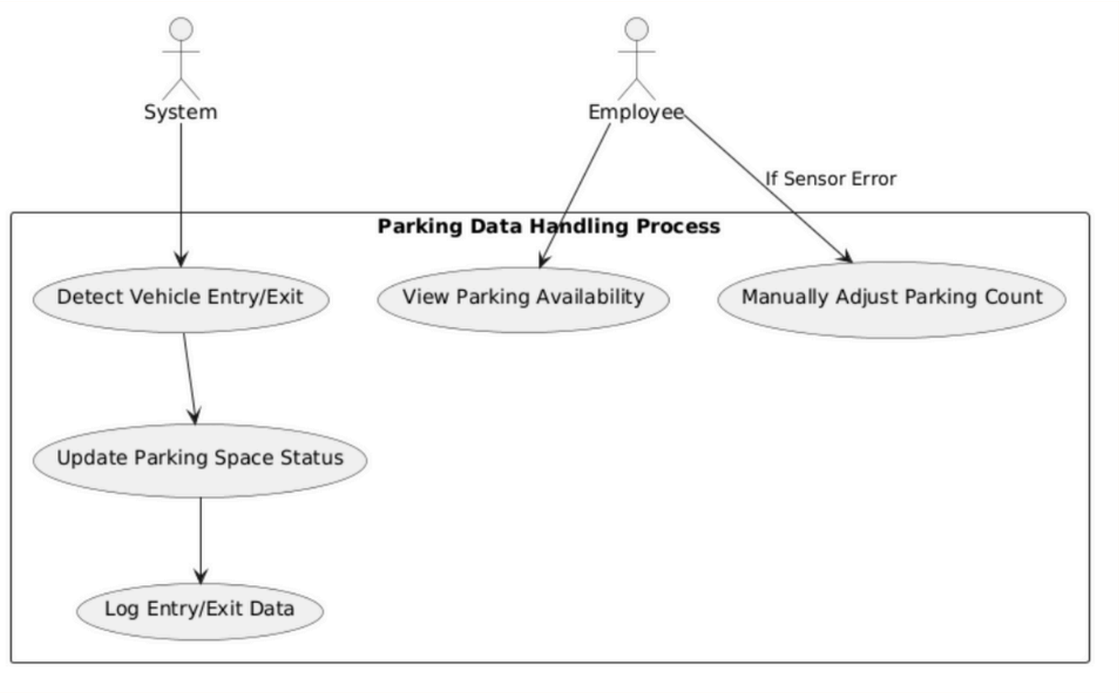
- System error preventing ticket issuance.

Related Use Cases: Handling Parking Data, Payment Processing

Sequence Diagram for Use Case 2: Ticket Handling



UML USE CASE: HANDLING PARKING DATA



Primary Actor: System, Employee

Pre-conditions: The system is running and actively tracking vehicles.

Post-conditions: The parking availability data is updated accurately.

Basic Flow:

1. A vehicle enters or exits the parking garage.
2. The system updates the number of occupied and available spaces.
3. The system assigns or frees up a parking space based on the transaction.
4. The system logs the event for tracking purposes.

Extensions/Alternate Flows:

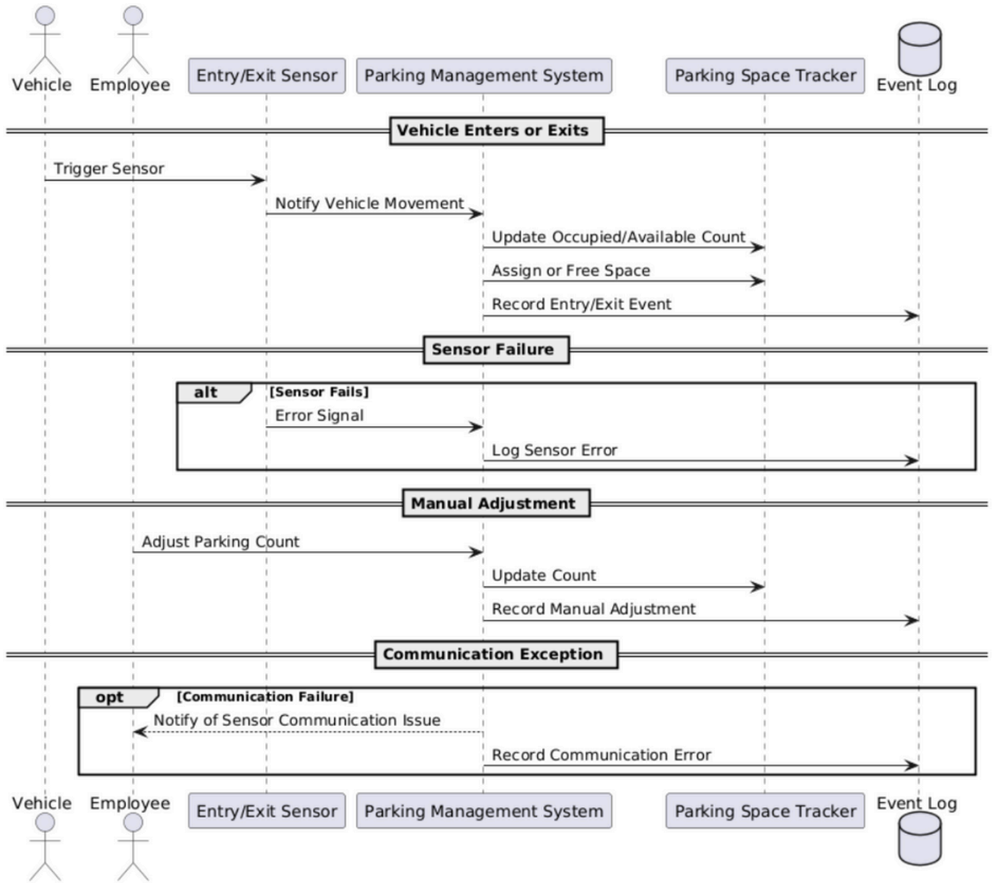
- If a sensor fails to detect a vehicle, the system logs an error.
- Employees can manually adjust parking counts in case of a discrepancy.

Exceptions:

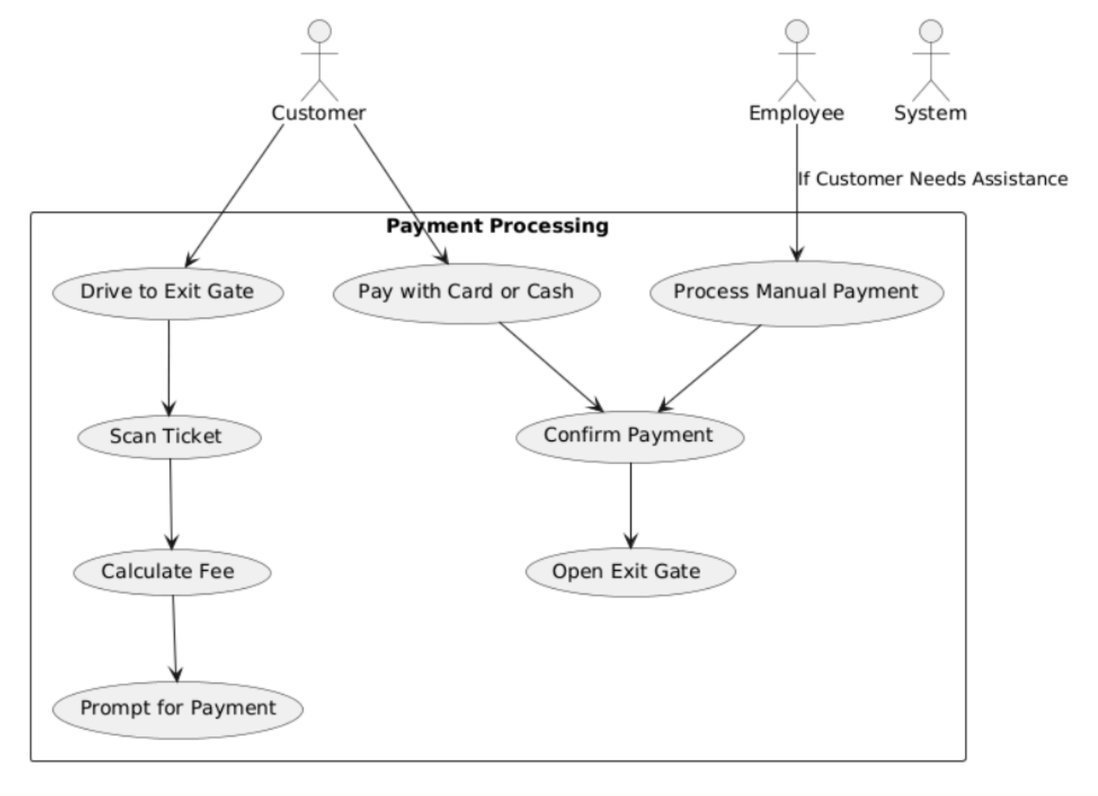
- System communication failure with parking sensors.

Related Use Cases: Ticket Handling, Payment Processing

Sequence Diagram for Use Case 3: Handling Parking Data



UML USE CASE: PAYMENT PROCESSING



Primary Actor: Customer, Employee, System

Pre-conditions: The customer has a valid ticket and is ready to leave.

Post-conditions: The payment is processed, and the exit gate opens.

Basic Flow:

1. The customer drives to the exit gate.
2. The system scans the ticket and calculates the fee based on duration.
3. The system prompts for payment.
4. The customer completes the payment using an accepted method (cash, card).
5. Upon successful payment, the system opens the exit gate.

Extensions/Alternate Flows:

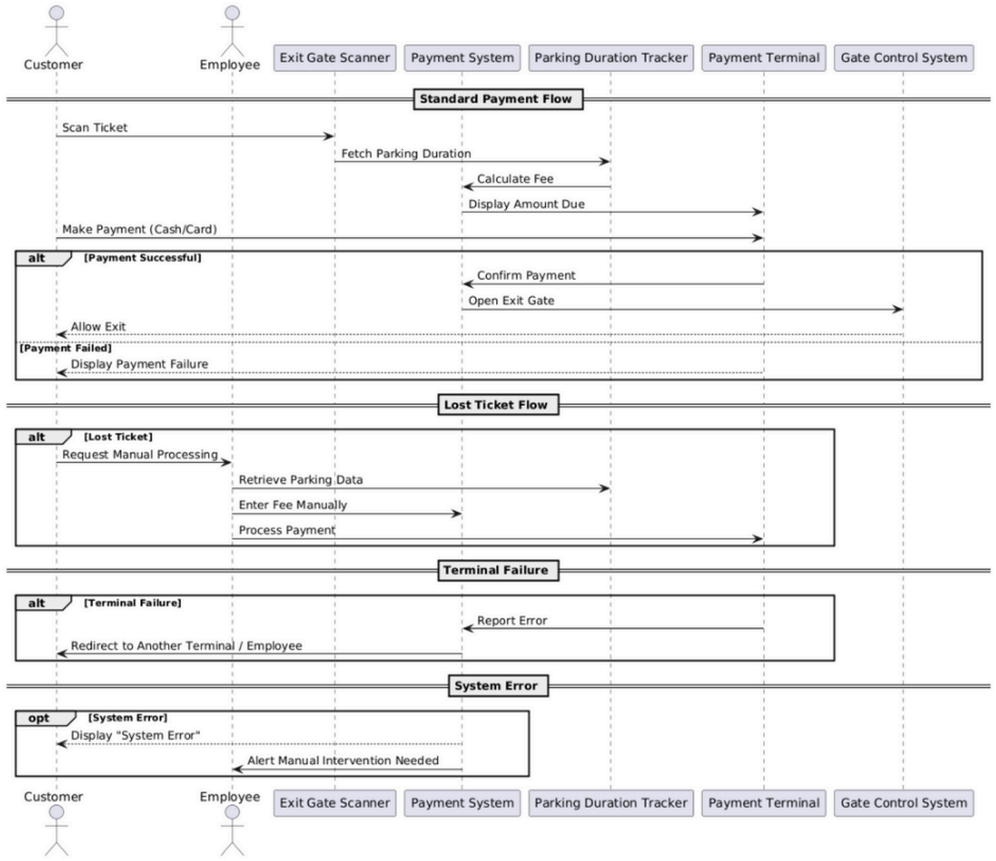
- If the customer lost their ticket, an employee manually retrieves parking data and processes the payment.
- If the payment terminal fails, the customer must use another terminal or pay manually.

Exceptions:

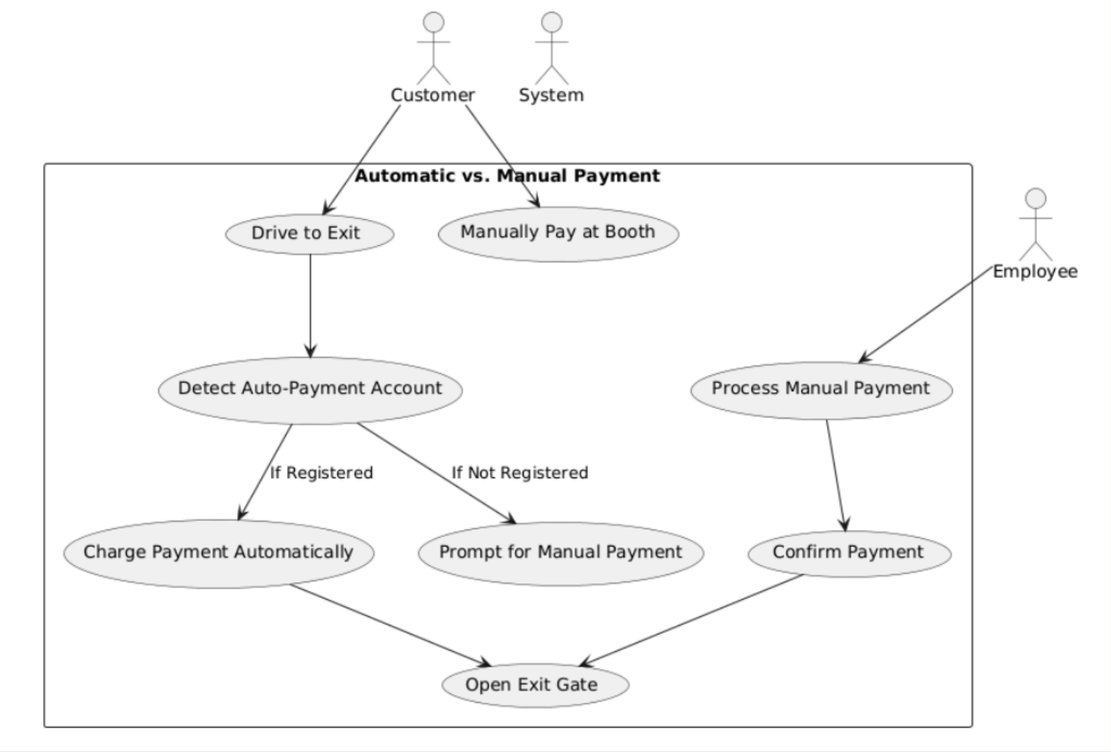
- System error preventing payment processing.

Related Use Cases: Automatic vs. Manual Payment

Sequence Diagram for Use Case 4: Payment Processing



UML USE CASE: AUTOMATIC VS MANUAL PAYMENT



Primary Actor: Customer, Employee, System

Pre-conditions: The system has ticket data stored for each customer.

Post-conditions: The customer is charged correctly based on the method used.

Basic Flow (Automatic Payment):

1. A customer with an auto-payment account is detected at the exit.
2. The system verifies the linked payment method and calculates the fee.
3. The system processes the payment automatically.
4. If successful, the exit gate opens.

Basic Flow (Manual Payment):

1. A customer without auto-payment drives to an employee checkout booth.
2. The employee scans the ticket and retrieves the parking fee.
3. The customer makes the payment.
4. The employee confirms payment and allows exit.

Extensions/Alternate Flows:

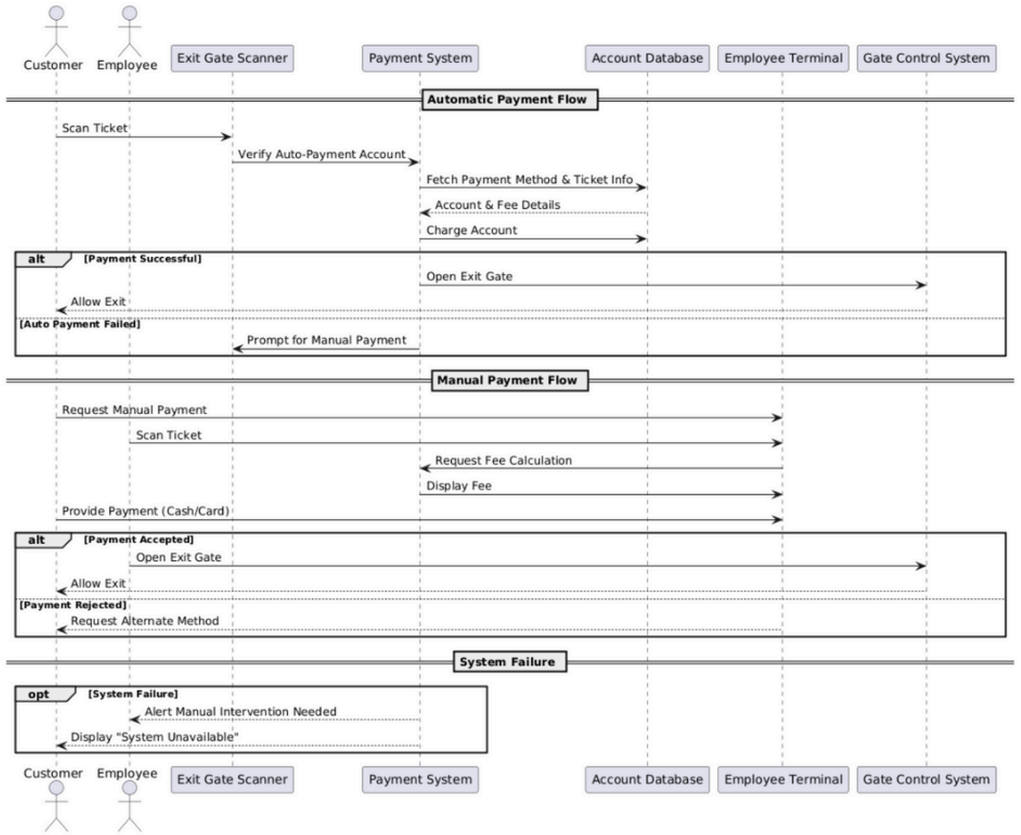
- If automatic payment fails, the system prompts for manual payment.
- If the employee is unable to process a payment, the customer must use another payment method.

Exceptions:

- System failure preventing payment authorization.

Related Use Cases: Payment Processing, Ticket Handling

Sequence Diagram for Use Case 5: Automatic vs. Manual Payment Handling



UML USE CASE: ADD PARKING LEVELS

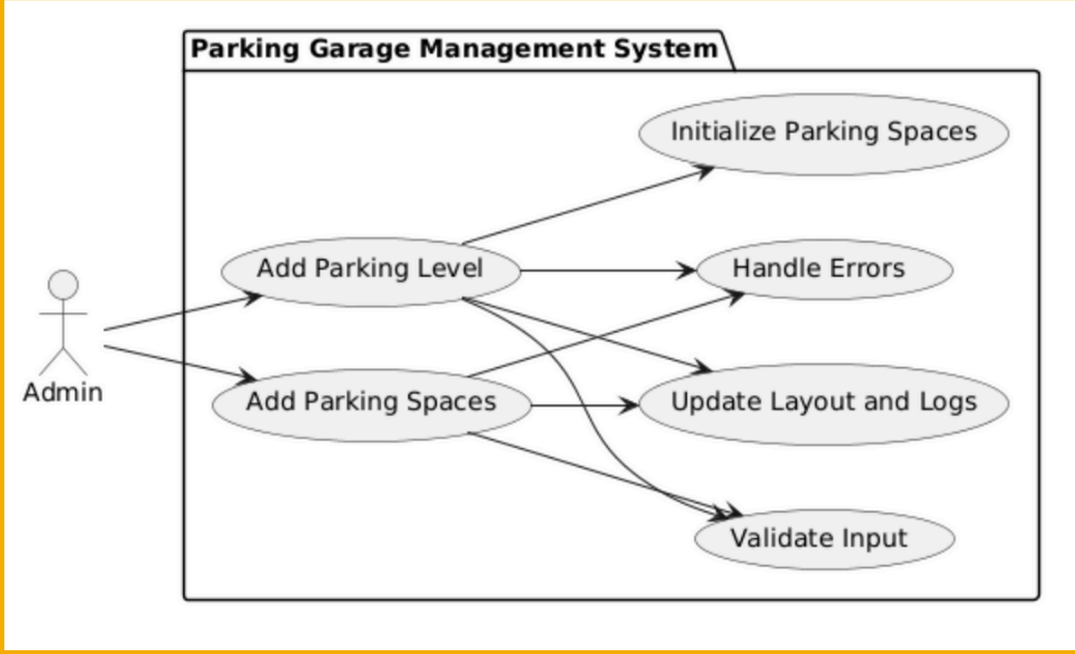
Primary Actor: Admin

Pre-conditions: Admin is authenticated and logged into the system. The system is operational and connected to the central database or employee files.

Post-conditions: The new parking level or spaces are successfully added and visible in the system. The garage layout is updated. Reported logs are updated to reflect the changes made to the system.

Basic Flow (Automatic Payment):

1. Admin selects "Add Parking Level" from the system dashboard.
2. System prompts for:
   - Level ID or Name
   - Number of parking spaces
   - Level-specific attributes (e.g., access restrictions, vehicle type support)
3. Admin enters the required information.
4. Admin clicks "Submit".
5. System validates input.
6. System adds the new level and initializes a specified number of empty parking spaces.
7. Confirmation is displayed to the admin.



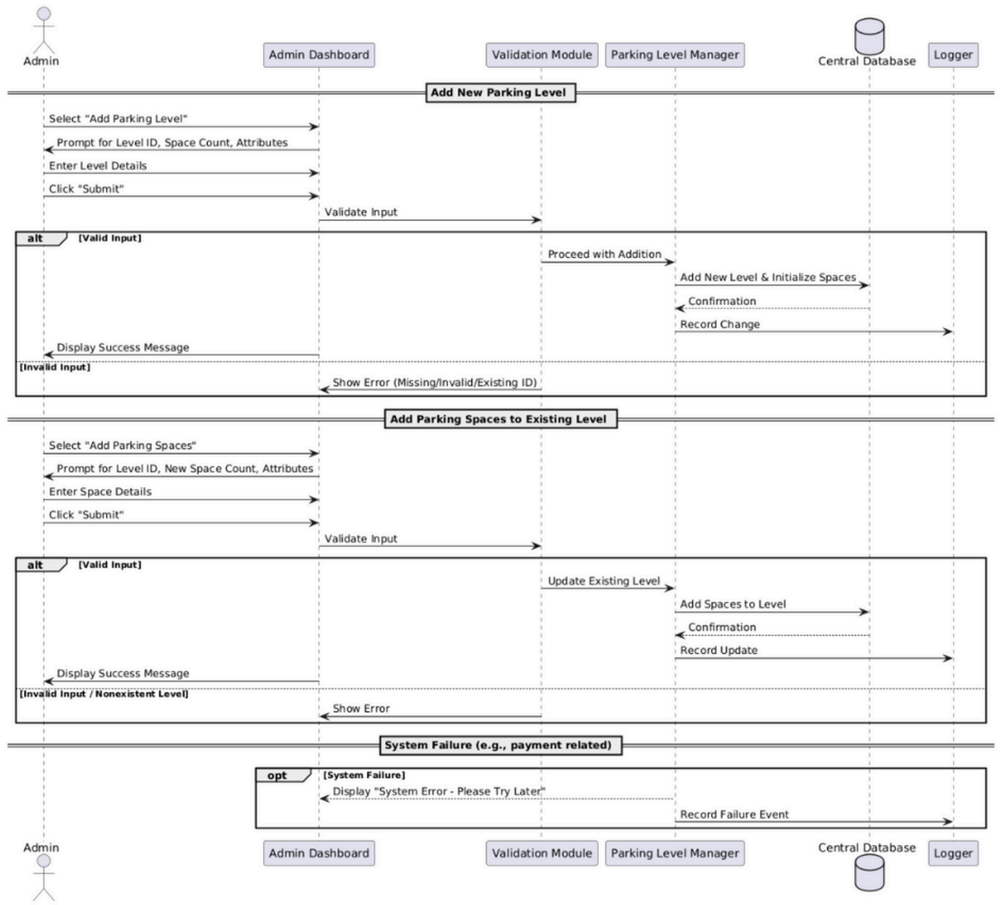
Extensions/Alternate Flows:

- If automatic payment fails, the system prompts for manual payment.
- If the employee is unable to process a payment, the customer must use another payment method.

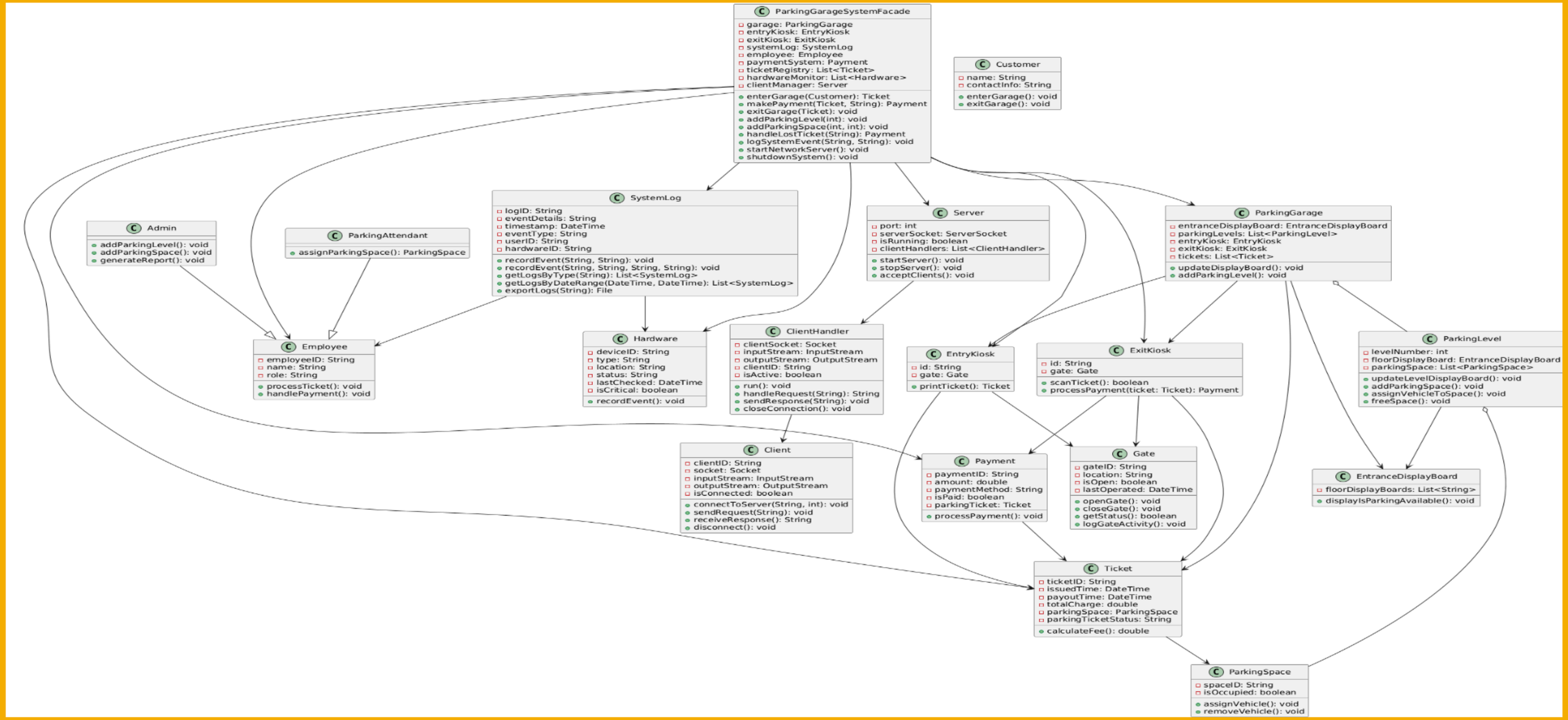
Exceptions:

- **System failure:** Preventing payment authorization.
- **Invalid Input:** If required fields are missing or invalid, the system displays an error and requests correction.
- **Duplicate Level ID:** If a Level ID already exists, the system prompts admin to enter a unique identifier.
- **Nonexistent Level (when adding space):** System shows an error if the specified level does not exist.

Sequence Diagram for Use Case 6: Add Parking Levels



CLASS DIAGRAM



ParkingGarage

Class Variables

- entranceDisplayBoard: EntranceDisplayBoard – Displays parking availability to oncoming vehicles.
- parkingLevels: List<ParkingLevel> – A list of all parking levels in the garage.
- entryKiosk: EntryKiosk – Kiosk where vehicles enter and tickets are printed.
- exitKiosk: ExitKiosk – Kiosk for scanning tickets and handling payments.
- tickets: List<Ticket> – A record of all issued parking tickets.

Methods:

- updateDisplayBoard(): void – Updates the entrance display based on parking availability.
- addParkingLevel(): void – Adds a new parking level to the garage.

ParkingLevel

Class Variables

- levelNumber: int – The level number or identifier.
- floorDisplayBoard: EntranceDisplayBoard – A display board showing availability on this level.
- parkingSpace: List<ParkingSpace> – List of parking spaces on the level.

Methods

- updateLevelDisplayBoard(): void – Updates the floor-level display.
- addParkingSpace(): void – Adds new spaces to this level.
- assignVehicleToSpace(): void – Assigns a vehicle to an available space.
- freeSpace(): void – Frees a parking space when a vehicle exits.

ParkingSpace

Class Variables

- spaceID: String – Unique identifier for the parking space.
- isOccupied: boolean – Indicates whether the space is currently occupied.

Methods

- assignVehicle(Vehicle v): void – Marks the space as occupied by the vehicle.
- removeVehicle(): void – Frees the space when a vehicle exits.

EntranceDisplayBoard

Class Variables

- floorDisplayBoards: List<String> – Text/status displays for each level.

Methods

- displayIsParkingAvailable(): void – Displays whether parking is currently available.

Ticket

Class Variables

- ticketID: String – Unique ticket identifier.
- issuedTime: DateTime – Time the ticket was issued.
- payoutTime: DateTime – Time the customer exits.
- totalCharge: double – Total amount to be paid.
- parkingSpace: ParkingSpace – Space assigned to the ticket.
- parkingTicketStatus: String – Status ("Active", "Paid", "Lost").

Methods

- calculateFee(): double – Calculates total parking fee based on duration.

Payment

Class Variables

- paymentID: String – Unique ID for the payment transaction.
- amount: double – Amount to be paid.
- paymentMethod: String – Payment type (e.g., "Card", "Cash").
- isPaid: boolean – Status of the payment.
- parkingTicket: Ticket – Associated ticket being paid for.

Methods

- processPayment(): void – Marks the payment as completed.

EntryKiosk

Class Variables

- id: String – Identifier for the kiosk.
- gate: Gate – Entry gate controlled by the kiosk.

Methods

- printTicket(vehicle: Vehicle): Ticket – Prints a new ticket for the entering vehicle.

ExitKiosk

Class Variables

- id: String – Identifier for the kiosk.
- gate: Gate – Exit gate controlled by the kiosk.

Methods

- scanTicket(): boolean – Scans the customer's ticket to validate.
- processPayment(ticket: Ticket): Payment – Processes payment before exit.

Employee

Class Variables

- employeeID: String – Unique identifier for the employee.
- name: String – Full name of the employee.
- role: String – Role of the employee (e.g., Attendant, Admin).

Methods:

- processTicket(): void – Handles the verification and management of parking tickets.
- handlePayment(): void – Manages the processing of customer payments, including manual or automated methods.

ParkingAttendant extends Employee

Class Variables

- All inherited from the Employee class.

Methods

- assignParkingSpace(vehicle): ParkingSpace – Assigns an appropriate available parking space to the incoming vehicle based on size, type, and current availability.

Admin extends Employee

Class Variables

- All inherited from the Employee class.

Methods

- addParkingLevel(): void – Adds a new parking level to the system.
- addParkingSpace(): void – Adds new parking spaces to an existing level.
- generateReport(): Report – Generates reports on garage activity, revenue, or performance.

Customer

Class Variables:

- name: String – Full name of the customer.
- contactInfo: String – Contact details (e.g., phone number or email).

Methods:

- enterGarage(): void – Allows the customer to enter the parking garage.
- exitGarage(): void – Allows the customer to exit after completing payment.

SystemLog

Class Variables:

- logID: String – Unique identifier for each log entry.
- eventDetails: String – Description of the logged event (e.g., "Payment successful", "Vehicle entered", "Sensor failure").
- timestamp: DateTime – Date and time the event occurred.
- eventType: String – Type of event ("INFO", "ERROR", "WARNING", "SECURITY").
- userID: String – (Optional) ID of the user or employee associated with the event.
- hardwareID: String – (Optional) ID of hardware involved in the event.

Methods:

- recordEvent(String eventDetails, String eventType): void – Logs a new system event with the current timestamp.
- recordEvent(String eventDetails, String eventType, String userID, String hardwareID): void – Overloaded version to capture more context.
- getLogsByType(String eventType): List<SystemLog> – Retrieves logs filtered by event type.
- getLogsByDateRange(DateTime from, DateTime to): List<SystemLog> – Retrieves logs within a specific time range.
- exportLogs(String format): File – Exports logs for auditing in CSV, PDF, etc.

Hardware

Class Variables

- deviceID: String – Unique identifier for the hardware component.
- type: String – Type of device ("scanner", "gate", "camera", "sensor" etc.).
- location: String – Physical location within the garage.
- status: String – Operational status ("active", "offline", "maintenance").
- lastChecked: DateTime – Last time the device was tested or pinged.
- isCritical: boolean – Indicates if it's critical to garage operation (e.g., gates, ticket printers).

Methods

- recordEvent(): Logs system activities such as vehicle entries, payments, and errors.

Client

Class Variables

- clientID: String – Unique ID for the client instance.
- socket: Socket – The network socket used for communication.
- inputStream: InputStream – Stream to receive data from the server.
- outputStream: OutputStream – Stream to send data to the server.
- isConnected: boolean – Tracks the connection status with the server.

Methods

- connectToServer(String serverIP, int port): void – Establishes a connection to the server.
- sendRequest(String request): void – Sends a formatted request to the server.
- receiveResponse(): String – Waits for and returns the server's response.
- disconnect(): void – Gracefully disconnects the client from the server.

Server

Class Variables

- port: int – The port number the server listens on.
- serverSocket: ServerSocket – The server's listening socket.
- isRunning: boolean – Indicates whether the server is actively running.
- clientHandlers: List<ClientHandler> – Tracks all connected client sessions.

Methods

- startServer(): void – Initializes the server and starts listening for connections.
- stopServer(): void – Shuts down the server and closes all connections.
- acceptClients(): void – Accepts incoming client connections and spawns ClientHandler threads.

ClientHandler

Class Variables

- clientSocket: Socket – The socket for communication with the specific client.
- inputStream: InputStream – Input stream from the client.
- outputStream: OutputStream – Output stream to the client.
- clientID: String – Unique ID for the connected client.
- isActive: boolean – Indicates if the handler is actively managing the connection.

Methods

- run(): void – Main execution loop for receiving and processing client messages.
- handleRequest(String request): String – Interprets a request and generates an appropriate response.
- sendResponse(String response): void – Sends a response back to the client.
- closeConnection(): void – Closes all streams and the socket when the session ends.

Gate

Class Variables

- gateID: String – Unique identifier for the gate.
- location: String – Position of the gate (e.g., "Entry", "Exit").
- isOpen: boolean – Indicates whether the gate is currently open.
- lastOperated: DateTime – Timestamp of the last gate operation.

Methods

- openGate(): void – Opens the gate to allow a vehicle to pass.
- closeGate(): void – Closes the gate after vehicle passage.
- getStatus(): boolean – Returns whether the gate is open or closed.
- logGateActivity(): void – Records gate operations in the system log.

GROUP TASK & WORKLOAD

PHASE 2 - DESIGN

//Every Saturday @ 1pm

//Communication over Discord

- DESIGN SOFTWARE DOCUMENT
  1. INTRODUCTION
  2. USE CASES / SEQUENCE DIAGRAMS
  3. CLASSES

  - Raymond
  - Vishal
  - Kurt

- PRESENTATION

  - Raymond
  - Vishal
  - Kurt

THANK YOU!

ANY QUESTIONS?

PHASE 2 - DESIGN