**SDD**

Parking Garage Project

Software Design Document

# Revision History

| Date | Revision | Description | Author |
| --- | --- | --- | --- |
| 3/22/2025 | 1.0 | Started adding content to the SDD from SRS. | Kurt, Vishal, Raymond |
| 4/3/2025 | 2.0 | Expanded on content in SDD and added class candidates. | Kurt, Vishal, Raymond |
| 4/4/2025 | 3.0 | Expanded initial class diagram and created more sequence and use case diagrams to cover more in depth. | Kurt, Vishal, Raymond |
| 4/6/2025 | 4.0 | Worked on some class candidates and use cases. | Kurt, Vishal, Raymond |
| 4/9/2025 | 5.0 | Added some missing class candidates and updated class diagrams. | Kurt, Vishal, Raymond |
| 4/23/2025 | 6.0 | Changed some minor details in the system overview. | Kurt, Vishal, Raymond |
| 4/29/2025 | 7.0 | Added some methods for classes and deleted some classes. | Kurt, Vishal, Raymond |
| 5/3/2025 | 8.0 | Added some new classes and details. | Kurt, Vishal, Raymond |
| 5/4/2025 | 9.0 | Added more classes. | Kurt, Vishal, Raymond |

# Table of Contents

[Revision History 2](#_ffcjvve8rkh1)

[Table of Contents 3](#_cl1tmj2disvg)

[● Introduction 5](#_5el1inidzkoa)

[● System Overview 6](#_7bkvluik8pns)

[Entry Module: 6](#_tid45t9733vg)

[● Automated Fee Calculation: 7](#_hzdri0dnyys5)

[● Issuing Penalties: 7](#_myxq891qx1ts)

[● Types of Logged Data: 7](#_dxu6yhao7lh)

[● Data Accessibility: 8](#_yzxwy26gulbp)

[● User Management: 8](#_iruk1zl78nso)

[● System Configuration: 8](#_gn0nqcdwsurc)

[● Maintenance Alerts: 8](#_pkt9se0aatu)

[● Multi-Garage Management: 8](#_2zhh9l20051c)

[● Use Cases 9](#_yf1v1k3wn70w)

[1. Use Case: Login and Logout 9](#_yoxt4wq97c99)

[2. Use Case: Ticket Handling 9](#_z891yjif5shp)

[3. Use Case: Handling Parking Data 10](#_xx4u9rbq1r0m)

[4. Use Case: Payment Processing 10](#_lpslftu354v7)

[5. Use Case: Automatic vs. Manual Payment Handling 11](#_xppx7ykbgzqe)

[6. Use Case: Add Parking Levels 11](#_migiv033f58l)

[UML Use Case Diagrams Document 13](#_di1b5bmitpy4)

[1. Use Case: Login and Logout 13](#_9ipg5hxlouoe)

[2. Use Case: Ticket Handling 13](#_3qzgzd8m47zm)

[3. Use Case: Handling Parking Data 14](#_1j1wt3fyo6e)

[4. Use Case: Payment Processing 14](#_gynrsha139tv)

[5. Use Case: Automatic vs. Manual Payment Handling 15](#_si2md9gimctk)

[6. Use Case: Add Parking Levels 15](#_qkn8ww60v9f6)

[Class Diagrams 16](#_z43vm5817sbq)

[Class Diagram: 16](#_jh0667uqctlr)

[Sequence Diagrams 17](#_xbho7s906lrv)

[Sequence Diagram - Employee Side 18](#_f3p3yce0r79p)

[Sequence Diagram for Use Case 1: Login and Logout 19](#_yl9s42336mcj)

[Sequence Diagram for Use Case 2: Ticket Handling 20](#_ndm7fnsj8su5)

[Sequence Diagram for Use Case 3: Handling Parking Data 21](#_6jj6ytnmj7r7)

[Sequence Diagram for Use Case 4: Payment Processing 22](#_g8vqhob4fxw4)

[Sequence Diagram for Use Case 5: Automatic vs. Manual Payment Handling 23](#_57i8ulgzpvei)

[Sequence Diagram for Use Case 6: Add Parking Levels 24](#_8xxrccay5tw)

[● Classes and Objects 25](#_w0dchs3e9l74)

[ParkingGarage implements Serializable 25](#_ws05y23n9xup)

[ParkingLevel implements Serializable 26](#_mtxjf3p7thur)

[ParkingSpace implements Serializable 27](#_uzgvotsu8gkl)

[EntranceDisplayBoard implements Serializable 27](#_scs6cpm9ippv)

[EntryKiosk implements Serializable 28](#_ucq5ewld7g6h)

[ExitKiosk implements Serializable 28](#_vgxfx62k0tx)

[Payment implements Serializable 30](#_k4zy2ovy924n)

[Employee (Interface) 31](#_dlc9u9il23wx)

[ParkingAttendant extends User, implements Serializable 31](#_k08pizpl8oau)

[Admin implements Employee, Serializable 31](#_8xu4di64kya7)

[Customer 32](#_7dc2bs4t29e0)

[SystemLog 33](#_arqatyf5rcnx)

[Hardware 33](#_c3notclzi5eb)

[Gate implements Serializable 34](#_farhiouiet76)

[BuildParkingGarage implements CreateGarage, Ticketing, GarageEmployee 34](#_a211n41k6du5)

[CreateGarage (Interface) 35](#_hqs06cb4qbof)

[ParkingGarageSystem (Abstract Superclass) 35](#_j2of8z5nscj0)

[UpdateGarage (Interface) 36](#_5jkt1a99zncd)

[Driver (class to initialize garages) 36](#_x50cspxd255x)

[FixModel 36](#_2c9899skd9u5)

[FixScale 37](#_tpfrg53aqh66)

[ParkingExceptions 37](#_aqd3ndb9g0c3)

[Address implements Serializable 38](#_9ojtcaqlbof0)

[LevelDisplayBoard implements Serializable 38](#_qqb6cnpx62r0)

[EntryKioskClientGUI (Entry point to the project) 38](#_a6dlm2h8vqch)

[ExitKioskClientGUI (Exit point of the project) 39](#_7sywug4avjfq)

[ParkingGarageServer 39](#_piuwb45qi16c)

[ClientHandler implements Runnable 40](#_4c11ifhcbg36)

[FileIO implements Serializable 40](#_ph3to8xoz6y6)

[AccessEmployeeControl 41](#_nazwo2mlaolk)

[AuthenticationManager 41](#_3rhbdp515r8n)

[User (Abstract Superclass) 42](#_8hn7efrduhds)

[PaymentFrame 42](#_nd41zxxl3k88)

[Ticketing (Interface) 42](#_a5djcycd6grb)

[GarageEmployee (Interface) 43](#_mv1kdat8urs6)

[FixAdapter 43](#_fafae7y78vfv)

# Introduction

1.1. Overview

The parking garage software delivers administrative logging reports and a graphical user interface (GUI) for users. The software will be integrated to hardware components such as automated payment kiosks, tracking monitors/sensors, hardware components for entering/exiting barriers and as well as parking garage’s work tablets.

1.2. Purpose

The parking garage software is intended 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.

1.3 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.

1.4 Target Audience

1.4.1. Investors seeking to bring the parking garage system

1.4.2. Potential business associates such as hardware solution manufacturers and parking resources.

1.4.3. Neighboring businesses

1.4.4. Entertainment venues, sports stadiums, and large event spaces

# System Overview

2.1. Software System’s functionality

The Vehicle Entry & Exit Management module is responsible for handling vehicle movement within the parking facility. It ensures smooth operations at entry and exit points using automated ticketing and fee calculation mechanisms.

#### Entry Module:

* Upon arrival, the system detects an incoming vehicle using RFID scanners, or 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
* The barrier gate automatically opens for the vehicle to enter after a ticket has been issued.

Exit Module:

When a vehicle approaches the exit, the dispenser takes back the ticket to scan internally.

* 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 digital or printed receipt.
  + Opens the exit barrier for the vehicle to leave.
* If a vehicle exits without a valid ticket, an employee override option is available to manually process the exit.

2[.1.1. Vehicle Entry & Exit Management](#)

* Entry module: Issuing/Generating a ticket and prints a ticket containing the start time, parking location, and a unique id.
* Exit module: Calculates fees and processes payments.

2[.1.2. Parking Space Management](#)

* Real-time Availability Tracking: Monitoring guest entering and exiting the parking facility, we are able to provide the status of available parking.
* The system updates availability status dynamically and displays real-time capacity on an electronic board placed near the entrance.
* Parking zones can belong in different categories (general parking, reserved spots, electric vehicle charging stations, handicapped spots) for better customer satisfaction.
* If a discrepancy is detected (e.g., a vehicle leaves but the space is still marked as occupied), an alert is sent for manual verification.

2[.1.3. 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.

2[.1.4. Logging and Garage Report](#)s

The system continuously logs transactions, occupancy data, and security events to provide detailed insights for management.

#### 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 in multiple formats (PDF, CSV, Excel).
  + System logs are archived periodically for auditing and compliance.

2[.1.5. Administration and Maintenance](#)

This module provides administrative control over system settings, configurations, and maintenance operations.

#### 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.

# Use Cases

## **1. Use Case: Login and Logout**

Use Case ID: UC-01  
Relevant Requirements: Employee Login, Graphical User Interface  
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

## **2. Use Case: Ticket Handling**

Use Case ID: UC-02  
Relevant Requirements: Ticketing System, Parking Space Tracking  
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

## **3. Use Case: Handling Parking Data**

Use Case ID: UC-03  
Relevant Requirements: Parking Space Tracking, Multi-Garage Management  
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

## **4. Use Case: Payment Processing**

Use Case ID: UC-04  
Relevant Requirements: Payment Method, Parking Duration  
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

## **5. Use Case: Automatic vs. Manual Payment Handling**

Use Case ID: UC-05  
Relevant Requirements: Employee Charging, Payment Processing  
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

## **6. Use Case: Add Parking Levels**

Use Case ID: UC-06  
Relevant Requirements:  
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.

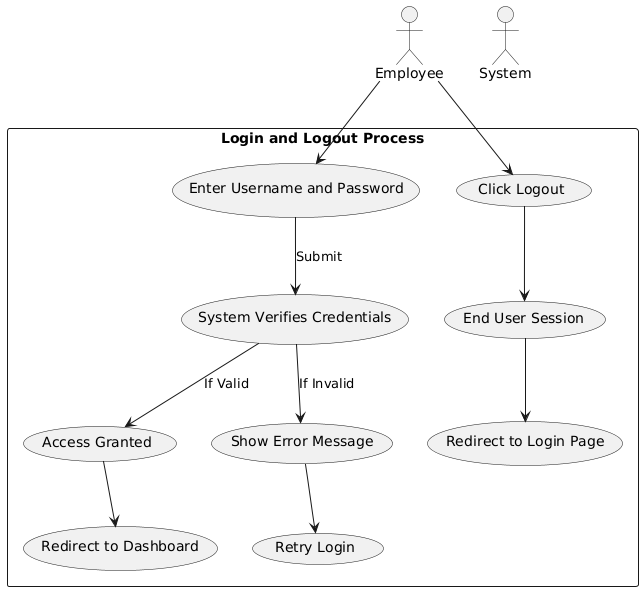
1. Admin selects "Add Parking Spaces" from the system dashboard.
2. System prompts for:
   * Existing Level ID
   * Number of spaces to add
   * Space-specific details (e.g., space size, type – compact, large, electric)
3. Admin enters the information.
4. Admin clicks "Submit".
5. System validates that the level exists and input is correct.
6. System updates the level with the new spaces.
7. Confirmation is shown.

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.

# UML Use Case Diagrams Document

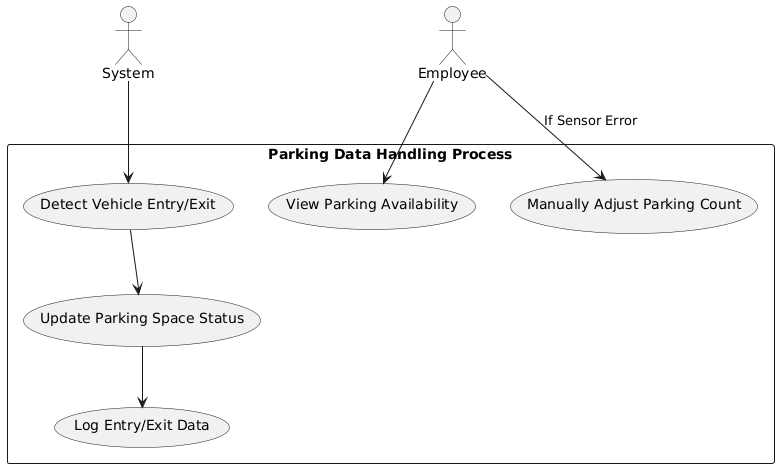
## **1. Use Case: Login and Logout**



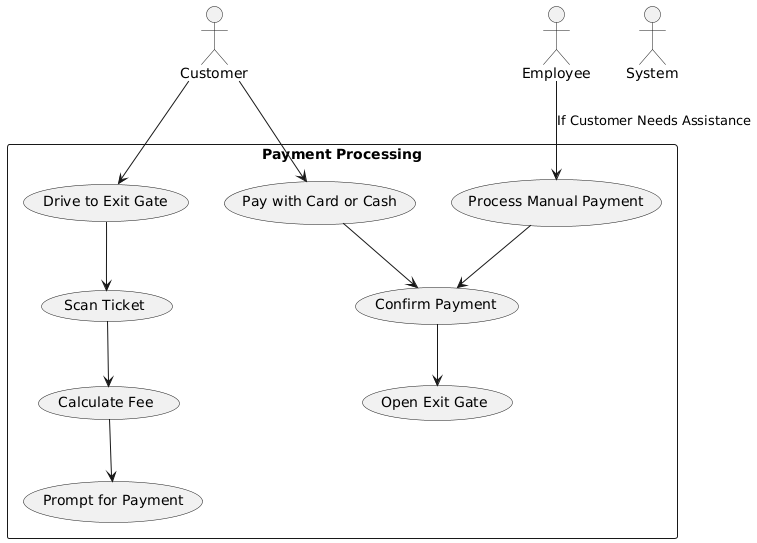
## **2. Use Case: Ticket Handling**

## 

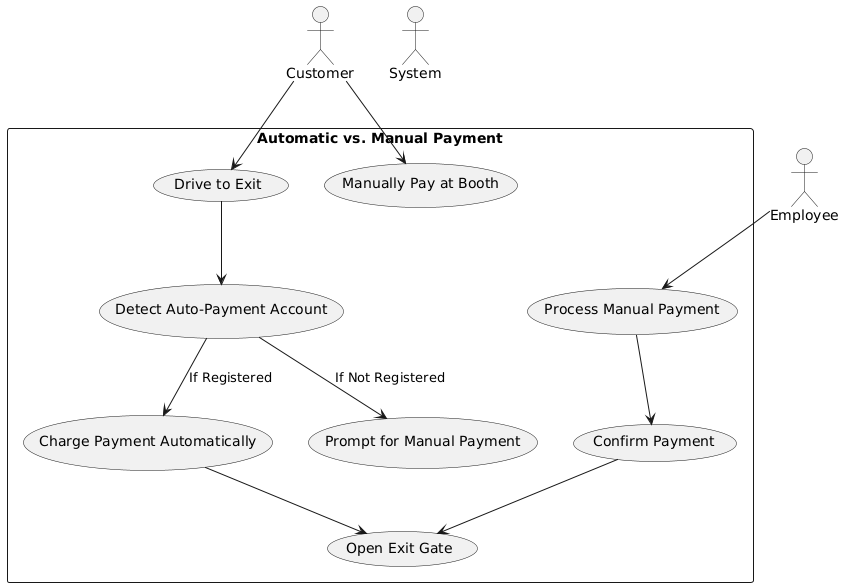
## **3. Use Case: Handling Parking Data**



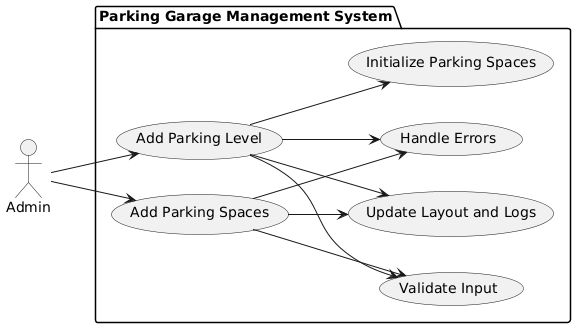
## **4. Use Case: Payment Processing**



## **5. Use Case: Automatic vs. Manual Payment Handling**

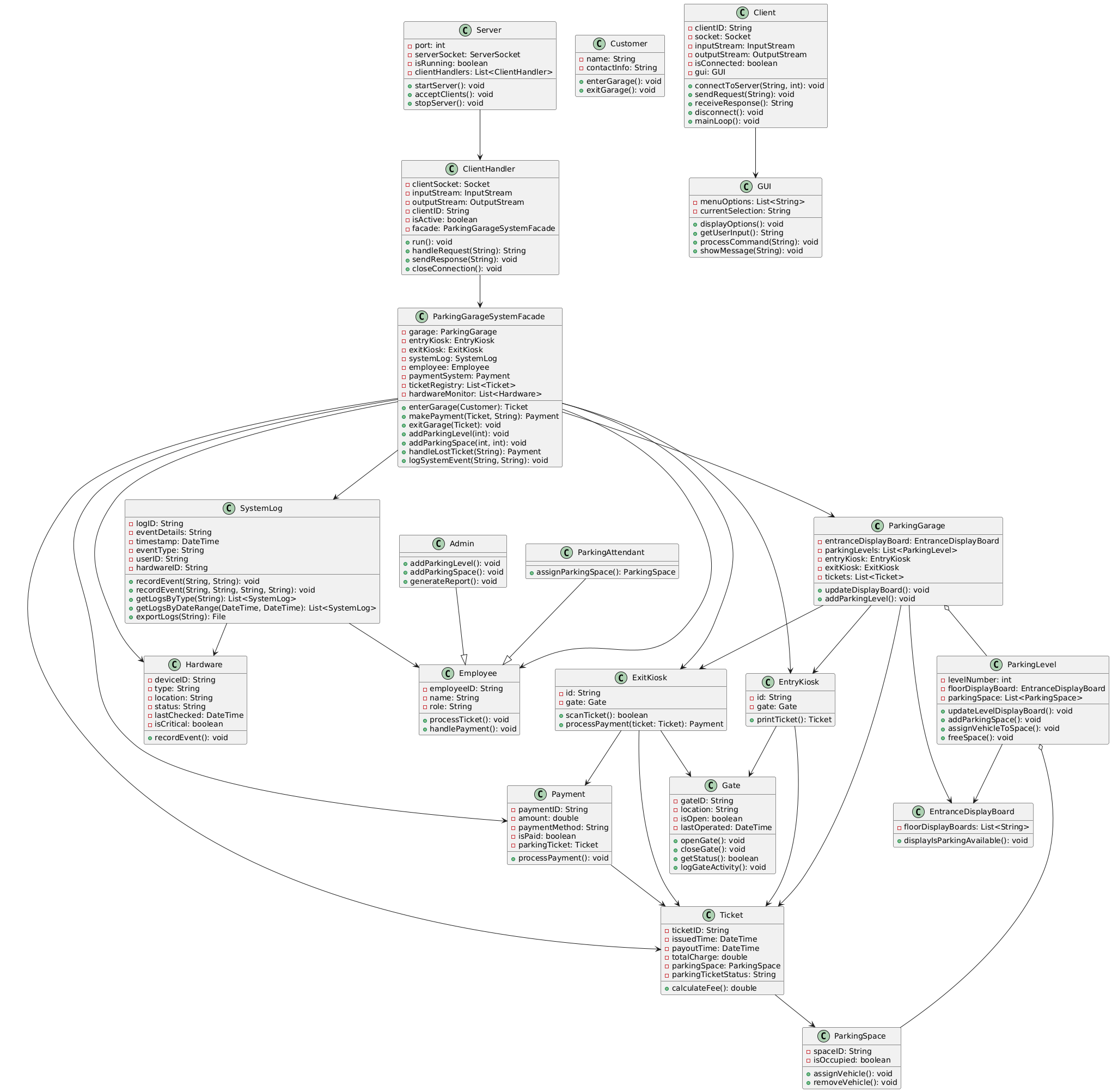


## **6. Use Case: Add Parking Levels**

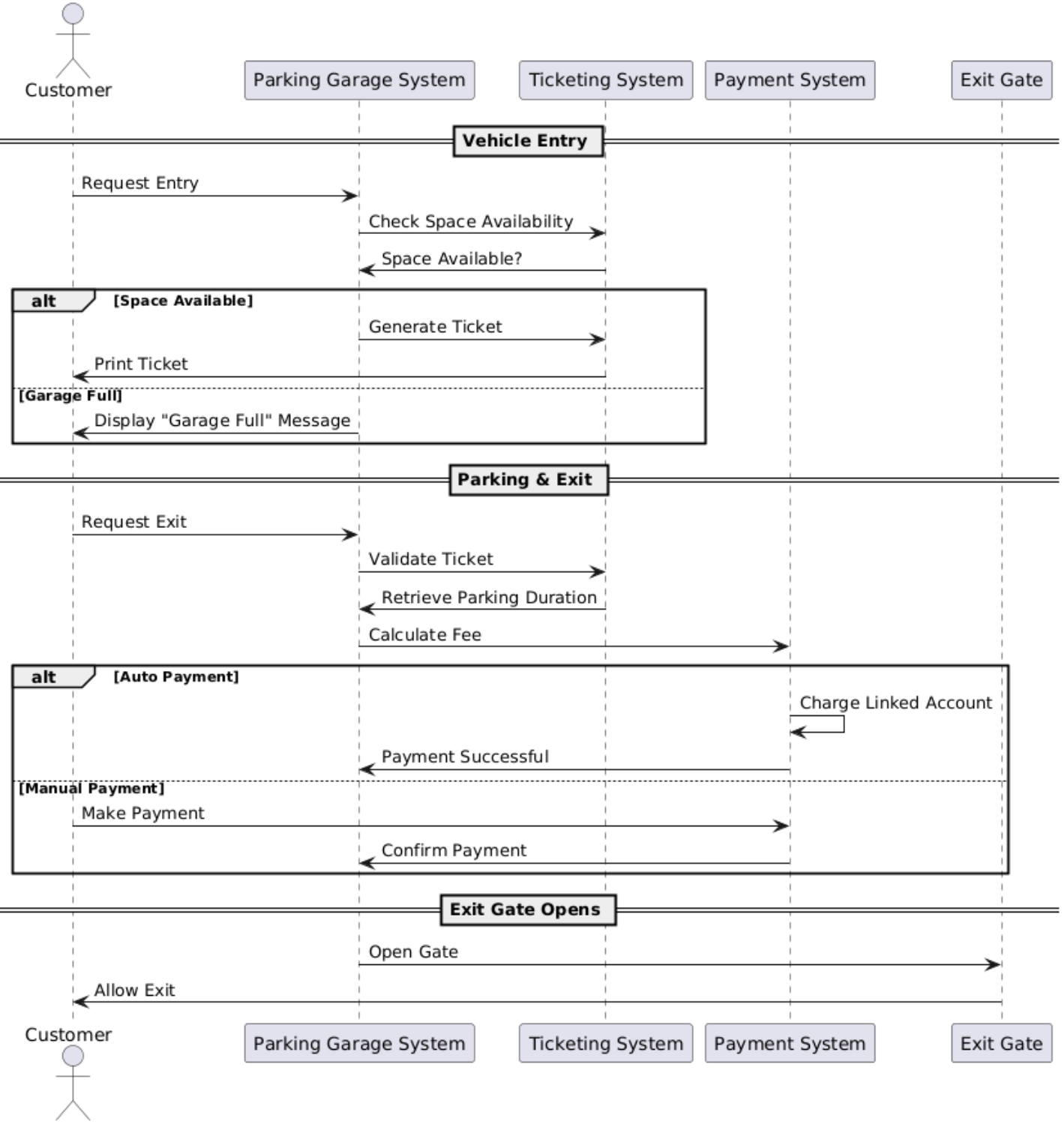


# Class Diagrams

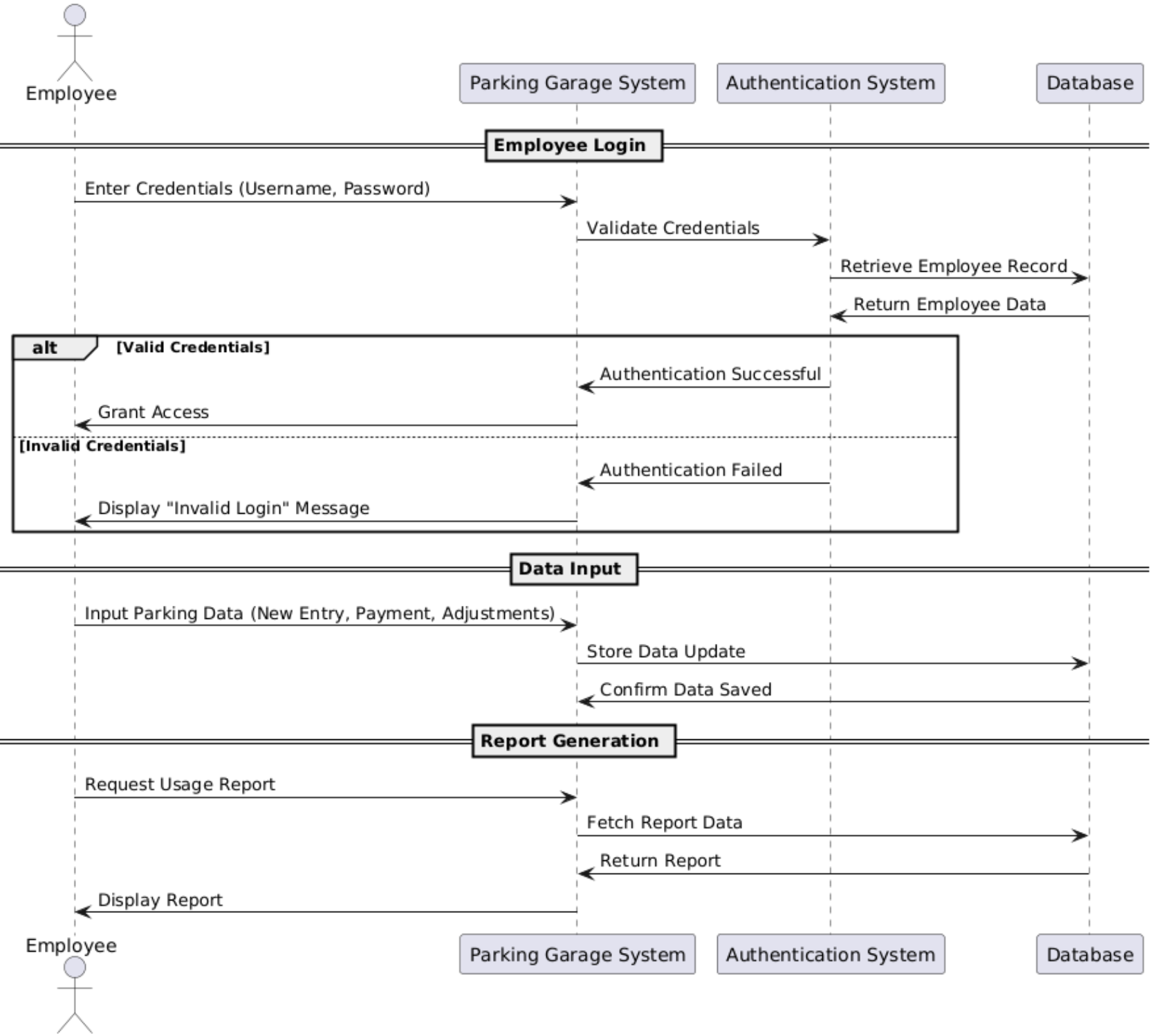
### **Class Diagram:**



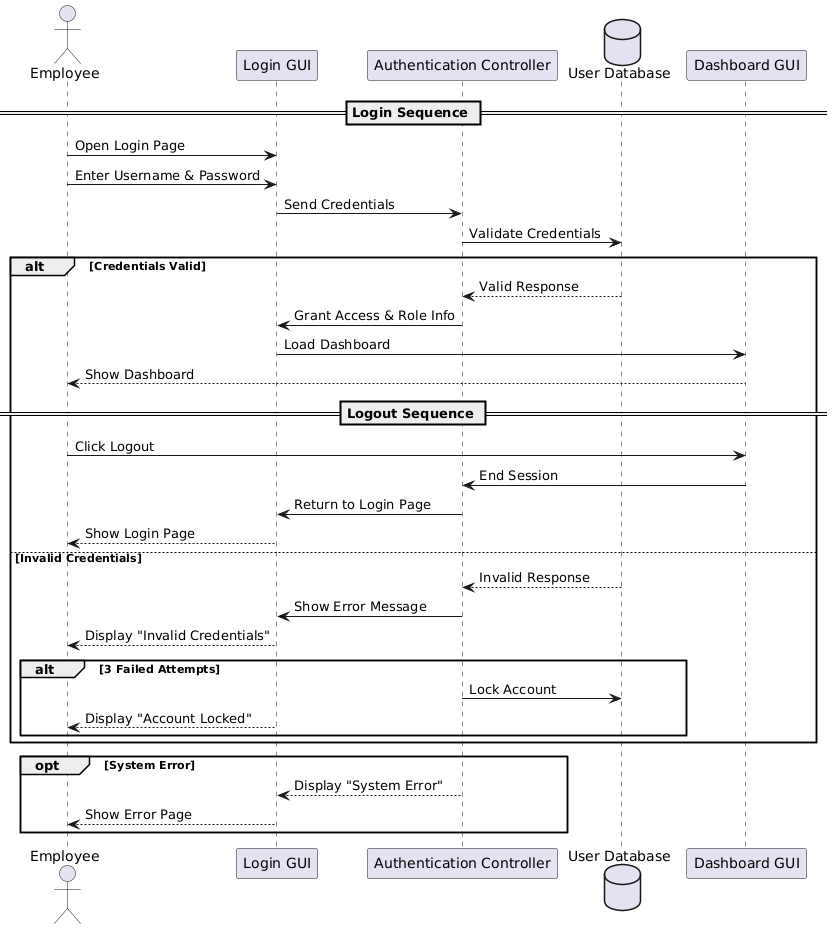
# Sequence Diagrams



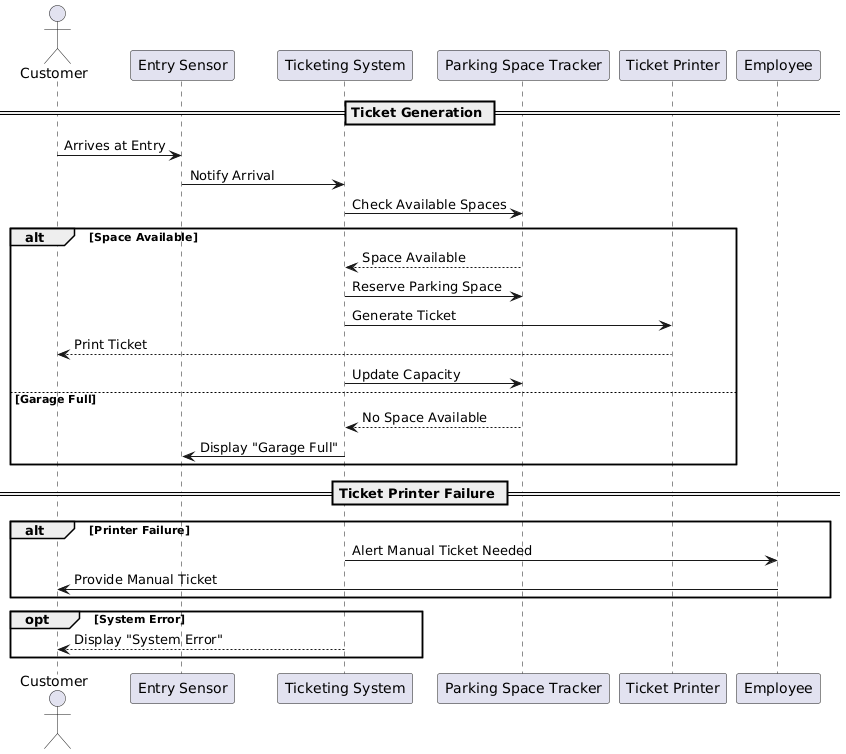
## **Sequence Diagram - Employee Side**



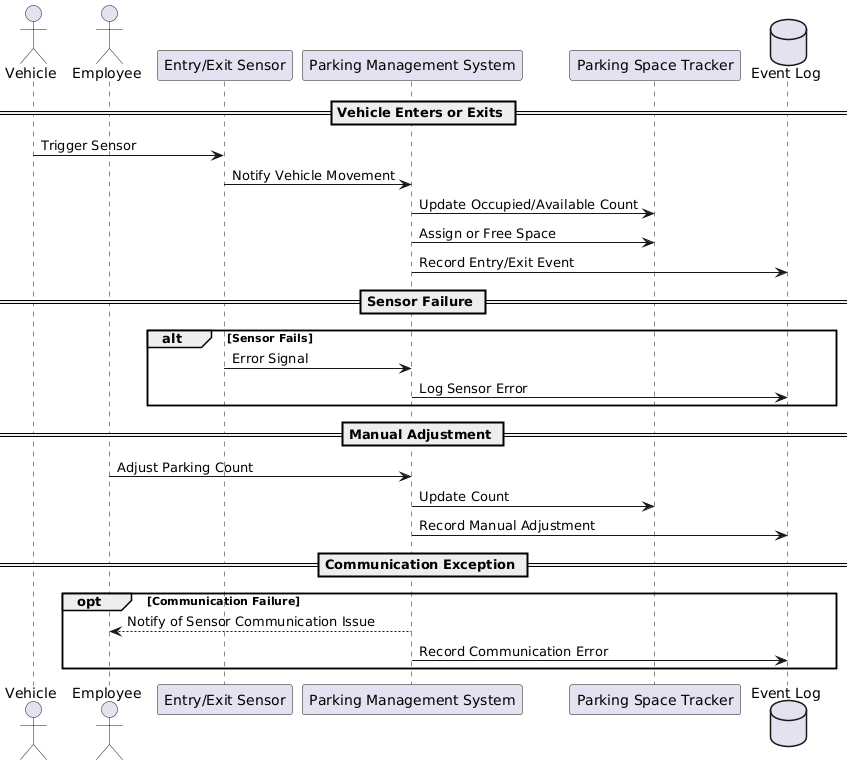
### **Sequence Diagram for Use Case 1: Login and Logout**



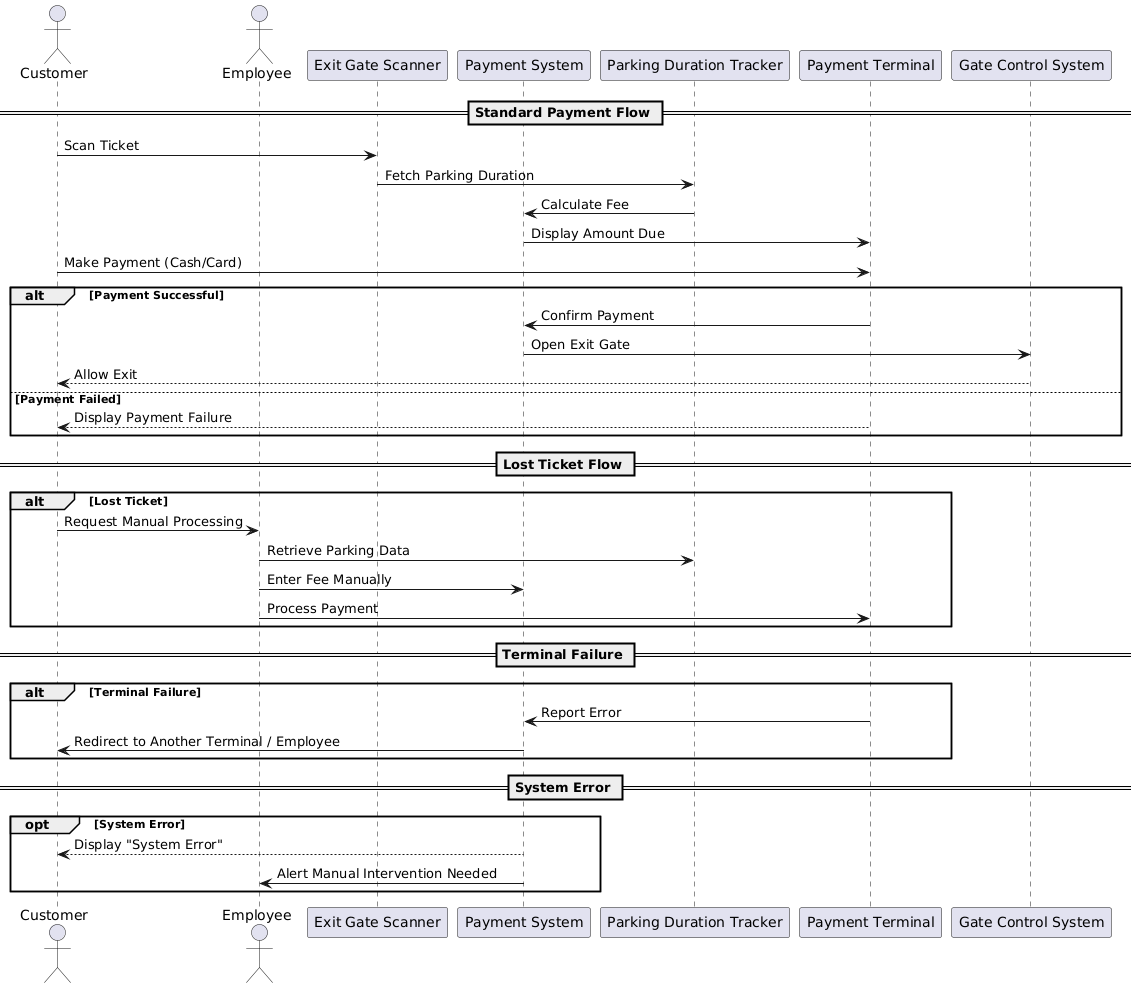
### **Sequence Diagram for Use Case 2: Ticket Handling**



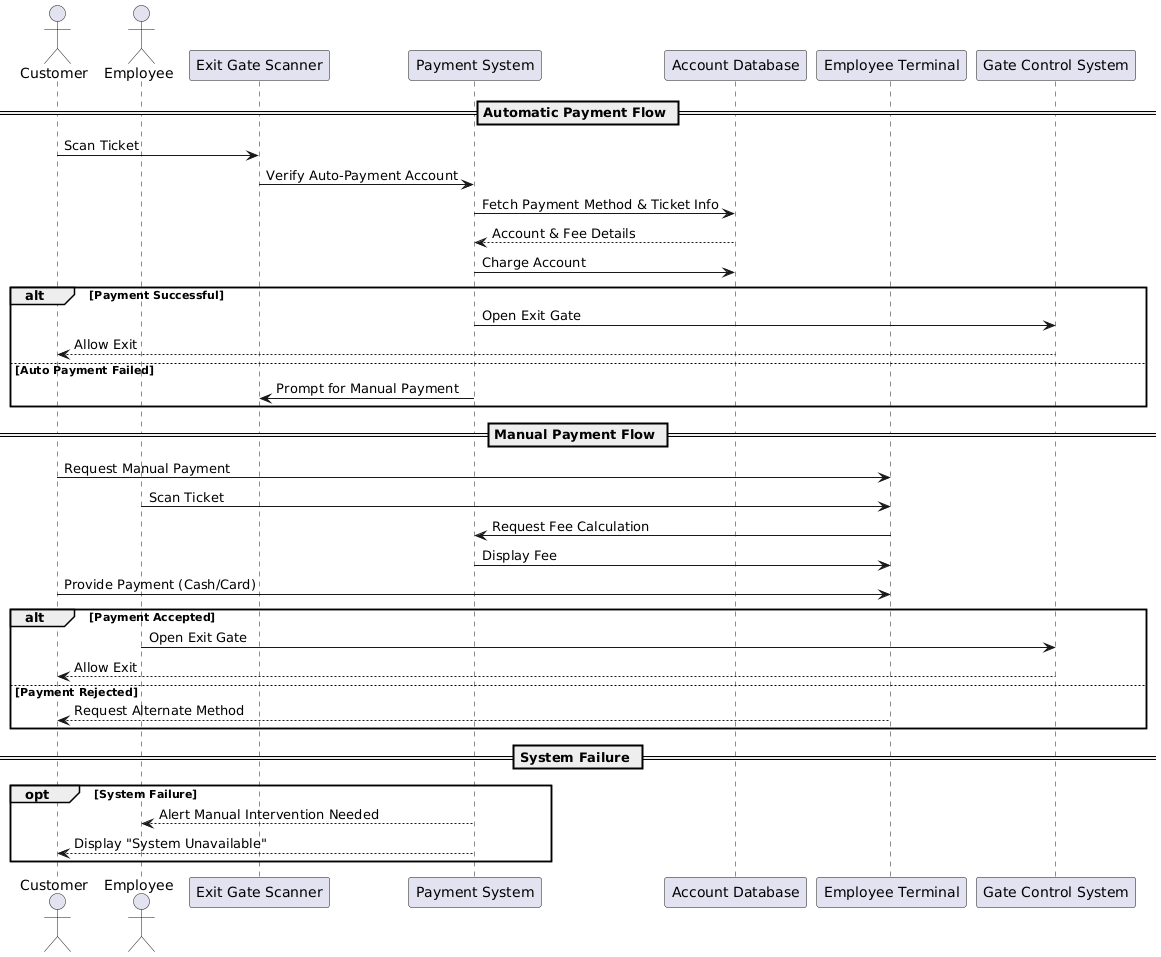
### **Sequence Diagram for Use Case 3: Handling Parking Data**



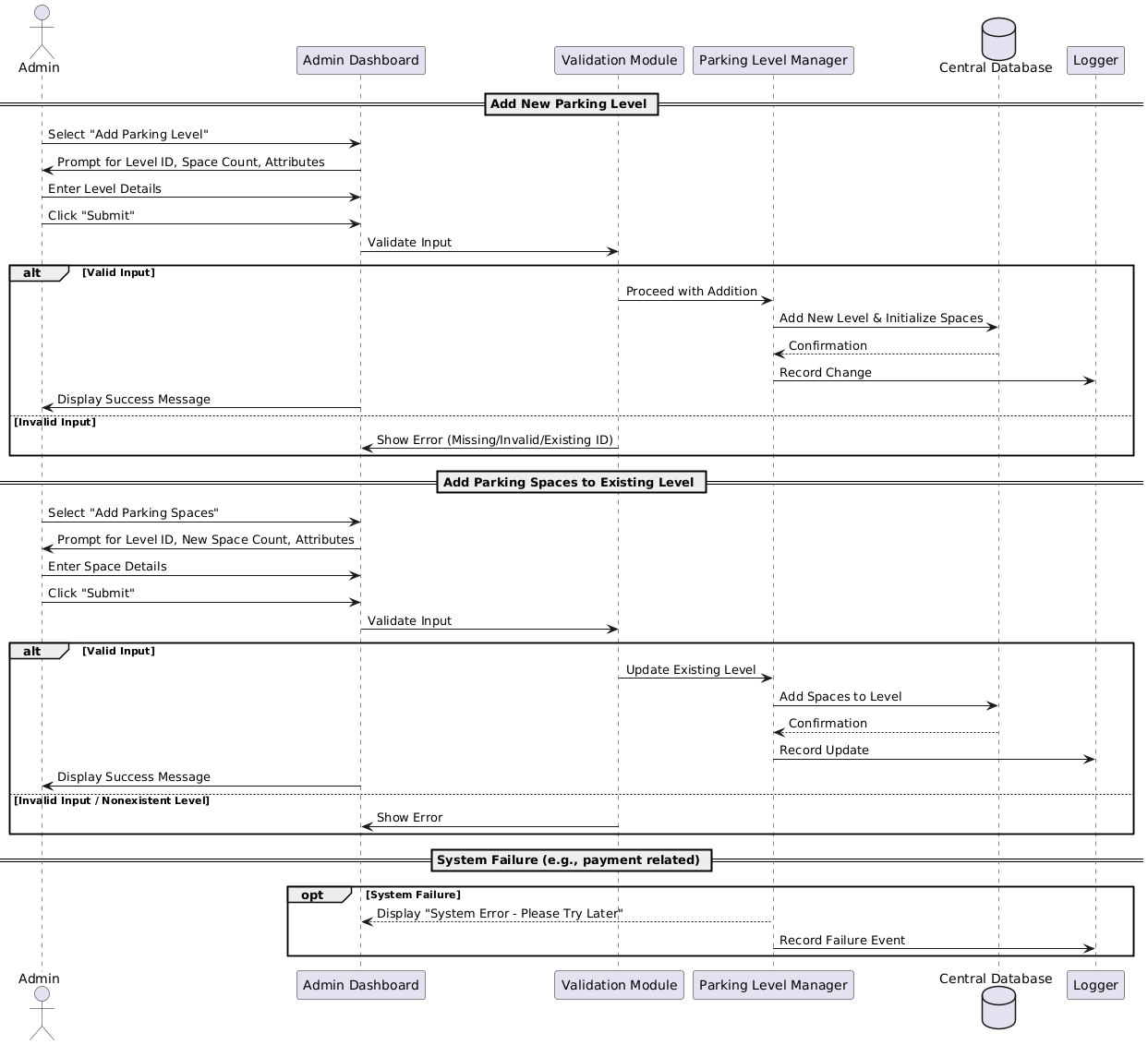
### **Sequence Diagram for Use Case 4: Payment Processing**



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



### **Sequence Diagram for Use Case 6: Add Parking Levels**



# Classes and Objects

#### 

#### ParkingGarage implements Serializable

Class Variables

* numAvailable: int – Tracks the number of available parking spaces in the garage
* address: Address – The location of the garage
* numLevels: int – Number of parking levels in the garage
* parkingLevels: ArrayList<ParkingLevel> – List of all levels and their parking spaces
* entranceBoard: EntranceDisplayBoard – Display showing garage-wide availability
* entryKiosk: EntryKiosk – Entry kiosk that prints tickets and controls the entrance gate
* exitKiosk: ExitKiosk – Exit kiosk that scans tickets and controls the exit gate

Methods:

* ParkingGarage(numLevels: int, numSpacesPerLevel: int): void – Constructor that initializes levels, display, kiosks, and availability
* getAddress(): Address – Returns the garage address
* getNumLevels(): int – Returns the number of parking levels
* getParkingLevels(): ArrayList<ParkingLevel> – Returns the list of parking levels
* getEntranceBoard(): EntranceDisplayBoard – Returns the entrance display board
* getEntryKiosk(): EntryKiosk – Returns the entry kiosk object
* getExitKiosk(): ExitKiosk – Returns the exit kiosk object
* setAddress(address: Address): void – Sets the garage address
* setNumLevels(numLevels: int): void – Sets the number of parking levels
* setParkingLevels(parkingLevels: ArrayList<ParkingLevel>): void – Sets the list of parking levels
* setEntranceBoard(entranceBoard: EntranceDisplayBoard): void – Sets the entrance display board
* setEntryKiosk(entryKiosk: EntryKiosk): void – Sets the entry kiosk
* setExitKiosk(exitKiosk: ExitKiosk): void – Sets the exit kiosk
* addParkingLevel(numSpacesPerLevel: int): void – Adds a new parking level with the given number of spaces
* updateParkingSpace(parkingLevel: int, parkingSpace: int, spaceToOccupied: boolean): void – (Placeholder) Updates a specific space’s occupancy state
* enterParkingGarage(): Ticket – Assigns the first available parking space and returns a new ticket
* decrementAvailablity(): void – Decreases available space count and updates the display
* incrementAvailability(): void – Increases available space count and updates the display
* toString(): String – Returns a formatted string representation of the garage and its levels

#### ParkingLevel implements Serializable

Class Variables

* count: int – Static counter to assign unique internal IDs to each level
* id: int – Unique ID for this parking level instance
* levelNumber: int – Logical identifier (e.g., level 1, level 2, etc.)
* num\_spaces: int – Total number of parking spaces on this level
* parkingSpaces: ArrayList<ParkingSpace> – List of all parking spaces on this level
* floorDisplayBoard: LevelDisplayBoard – Board displaying the number of free spaces on this level

Methods

* ParkingLevel(levelNumber: int, num\_spaces: int): void – Constructor that initializes all parking spaces and display board for the level
* updateDisplayBoard(): void – Calls the display board to show current availability
* getId(): int – Returns the internal unique ID of this level
* getLevelNumber(): int – Returns the logical level number
* getFloorDisplayBoard(): LevelDisplayBoard – Returns the level’s display board
* getParkingSpaces(): ArrayList<ParkingSpace> – Returns the list of spaces on the level
* getParkingSpaceByNum(spaceNum: int): ParkingSpace – Retrieves a space by its assigned number
* getParkingSpace(\_space: ParkingSpace): ParkingSpace – Retrieves a matching space by reference
* occupyAvailableSpace(): ParkingSpace – Returns and marks the first unoccupied space as occupied
* setLevelNumber(levelNumber: int): void – Sets the level number after validating it
* setFloorDisplayBoard(floorDisplayBoard: LevelDisplayBoard): void – Sets the display board for the level
* addParkingSpace(): void – Adds a new parking space and updates the display board
* freeSpace(spaceNumber: int): void – Frees a space by number and updates display
* freeSpace(\_space: ParkingSpace): void – Frees a specific space by reference and updates display
* toString(): String – Returns a formatted string showing all level details and spaces

#### ParkingSpace implements Serializable

Class Variables

* count: int – Static counter to assign unique IDs to all spaces
* id: int – Unique internal identifier for each parking space
* space\_number: int – The number of the space within the level
* isOccupied: boolean – Flag indicating whether the space is currently occupied

Methods

* ParkingSpace(space\_number: int): void – Constructor that initializes the space with a number and sets it as free
* getId(): int – Returns the unique ID of the space
* getCount(): int – Returns the total number of spaces created (static count)
* getSpace\_number(): int – Returns the space number within the level
* setSpace\_number(space\_number: int): void – Validates and sets the space number
* isOccupied(): boolean – Returns true if the space is currently occupied
* setOccupiedSpace(): void – Marks the space as occupied
* setFreeSpace(): void – Frees the space (marks it as unoccupied)
* toString(): String – Returns a formatted string showing the space number and its status
* equals(otherSpace: Object): boolean – Compares two ParkingSpace objects for equality based on ID and space number

#### EntranceDisplayBoard implements Serializable

Class Variables

* isParkingAvailable: boolean – Indicates whether there are open parking spaces available in the garage

Methods

* EntranceDisplayBoard(): void – Constructor that initializes the board as available
* validateCapacity(numAvailable: int): void – Sets availability based on the number of open parking spaces
* displayIsParkingAvailable(): void – Prints a message to the console indicating parking availability

#### EntryKiosk implements Serializable

Class Variables

* serialVersionUID: long – Standard identifier for serialization
* id: String – Unique ID of the entry kiosk (auto-incremented)
* count: int – Tracks how many kiosks have been created (used for ID generation)
* gate: Gate – The gate controlled by this kiosk
* parkingGarage: ParkingGarage – The garage this kiosk is associated with (1-to-1 relationship)

Methods

* EntryKiosk(): void – Constructor that creates a kiosk with a new gate and auto-generated ID
* EntryKiosk(gate: Gate): void – Constructor that creates a kiosk with a custom gate
* printTicket(): Ticket – Generates and returns a new parking ticket
* getId(): String – Returns the kiosk’s ID
* setId(id: String): void – Sets the kiosk’s ID
* getGate(): Gate – Returns the gate object
* setGate(gate: Gate): void – Sets the gate object
* openGate(): void – Opens the gate
* closeGate(): void – Closes the gate
* toString(): String – Returns the kiosk’s ID as a string
* getParkingGarage(): ParkingGarage – Returns the garage associated with this kiosk
* setParkingGarage(parkingGarage: ParkingGarage): void – Sets the associated garage

#### ExitKiosk implements Serializable

Class Variables

* serialVersionUID: long – Standard identifier for serialization
* id: String – Unique ID of the exit kiosk (auto-incremented)
* count: int – Tracks how many kiosks have been created (used for ID generation)
* gate: Gate – The gate controlled by this exit kiosk
* parkingGarage: ParkingGarage – The garage this kiosk is associated with (1-to-1 relationship)

Methods

* ExitKiosk(): void – Constructor that creates a kiosk with a new gate and auto-generated ID
* ExitKiosk(gate: Gate): void – Constructor that creates a kiosk with a custom gate
* scanTicket(ticket: Ticket): boolean – Verifies if the ticket is valid and active; returns true if scanned successfully
* getId(): String – Returns the kiosk’s ID
* setId(id: String): void – Sets the kiosk’s ID
* getGate(): Gate – Returns the gate object
* setGate(gate: Gate): void – Sets the gate object
* openGate(): void – Opens the gate
* closeGate(): void – Closes the gate
* toString(): String – Returns the kiosk’s ID as a string
* getParkingGarage(): ParkingGarage – Returns the garage associated with this kiosk
* setParkingGarage(parkingGarage: ParkingGarage): void – Sets the associated garag

Ticket implements Serializable

Class Variables

* ticketID: String – Unique identifier for the ticket
* count: int – Static counter used to auto-generate ticket IDs
* issuedTime: LocalTime – Time when the ticket was issued (vehicle entry)
* payoutTime: LocalTime – Time when the ticket was processed for exit/payment
* totalCharge: double – Final amount charged based on time parked
* parkingTicketStatus: TicketStatus – Enum indicating ticket status (ACTIVE or PAID)

Methods

* Ticket(): void – Constructor that initializes ticket ID, marks status as ACTIVE, and records the entry time
* calculateFee(): double – Computes total charge based on time between issuedTime and payoutTime and updates status to PAID
* getTicketID(): String – Returns the ticket ID
* setTicketID(ticketID: String): void – Sets the ticket ID
* getEntryTime(): LocalTime – Returns the time the ticket was issued
* setIssuedTime(issuedTime: LocalTime): void – Sets the ticket issue time
* getPayoutTime(): LocalTime – Returns the time when payment was made
* setPayoutTime(payoutTime: LocalTime): void – Sets the payout time
* getTotalCharge(): double – Returns the total amount charged
* setTotalCharge(totalCharge: double): void – Sets the charge amount
* getParkingTicketStatus(): TicketStatus – Returns the current status of the ticket
* setParkingTicketStatus(parkingTicketStatus: TicketStatus): void – Updates the status of the ticket

#### Payment implements Serializable

Class Variables

* paymentID: String – Unique identifier for the payment transaction
* count: int – Static counter used to generate unique payment IDs
* amount: double – Amount paid for the ticket
* paymentMethod: PaymentMethodStatus – Enum specifying the method of payment (CARD, CASH, or UNDEFINED)
* isPaid: boolean – Flag indicating whether the payment was successfully completed
* parkingTicket: Ticket – The ticket this payment is associated with

Methods

* Payment(parkingTicket: Ticket): void – Constructor that initializes a new payment linked to a ticket
* processPayment(paymentMethod: PaymentMethodStatus): void – Calculates the fee from the ticket, sets the method, and marks the payment as complete
* getPaymentID(): String – Returns the unique ID of the payment
* setPaymentID(paymentID: String): void – Sets the payment ID
* getAmount(): double – Returns the amount paid
* setAmount(amount: double): void – Sets the amount to be paid
* getPaymentMethod(): PaymentMethodStatus – Returns the method of payment used
* setPaymentMethod(paymentMethod: PaymentMethodStatus): void – Sets the payment method
* getIsPaid(): boolean – Returns whether the payment is marked as complete
* setPaid(isPaid: boolean): void – Sets the paid status
* getParkingTicket(): Ticket – Returns the ticket associated with this payment
* setParkingTicket(parkingTicket: Ticket): void – Links a ticket to the payment

#### 

#### Employee (Interface)

Class Variables

* None

Methods:

* getUsername(): String – Returns the employee’s username
* getPassword(): String – Returns the employee’s password
* login(username: String, password: String): boolean – Returns true if the provided credentials match the employee’s login

#### 

#### ParkingAttendant extends User, implements Serializable

Class Variables

* parkingAttendantID: String – Unique identifier for the parking attendant
* count: int – Static counter used to auto-generate unique IDs
* name: String – Full name of the parking attendant
* username: String – Username used for login (defaults to the ID)
* password: String – Password used for authentication (default is "password")

Methods:

* ParkingAttendant(name: String): void – Constructor that creates a new attendant with a name and default credentials
* processTicket(): void – Creates a new ticket manually and prints its ID
* handlePayment(ticket: Ticket): void – Processes payment on a provided ticket if it's valid and active
* login(username: String, password: String): boolean – Returns true if login credentials match
* getParkingAttendantID(): String – Returns the attendant's ID
* setParkingAttendantID(employeeID: String): void – Sets the attendant's ID
* getName(): String – Returns the attendant’s name
* setName(name: String): void – Sets the attendant’s name
* getUsername(): String – Returns the username
* setUsername(username: String): void – Sets the username
* getPassword(): String – Returns the password
* setPassword(password: String): void – Sets the password

#### Admin implements Employee, Serializable

Class Variables

* count: int – Tracks the number of admins created to generate unique IDs
* adminID: String – Unique identifier for the admin
* name: String – Name of the admin
* username: String – Username used for login (defaults to adminID)
* password: String – Password for authentication (default is "admin123")
* levels: List<ParkingLevel> – Stores parking levels added by the admin

Methods

* Admin(name: String): void – Constructor that initializes an admin with a name and auto-generated credentials
* addParkingLevel(levelNum: int, numSpaces: int): void – Adds a new parking level and logs the action
* login(username: String, password: String): boolean – Checks admin credentials for login
* getAdminID(): String – Returns the admin's ID
* getName(): String – Returns the admin's name
* getUsername(): String – Returns the admin's username
* getPassword(): String – Returns the admin's password
* setName(name: String): void – Sets the admin’s name
* setUsername(username: String): void – Sets the admin’s username
* setPassword(password: String): void – Sets the admin’s password

#### Customer

Class Variables:

* name: String – Full name of the customer
* contactInfo: String – Contact information such as phone number or email
* entryTime: LocalDateTime – The timestamp when the customer enters the garage
* exitTime: LocalDateTime – The timestamp when the customer exits the garage

Methods:

* Customer(name: String, contactInfo: String): void – Constructor that creates a new customer with the given name and contact info
* getName(): String – Returns the customer’s name
* getContactInfo(): String – Returns the customer’s contact information
* getEntryTime(): LocalDateTime – Returns the time the customer entered the garage
* getExitTime(): LocalDateTime – Returns the time the customer exited the garage
* enterGarage(): void – Sets and prints the time the customer entered the garage
* exitGarage(): void – Sets and prints the time the customer exited the garage
* formatDateTime(dateTime: LocalDateTime): String – Converts a timestamp to a readable format (private helper method)

#### 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

* Hardware(deviceID: String, type: String, location: String, isCritical: boolean): void – Constructor that initializes the hardware with ID, type, location, and criticality
* getDeviceID(): String – Returns the device ID
* getType(): String – Returns the device type
* getLocation(): String – Returns the device location
* getStatus(): String – Returns the device status
* getLastChecked(): LocalDateTime – Returns the time of last health check
* isCritical(): boolean – Returns whether the device is critical
* setStatus(status: String): void – Sets the device’s status
* updateLastChecked(): void – Updates the last checked time to now
* recordEvent(eventDetails: String, eventType: String): void – Logs an event with the device ID but no user ID
* recordEvent(eventDetails: String, eventType: String, userID: String): void – Logs an event with both user and device IDs
* performHealthCheck(): void – Performs a check on the device status and logs system events based on its condition
* toString(): String – Returns a formatted string representation of the hardware object

#### Gate implements Serializable

Class Variables

* isOpen: boolean – Indicates whether the gate is currently open (true) or closed (false)

Methods

* Gate(): void – Constructor that initializes the gate as closed (isOpen = false)
* openGate(): void – Sets the gate’s state to open (true)
* closeGate(): void – Sets the gate’s state to closed (false)

#### BuildParkingGarage implements CreateGarage, Ticketing, GarageEmployee

Class Variables

* None directly declared; inherits everything from ParkingGarageSystem

Methods

* None directly implemented.

#### CreateGarage (Interface)

Class Variables

* None

Methods

* BuildGarage(Address address, int numParkingLevels, int numSpacesPerLevel): void – Builds a parking garage at the specified address with the given number of levels and parking spaces per level. – Throws: ParkingExceptions, IOException
* printGarage(Address address): void – Prints or displays details about the garage located at the specified address.

#### ParkingGarageSystem (Abstract Superclass)

Class Variables

* parkingGarage: ParkingGarage – The currently accessed or loaded garage object
* pg1: LinkedHashMap<String, ParkingGarage> – Stores parking garages keyed by ZIP code for quick access

Methods

* getGarage(address: Address): ParkingGarage – Returns the garage for the given address, loading it from file if it exists
* updateParkingSpot(location: String, parkingLevel: int, parkingSpace: int): void – Placeholder method to update a specific parking spot
* BuildGarage(address: Address, numParkingLevels: int, numSpacesPerLevel: int): void – Creates a new garage object and saves it to disk
* printGarage(address: Address): void – Prints the garage object associated with the given address

#### UpdateGarage (Interface)

Class Variables

* None

Methods

* updateParkingSpot(location: String, parkingLevel: int, parkingSpace: int): void – Updates the occupied status of a specific parking space on a given level and location

#### Driver (class to initialize garages)

Class Variables

* None

Methods

* main(args: String[]): void – Carries out garage creation and prints garage information using BuildParkingGarage

#### FixModel

Class Variables

* None

Methods

* isValidInteger(test: int): boolean – Checks if an integer is non-negative
* checkdataEntry(valueFloat: String): boolean – Returns true if the given string value is not a valid float
* isFileValid(filename: String): boolean – Checks if a file with the given name exists in the current working directory
* isParkingAvailable(numAvailable: int): boolean – Returns true if parking spaces are available
* validateZipCode(zipCode: int): void – Validates that a ZIP code is exactly 5 digits and numeric; throws an exception if not

#### FixScale

Class Variables

* None

Methods

* isValidTicket(test: Ticket): boolean – Returns true if the given ticket object is not null
* checkTicketScan(valueInt: String): boolean – Returns true if the input string is a valid integer; false if it throws a NumberFormatException

#### ParkingExceptions

Class Variables

* serialVersionUID: long – Standard ID for Java serialization
* errorno: int – Represents a specific error code for categorizing known problems
* errormsg: String – Description of the error to be logged or displayed

Methods

* ParkingExceptions(): void – Default constructor that logs and prints the error
* ParkingExceptions(errormsg: String): void – Logs and prints the error with a custom message
* ParkingExceptions(errorno: int): void – Uses an error number to generate a predefined error message
* ParkingExceptions(errorno: int, errormsg: String): void – Initializes with both an error number and message
* ParkingExceptions(e: CloneNotSupportedException): void – Handles specific exception types
* ParkingExceptions(e: Exception): void – Handles any generic exception
* getErrorno(): int – Returns the current error number
* getErrormsg(): String – Returns the current error message
* setErrorno(errorno: int): void – Sets the error number
* setErrormsg(errormsg: String): void – Sets the error message
* printmyproblem(): void – Prints the error number and message to the console
* writemyproblem(): String – Returns the formatted error message string
* logException(): void – Appends the current error to a log file with a timestamp
* fix(errno: int): void – Assigns a predefined message based on the error code

#### Address implements Serializable

Class Variables

* street: String – The street portion of the address
* city: String – The city in which the garage is located
* state: String – The state in which the garage is located
* zipcode: String – The postal ZIP code for the garage

Methods

* getStreet(): String – Returns the street
* getCity(): String – Returns the city
* getState(): String – Returns the state
* getZipcode(): String – Returns the ZIP code
* setStreet(street: String): void – Sets the street
* setCity(city: String): void – Sets the city
* setState(state: String): void – Sets the state
* setZipcode(zipcode: String): void – Sets the ZIP code
* toString(): String – Returns a formatted string representation of the address

#### LevelDisplayBoard implements Serializable

Class Variables

* freeSpaces: int – Number of available parking spaces on the level

Methods

* LevelDisplayBoard(freeSpaces: int): void – Constructor that initializes the display board with the number of available spaces
* displayNumFreeSpaces(): void – Prints a message showing how many spaces are free or if the level is full
* getFreeSpaces(): int – Returns the current number of free spaces
* setFreeSpaces(freeSpaces: int): void – Sets the number of available spaces
* occupySpace(): void – Decreases the count of free spaces by one
* leaveSpace(): void – Increases the count of free spaces by one

#### EntryKioskClientGUI (Entry point to the project)

Class Variables

* frame: JFrame – Main window for the GUI
* outputArea: JTextArea – Text area to display ticket information or errors
* ticket: Ticket – The ticket received from the server upon successful entry
* socket: Socket – TCP connection to the server
* out: ObjectOutputStream – Stream to send data to the server
* in: ObjectInputStream – Stream to receive data from the server

Methods

* EntryKioskClientGUI(): void – Constructor that sets up the GUI layout and click listener
* requestEntry(): void – Sends "ENTRY" to the server, receives a ticket, and displays ticket info in the text area
* main(args: String[]): void – Launches the GUI by creating an instance of the class

#### ExitKioskClientGUI (Exit point of the project)

Class Variables

* frame: JFrame – Main GUI window
* label: JLabel – Text label prompting the user for ticket ID
* exitButton: JButton – Button to trigger the exit request
* ticketField: JTextField – Input field where the user enters the ticket ID
* outputArea: JTextArea – Displays the server's response or errors
* panel: JPanel – Container holding all GUI components
* socket: Socket – TCP socket for communicating with the server
* out: ObjectOutputStream – Stream used to send data to the server
* in: ObjectInputStream – Stream used to receive data from the server

Methods

* ExitKioskClientGUI(): void – Constructor that sets up and displays the GUI components
* requestExit(): void – Sends "EXIT" and the entered ticket ID to the server, displays the server's response
* main(args: String[]): void – Launches the GUI application

#### ParkingGarageServer

Class Variables

* PORT: int – Constant port number (12345) the server listens on
* garage: ParkingGarage – The garage instance that handles all operations (shared with client handlers)

Methods

* ParkingGarageServer(garage: ParkingGarage): void – Starts a TCP server socket and spawns a new ClientHandler thread for each incoming connection
* main(args: String[]): void – Loads a serialized ParkingGarage object from file and launches the server

#### ClientHandler implements Runnable

Class Variables

* LOG\_FILE: String – Name of the log file for recording server activity
* nextTicketId: int – Counter to assign sequential ticket IDs (not used in current version, but reserved)
* tickets: Map<Integer, Ticket> – Stores currently active tickets by their ID
* parkingGarage: ParkingGarage – The garage instance passed from the server
* clientSocket: Socket – TCP socket for client communication
* in: ObjectInputStream – Stream to receive objects from client
* out: ObjectOutputStream – Stream to send objects to client

Methods

* ClientHandler(clientSocket: Socket, parkingGarage: ParkingGarage): void – Initializes socket, streams, and assigns the garage
* run(): void – Listens for either "ENTRY" or "EXIT" request and processes accordingly
  + For "ENTRY": Creates a new Ticket, stores it in the map, and sends it to the client
  + For "EXIT": Reads the ticket ID, validates and removes the ticket, calculates the fee, and returns the response to the client

#### FileIO implements Serializable

Class Variables

* serialVersionUID: long – Standard ID for version consistency during serialization
* LOG\_FILE: String – Path to the system log file used for writing timestamped messages

Methods

* writeGarageObject(a1: ParkingGarage): void – Serializes and writes the given garage object to a default path based on its address
* writeGarageObject(a1: ParkingGarage, filename: String): void – Serializes and writes the garage object to a specified file
* readGarageObject(filename: String): ParkingGarage – Reads and deserializes a ParkingGarage object from the given file path
* setGarageFile(filename: String): String – Checks if a file exists and falls back to a known valid file if not
* log(message: String): void – Appends the given message with a timestamp to the system log file

#### AccessEmployeeControl

Class Variables

* authorizedAttendants: HashSet<ParkingAttendant> – Set of attendants allowed to access system functionality
* authorizedAdmins: HashSet<Admin> – Set of admins authorized to modify garage configuration

Methods

* registerAttendants(attendant: ParkingAttendant): void – Adds the given parking attendant to the set of authorized attendants
* isAuthorized(attendant: ParkingAttendant): boolean – Checks if the given attendant is registered
* registerAttendants(admin: Admin): void – Adds the given admin to the set of authorized admins
* isAuthorized(admin: Admin): boolean – Checks if the given admin is registered

#### AuthenticationManager

Class Variables

* users: ArrayList<User> – A list of all registered system users (admins, attendants, etc.)

Methods

* AuthenticationManager(): void – Constructor that initializes the user list
* registerUser(user: User): void – Adds the given user to the system
* login(username: String, password: String): User – Checks credentials and returns the matching user if found; otherwise, returns null

#### User (Abstract Superclass)

Class Variables

* username: String – The unique login identifier for the user
* password: String – The password associated with the user account

Methods

* User(username: String, password: String): void – Constructor that sets the username and password
* checkCredentials(username: String, password: String): boolean – Returns true if the given credentials match the user's
* getUsername(): String – Returns the username

#### PaymentFrame

Class Variables

* balanceDue: double – The total amount the customer owes for parking

Methods

* PaymentFrame(balance: double): void – Constructor that sets the balance due and builds the GUI
* createPaymentUI(): void – Sets up all Swing components for card and cash payment and handles user interactions
* completePayment(frame: JFrame): void – Finalizes the transaction and closes the frame window

#### Ticketing (Interface)

Class Variables

* None

Methods

* issueTicket(): Ticket – Issues and returns a new parking ticket
* validateExit(something1: String): boolean – Validates whether the provided string (likely ticket ID or similar identifier) is allowed to exit
* getActiveTicketIDs(): Set<Integer> – Returns the IDs of currently active (unexpired/unpaid) tickets

#### GarageEmployee (Interface)

Class Variables

* None

Methods

* addParkingAttendent(parkingAttendant: ParkingAttendant): void – Registers a parking attendant in the system
* addAdmin(admin: Admin): void – Registers an admin in the system

#### FixAdapter

Class Variables

* None

Methods

* unAuthorized(aec: AccessEmployeeControl, admin: Admin): boolean – Returns true if the given admin is not authorized according to the access control list.
* isLevelOutOfBounds(level\_trial: int, garage: ParkingGarage): boolean – Checks whether a given level number is outside the valid range of levels for the specified garage.