Parking Garage Software

Design Documentation

Revision History

Date	Revision	Description	Author
09/23/2024	1.0	Initial Version	Aric Adiego Rajvir Kaur Maji Pearson Leslie Scott Zackary Stephens
10/29/2024	1.0.1	Reformatted document to fix TOC	Aric Adiego
10/29/2024	1.1	Updated UML Class Diagram	Aric Adiego
10/29/2024	1.2	Switched references to “assigned” parking spots to “update the count of available spots”	Maji Pearson
10/29/2024	1.3	Fixed errors in UML Class Diagram	Aric Adiego
10/29/2024	1.4	Added sequence diagram, wireframe diagram, and design patterns	Maji Pearson Aricf Adiego Abhriam Manda Zackary Stephens Rajvir Kaur
11/18/2024	1.5	Updates to UML Class Diagram and DataLoader class/abstract class	Aric Adiego

Table of Contents

Parking Garage Software.....1

Revision History.....2

Table of Contents.....3

Use Case Specifications.....4

UML Use Case Diagram.....10

Class Diagram.....11

Sequence Diagram.....17

Design Patterns.....20

Wireframe Design.....22

Use Case Specifications

- The goals of the customers are to generate tickets, process payment, and remove a vehicle.
- The employee will manage the parking spots, authenticate users, generate reports, and process payments.
- The payment system will process payments.
- The Parking garage system will manage the parking spaces, calculate fees, remove a vehicle, authenticate users, and generate reports
- The primary actors include customers, employees, the parking system, and the parking garage system

Use Case ID: UC1
Use Case Name: Ticket generation
Primary Actor: Employee or Customer
Pre-conditions:

- The user must be authenticated.
- The spot for parking needs to be present.

Post-conditions:

- Ticket generation
- The system updates the count of available spots

Basic Flow or Main Scenario:

1. The clients enter the garage.
2. The system checks in real time the parking spot availability.
3. A ticket is generated for the client.
4. The system updates the count of available spots.
5. The client can freely choose any unoccupied parking spot in the garage.
6. The user can view from the GUI the details of the ticket.

Alternate Flows:

- The client is notified if the garage is full in case no parking spots are present.

Exceptions:

- Invalid Ticket Details

Related Use Cases: UC3

Use Case ID: UC2
Use Case Name: Parking Spot Management
Primary Actor: Employee, Customer
Pre-conditions:

- The client is in the parking garage.
- The client has logged in to the system.
- There is an unoccupied parking spot.

Post-conditions:

- The parking system's available spots count is updated accordingly
- A ticket is generated with a timestamp.

Basic Flow or Main Scenario:

1. The clients enter the garage.
2. The system checks in real time the parking spot availability.
3. A ticket is generated for the client.
4. The system updates the count of available spots
5. The client can freely choose any unoccupied spot in the garage.
6. The customer views a printed ticket, which they may use later for exit or payment processing.
7. The user can view from the GUI the details of the ticket.

Alternate Flows:

- There are no spots available
- An error is recorded during ticket generation, prompting the process to restart.

Exceptions:

- System failure

Related Use Cases: UC3 and UC5

Use Case ID: UC3
Use Case Name: Payment processing
Primary Actor: Employee, Customer
Pre-conditions:

- The client wants to remove the car.
- The ticket has all the relevant details, and the parking fee is computed.

Post-conditions:

- The fee is paid.
- The transaction is logged.
- The spot is marked unoccupied.
- The revenue is updated.

Basic Flow or Main Scenario:

1. The clients pay for the tickets using their preferred mode.
2. The fee is computed based on the duration of the vehicle in the parking lot.
3. The payment is processed using the method used to pay for the fee.
4. The status of the parking spot is updated
5. The client receives a payment receipt.

Alternate Flows:

- Lack of funds

Exceptions:

- System failure

Related Use Cases: UC2

Use Case ID: UC4
Use Case Name: Generating reports
Primary Actor: System Admin
Pre-conditions:

- Admin logs to the system
- Monthly data is available.

Post-conditions:

- A comprehensive report is provided.

Basic Flow or Main Scenario:

1. The administrator logs into the system.
2. They generate reports, either a report on the availability of parking spaces or a revenue report.
3. The data is retrieved in the form of a text file.
4. The report is generated.

Alternate Flows:

- No data is selected

Exceptions:

- Corruption of data

Related Use Cases: UC2 and UC5

Use Case ID: UC5
Use Case Name: Removing car from the spot
Primary Actor: Employee, Customer
Pre-conditions:

- There is a valid ticket.

Post-conditions:

- The parking spot count in the garage is updated to reflect the spot as available.
- The client has exited the garage.

Basic Flow or Main Scenario:

1. The client offers a parking ticket to the machine or staff
2. The details of the ticket are retrieved by the system
3. The fees are computed, and the client pays
4. The system updates the count of available spots to reflect the freed space.
5. The client exits the garage.

Alternate Flows:

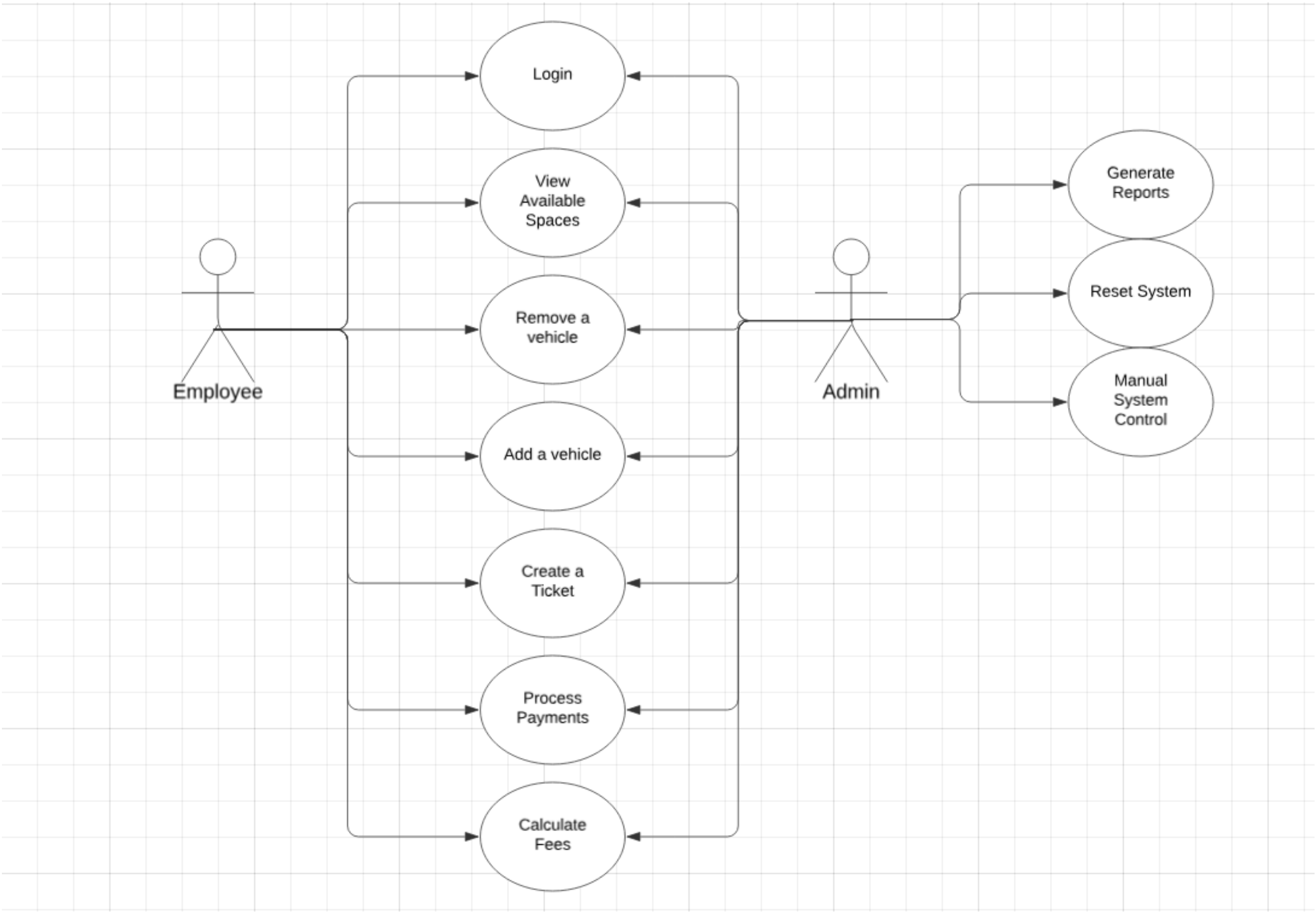
- Lost tickets requiring manual input

Exceptions:

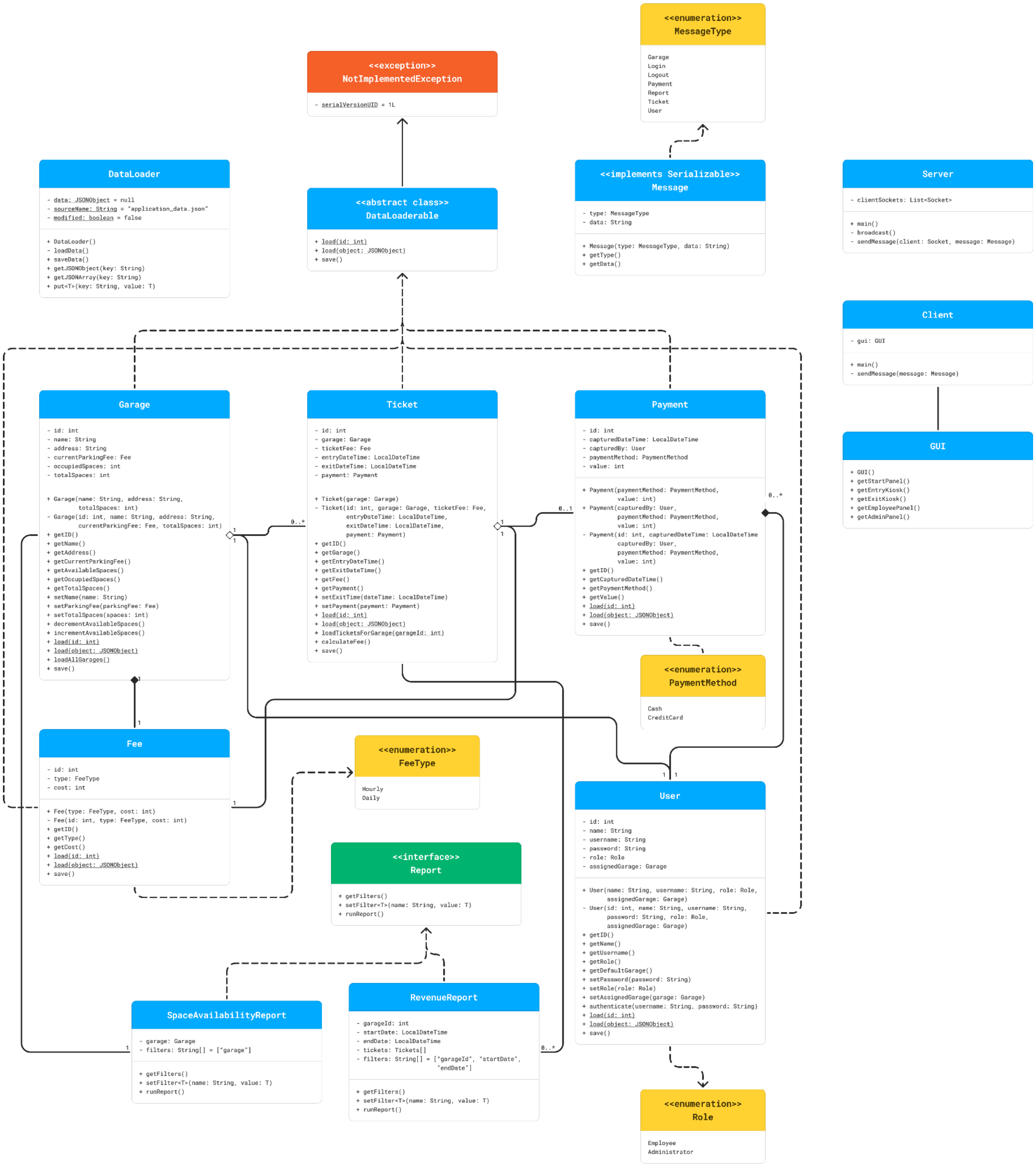
- N/A

Related Use Cases: UC2 and UC4

UML Use Case Diagram



Class Diagram


<<exception>>
NotImplementedException
- serialVersionUID = 1L
<<enumeration>>
MessageType
Garage
Login
Logout
Payment
Report
Ticket
User
DataLoader
- data: JSONObject = null
- sourceName: String = "application_data.json"
- modified: boolean = false
+ DataLoader()
- loadData()
+ saveData()
+ getJSONObject(key: String)
+ getJSONArray(key: String)
+ put<T>(key: String, value: T)
<<abstract class>>
DataLoaderable
+ load(id: int)
+ load(object: JSONObject)
+ save()
<<implements Serializable>>
Message
- type: MessageType
- data: String
+ Message(type: MessageType, data: String)
+ getType()
+ getData()
Server
- clientSockets: List<Socket>
+ main()
- broadcast()
- sendMessage(client: Socket, message: Message)
Client
- gui: GUI
+ main()
- sendMessage(message: Message)
GUI
+ GUI()
+ getStartPanel()
+ getEntryKiosk()
+ getExitKiosk()
+ getEmployeePanel()
+ getAdminPanel()
Garage
- id: int
- name: String
- address: String
- currentParkingFee: Fee
- occupiedSpaces: int
- totalSpaces: int
+ Garage(name: String, address: String, totalSpaces: int)
- Garage(id: int, name: String, address: String, currentParkingFee: Fee, totalSpaces: int)
+ getID()
+ getName()
+ getAddress()
+ getCurrentParkingFee()
+ getAvailableSpaces()
+ getOccupiedSpaces()
+ getTotalSpaces()
+ setName(name: String)
+ setParkingFee(parkingFee: Fee)
+ setTotalSpaces(spaces: int)
+ decrementAvailableSpaces()
+ incrementAvailableSpaces()
+ load(id: int)
+ load(object: JSONObject)
+ loadAllGarages()
+ save()
Ticket
- id: int
- garage: Garage
- ticketFee: Fee
- entryDateTime: LocalDateTime
- exitDateTime: LocalDateTime
- payment: Payment
+ Ticket(garage: Garage)
- Ticket(id: int, garage: Garage, ticketFee: Fee, entryDateTime: LocalDateTime, exitDateTime: LocalDateTime, payment: Payment)
+ getID()
+ getGarage()
+ getEntryDateTime()
+ getExitDateTime()
+ getFee()
+ getPayment()
+ setExitTime(dateTime: LocalDateTime)
+ setPayment(payment: Payment)
+ load(id: int)
+ load(object: JSONObject)
+ loadTicketsForGarage(garageId: int)
+ calculateFee()
+ save()
Payment
- id: int
- capturedDateTime: LocalDateTime
- capturedBy: User
- paymentMethod: PaymentMethod
- value: int
+ Payment(paymentMethod: PaymentMethod, value: int)
+ Payment(capturedBy: User, paymentMethod: PaymentMethod, value: int)
- Payment(id: int, capturedDateTime: LocalDateTime capturedBy: User, paymentMethod: PaymentMethod, value: int)
+ getID()
+ getCapturedDateTime()
+ getPaymentMethod()
+ getValue()
+ load(id: int)
+ load(object: JSONObject)
+ save()
<<enumeration>>
PaymentMethod
Cash
CreditCard
Fee
- id: int
- type: FeeType
- cost: int
+ Fee(type: FeeType, cost: int)
- Fee(id: int, type: FeeType, cost: int)
+ getID()
+ getType()
+ getCost()
+ load(id: int)
+ load(object: JSONObject)
+ save()
<<enumeration>>
FeeType
Hourly
Daily
<<interface>>
Report
+ getFilters()
+ setFilter<T>(name: String, value: T)
+ runReport()
SpaceAvailabilityReport
- garage: Garage
- filters: String[] = ["garage"]
+ getFilters()
+ setFilter<T>(name: String, value: T)
+ runReport()
RevenueReport
- garageId: int
- startDate: LocalDateTime
- endDate: LocalDateTime
- tickets: Tickets[]
- filters: String[] = ["garageId", "startDate", "endDate"]
+ getFilters()
+ setFilter<T>(name: String, value: T)
+ runReport()
User
- id: int
- name: String
- username: String
- password: String
- role: Role
- assignedGarage: Garage
+ User(name: String, username: String, role: Role, assignedGarage: Garage)
- User(id: int, name: String, username: String, password: String, role: Role, assignedGarage: Garage)
+ getID()
+ getName()
+ getUsername()
+ getRole()
+ getDefaultGarage()
+ setPassword(password: String)
+ setRole(role: Role)
+ setAssignedGarage(garage: Garage)
+ authenticate(username: String, password: String)
+ load(id: int)
+ load(object: JSONObject)
+ save()
<<enumeration>>
Role
Employee
Administrator
1
1
0..*
1
1
0..1
0..*
1
1
1
1
1
1
0..*

Here's a breakdown of each class in the UML diagram:

1. **DataLoader**
   - Attributes:
     - `data`: Holds static instance of parent `JSONObject`.
     - `sourcePath`: Path to the data file (`application_data.json`). *Immutable.*
     - `modified`: Boolean indicating if the data has been modified.
   - Methods:
     - `DataLoader()`: Constructor for initializing the `DataLoader`.
     - `loadData()`: Loads data from the file.
     - `saveData()`: Saves data to the file.
     - `getJSONObject(key: String)`: Loads a child `JSONObject` from the parent `JSONObject (data)` by key name.
     - `getJSONArray(key: String)`: Loads a child `JSONArray` from the parent `JSONObject (data)` by key name.
     - `put<T>(key: String, value: T)`: Sets a generic value in the parent `JSONObject` to the provided key name.

2. **DataLoaderable (Abstract Class)**
   - Methods:
     - `load(id: int)`: Static function that creates a new object from the provided `id` and the `DataLoader` class.
     - `load(object: JSONObject)`: Creates an instance of the object from the provided `JSONObject`.
     - `save()`: Saves the object.

3. **Garage**
   - Inherits from `DataLoaderable`.
   - Attributes:
     - `id`: Unique identifier. *Immutable.*
     - `name`: Name of the garage.
     - `address`: Address of the garage.
     - `currentParkingFee`: Current parking fee (of type `Fee`).
     - `occupiedSpaces`: Number of occupied spaces.
     - `totalSpaces`: Total number of parking spaces.
   - Constructors:
     - `Garage(name: String, address: String, totalSpaces: int)`: Creates a new `Garage` object with the `name`, `address`, and `totalSpaces` attributes set from argument values.
     - `Garage(id: int, name: String, address: String, currentParkingFee: Fee, totalSpaces: int)`: Private constructor used by `load` functions to reconstruct a saved object with all attributes initialized.

- Methods:
  - `getID()`, `getName()`, `getAddress()`, etc.: Getters and setters for attributes.
  - `decrementAvailableSpaces()`: Increments the `occupiedSpaces` count by 1.
  - `incrementAvailableSpaces()`: Decrements the `occupiedSpaces` count by 1.
  - `loadAllGarages()`: Returns an array of `Garage` objects loaded from the `DataLoader`.

4. **Ticket**
   - Inherits from `DataLoaderable`.
   - Attributes:
     - `id`: Ticket ID. *Immutable.*
     - `garage`: `Garage` associated with the ticket. *Immutable.*
     - `ticketFee`: `Fee` for the ticket. *Immutable.*
     - `entryDateTime`: Vehicle entry time. *Immutable.*
     - `exitDateTime`: Vehicle exit time.
     - `payment`: `Payment` associated with the ticket.
   - Constructors:
     - `Ticket(garage: Garage)`: Creates a new `Ticket` object with the `garage` attribute set from argument value. Additionally, sets the `ticketFee` based on the garage’s `currentParkingFee` and `entryDateTime` to now.
     - `Ticket(id: int, garage: Garage, ticketFee: Fee, entryDateTime: LocalDateTime, exitDateTime: LocalDateTime, payment: Payment)`: Private constructor used by `load` functions to reconstruct a saved object with all attributes initialized.
   - Methods:
     - `getID()`, `getGarage()`, `getEntryDateTime()`, etc.: Getters and setters for attributes.
     - `loadTicketsForGarage(garageId: int)`: Returns an array of `Ticket` objects loaded from the `DataLoader`.
     - `calculateFee()`: Calculates the fee owed by the vehicle owner for the ticket.

5. **Payment**
   - Inherits from `DataLoaderable`.
   - Attributes:
     - `id`: Payment ID. *Immutable.*
     - `capturedDateTime`: Time the payment was captured. *Immutable.*
     - `capturedBy`: `User` who captured the payment. *Immutable.*
     - `paymentMethod`: Method of payment (`PaymentMethod`). *Immutable*
     - `value`: Amount of payment, stored in cents as an integer to prevent any floating point arithmetic errors. *Immutable.*

- Constructors:
    - `Payment(paymentMethod: PaymentMethod, value: int)`: Creates a new `Payment` object with the `paymentMethod` and `value` attributes set from argument values. Additionally, sets the `capturedDateTime` to now.
    - `Payment(capturedBy: User, paymentMethod: PaymentMethod, value: int)`: Creates a new `Payment` object with the `capturedBy`, `paymentMethod`, and `value` attributes set from argument values. Additionally, sets the `capturedDateTime` to now.
    - `Payment(id: int, capturedDateTime: LocalDateTime, capturedBy: User, paymentMethod: PaymentMethod, value: int)`: Private constructor used by `load` functions to reconstruct a saved object with all attributes initialized.
- Methods:
    - `getID()`, `getCapturedDateTime()`, `getCapturedBy()`, etc.: Getters for attributes.

6. Fee

- Inherits from `DataLoaderable`.
- Attributes:
    - `id`: Fee ID. *Immutable.*
    - `type`: Type of fee (`FeeType`). *Immutable.*
    - `cost`: Cost of the fee. *Immutable.*
- Constructors:
    - `Fee(type: FeeType, cost: int)`: Creates a new `Fee` object with the `type` and `cost` attributes set from argument values.
    - `Fee(id: int, type: FeeType, cost: int)`: Private constructor used by `load` functions to reconstruct a saved object with all attributes initialized.
- Methods:
    - `getID()`, `getType()`, `getCost()`: Getters for attributes.

7. User

- Inherits from `DataLoaderable`.
- Attributes:
    - `id`: User ID. *Immutable.*
    - `name`: User's name. *Immutable.*
    - `username`: Username for login. *Immutable.*
    - `password`: Password for login.
    - `role`: Role of the user (`Role`).
    - `assignedGarage`: Garage assigned to the user (`Garage`).
- Constructors:
    - `User(name: String, username: String, role: Role, assignedGarage: Garage)`: Creates a new `User` object with the `name`,

    `username`, `role`, and `assignedGarage` attributes set from argument values.
    - `User(id: int, name: String, username: String, password: String, role: Role, assignedGarage: Garage)`: Private constructor used by `load` functions to reconstruct a saved object with all attributes initialized.
- Methods:
    - `getID()`, `getName()`, `getUsername()`, etc.: Getters for attributes.
    - `setPassword(password: String)`, `setRole(role: Role)`, `setDefaultGarage(garage: Garage)`: Setters for attributes.
    - `authenticate(username: String, password: String)`: Authenticates user login.

8. **Report (Interface)**
    - Methods:
        - `getFilters()`: Retrieves filters for the report.
        - `setFilter<T>(name: String, value: T)`: Sets a filter (report attribute).
        - `runReport()`: Executes the report.

9. **SpaceAvailabilityReport**
    - Inherits from `Report`.
    - Attributes:
        - `garage`: Garage associated with the report.
        - `filters`: Filter options (contains only "garage").

10. **RevenueReport**
    - Inherits from `Report`.
    - Attributes:
        - `garageId`: ID of the garage.
        - `startDate`: Start date for the report.
        - `endDate`: End date for the report.
        - `filters`: Filter options (contains "garageId", "startDate", and "endDate").

11. **Message**
    - Implements `Serializable`.
    - Attributes:
        - `type`: Message type (`MessageType`).
        - `data`: Data associated with the message.
    - Constructor:
        - `Message(type: MessageType, data: String)`: Initializes with message type and data.
    - Methods:
        - `getType()`: Retrieves the message type.

  - `getData()`: Retrieves the data.

12. Server

- Attributes:
  - `clientSockets`: Holds a list of the connected client sockets for broadcast messages.
- Methods:
  - `main()`: Entry point for the server application.
  - `broadcast()`: Iterates over client sockets and broadcasts a message (ping).
  - `sendMessage(client: Socket, message: Message)`: Sends a message to the provided client.

13. Client

- Attributes:
  - `gui`: Stores the GUI class instance.
- Methods:
  - `main()`: Entry point for the client application.
  - `sendMessage(message: Message)`: Sends a message to the server.

14. GUI

- Constructor:
  - `GUI`: Creates a new `GUI` object instance.
- Methods:
  - `getStartPanel()`: Returns the GUI elements for the start panel (login and kiosk mode selection).
  - `getEntryKiosk()`: Returns the GUI elements for the entry kiosk mode.
  - `getExitKiosk()`: Returns the GUI elements for the exit kiosk mode.
  - `getEmployeePanel()`: Returns the GUI elements for the employee panel.
  - `getAdminPanel()`: Returns the GUI elements for the admin panel.

Enumerations

1. MessageType

- Values: `Garage`, `Login`, `Logout`, `Payment`, `Report`, `Ticket`, `User`.

2. FeeType

- Values: `Hourly`, `Daily`.

3. PaymentMethod

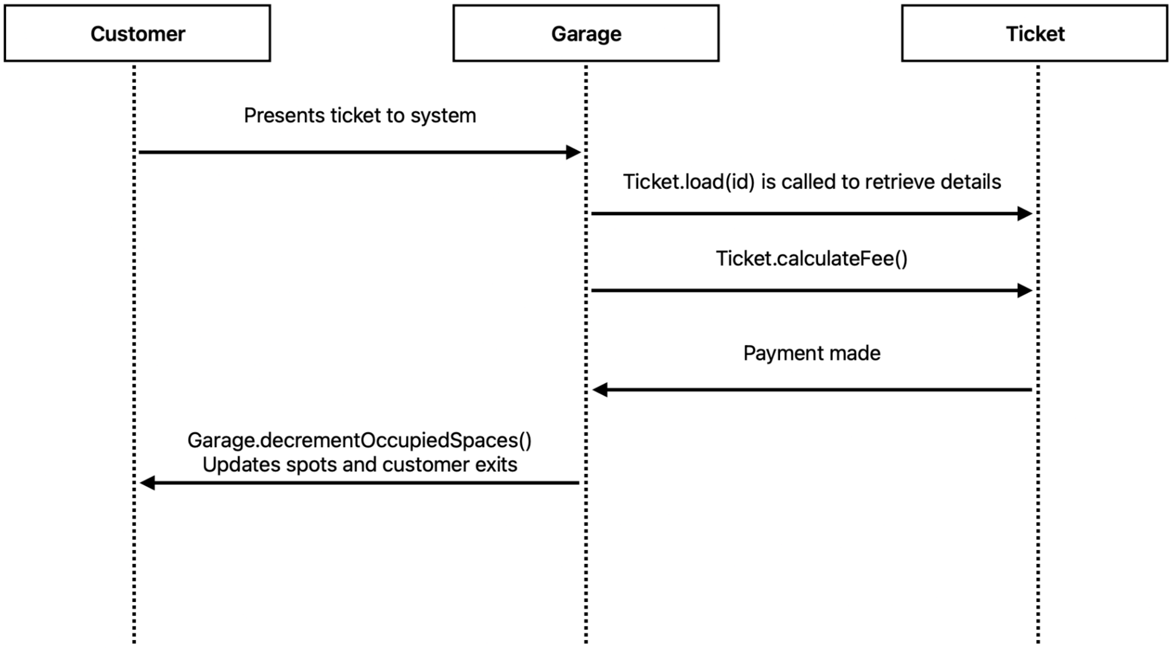
- Values: `Cash`, `CreditCard`.

4. Role

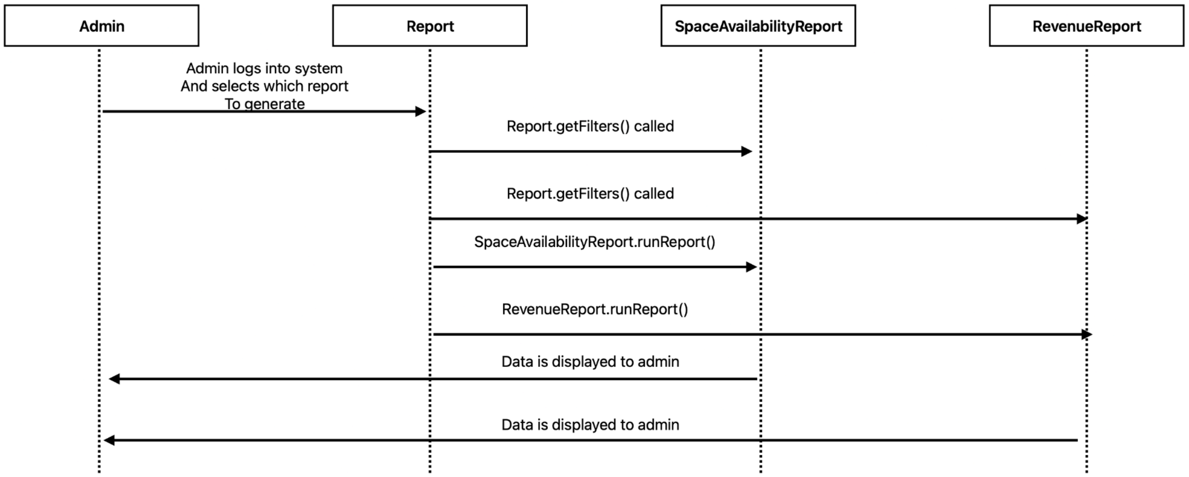
- Values: `Employee`, `Administrator`.

Sequence Diagram

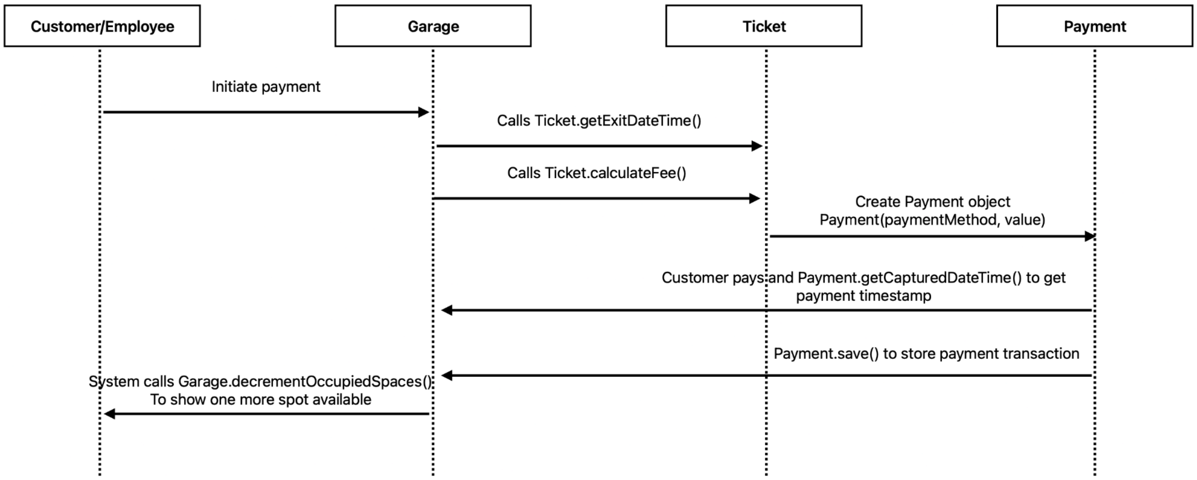
Car Removal


Customer
Garage
Ticket
Presents ticket to system
Ticket.load(id) is called to retrieve details
Ticket.calculateFee()
Payment made
Garage.decrementOccupiedSpaces()
Updates spots and customer exits

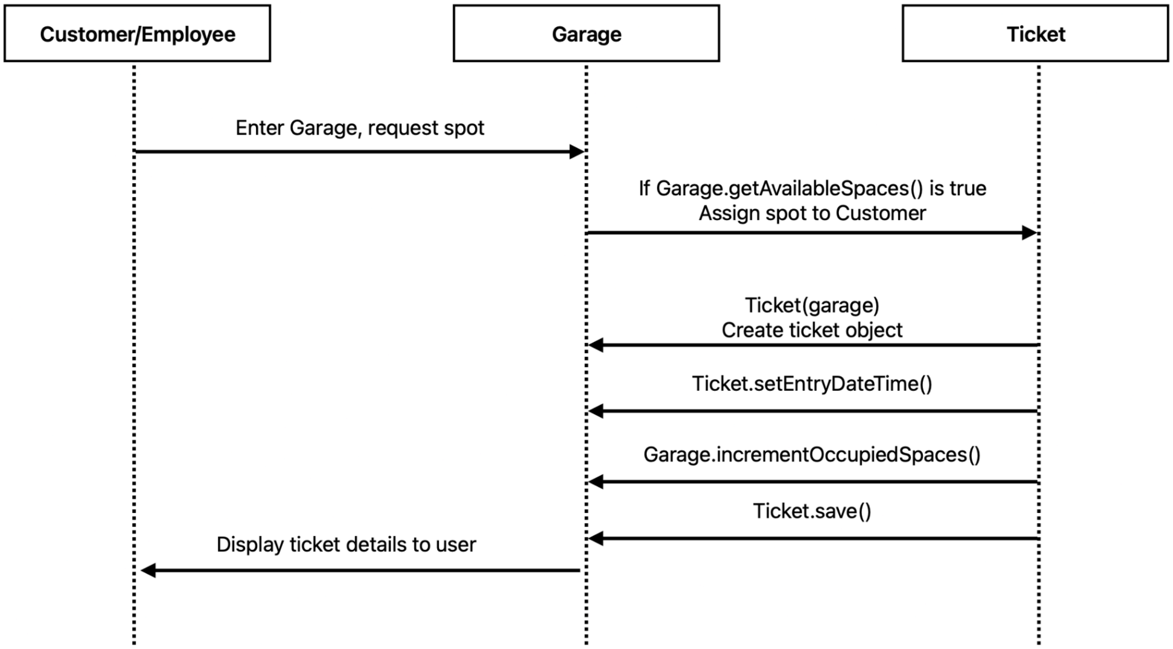
Report Generation


Admin
Report
SpaceAvailabilityReport
RevenueReport
Admin logs into system
And selects which report
To generate
Report.getFilters() called
Report.getFilters() called
SpaceAvailabilityReport.runReport()
RevenueReport.runReport()
Data is displayed to admin
Data is displayed to admin

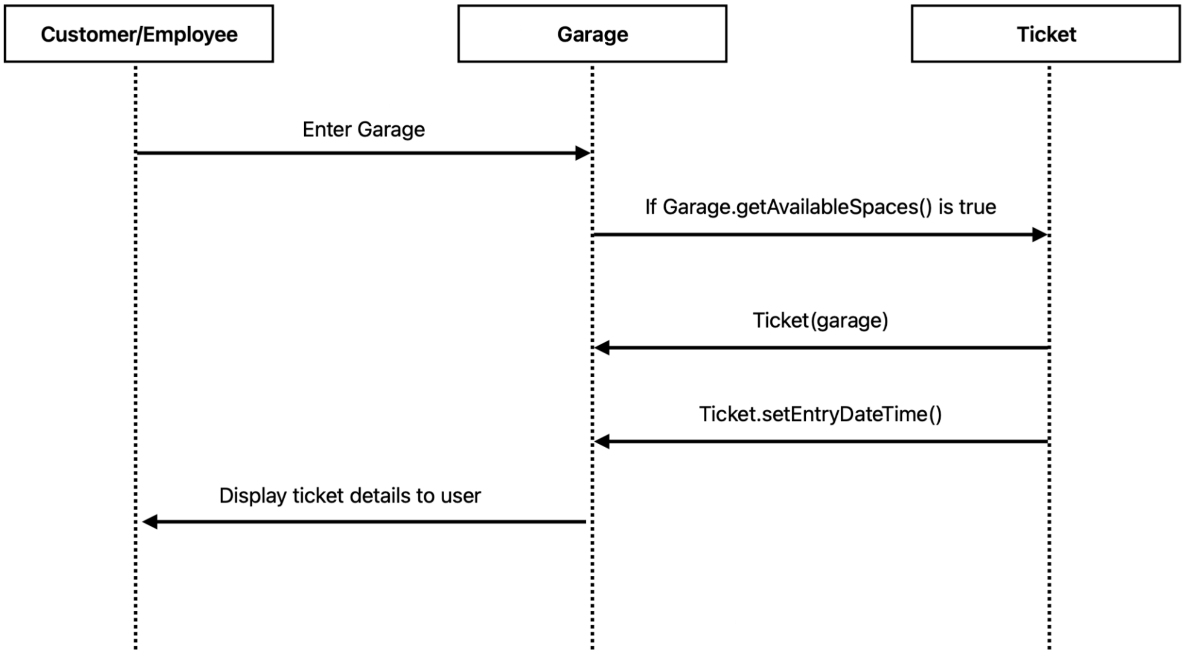
Payment Processing


Customer/Employee
Garage
Ticket
Payment
Initiate payment
Calls Ticket.getExitDateTime()
Calls Ticket.calculateFee()
Create Payment object
Payment(paymentMethod, value)
Customer pays and Payment.getCapturedDateTime() to get
payment timestamp
Payment.save() to store payment transaction
System calls Garage.decrementOccupiedSpaces()
To show one more spot available

Parking Spot Assignment


Customer/Employee
Garage
Ticket
Enter Garage, request spot
If Garage.getAvailableSpaces() is true
Assign spot to Customer
Ticket(garage)
Create ticket object
Ticket.setEntryDateTime()
Garage.incrementOccupiedSpaces()
Ticket.save()
Display ticket details to user

Ticket Generation


Customer/Employee
Garage
Ticket
Enter Garage
If Garage.getAvailableSpaces() is true
Ticket(garage)
Ticket.setEntryDateTime()
Display ticket details to user

Design Patterns

To create a maintainable and efficient design for the Parking Garage Management System (PGMS), we incorporated the Facade and Singleton patterns. These patterns simplify interactions between system components, centralize resource management, and enhance flexibility as the system grows.

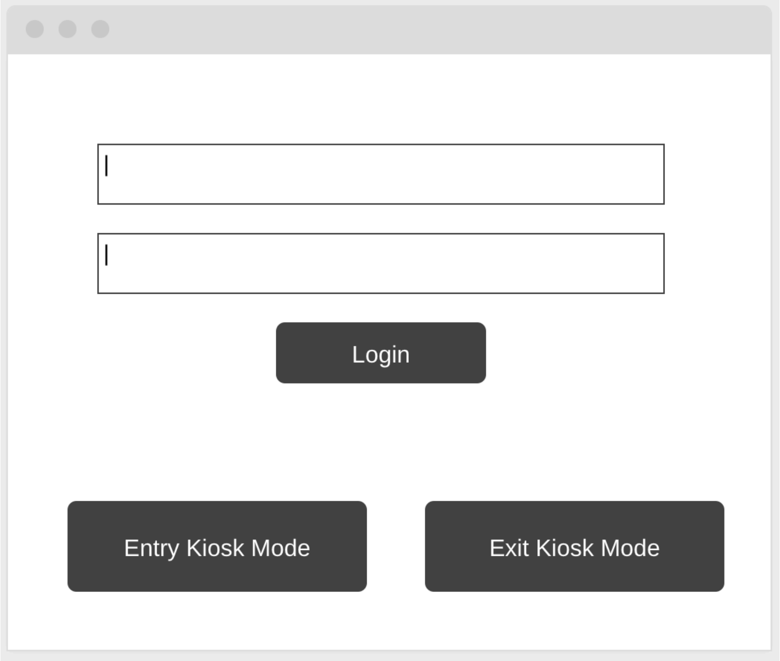
Client-Server Pattern

- Purpose: The Client-Server pattern separates the system into two main parts:
  - The Client (UI) handles interactions like ticket generation, payment processing, and viewing reports.
  - The Server processes these requests, manages data, and sends results back to the client
- Application in PGMS:
  - Client Side: Used by Employees and Admins to request actions (e.g., generating tickets or processing payments). The client interface sends these requests to the server.
  - Server Side: Handles data processing and storage. It receives client requests, updates data, and sends responses. DataLoader acts as the server's data manager by loading and saving information in JSON files.
- Benefits:
  - Centralized Data
  - Modularity
  - Scalability

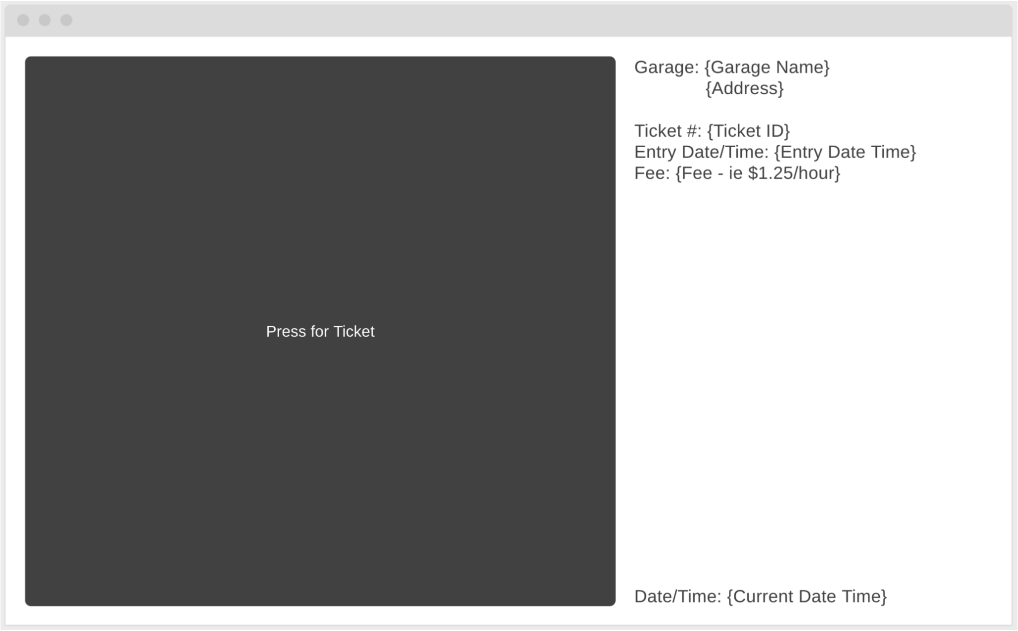
Facade Pattern

- Purpose: The Facade pattern provides a simplified, unified interface to complex subsystems within the PGMS, By using a Facade, the system hides intricate details of internal processes, allowing client modules to perform functions without extensive knowledge of underlying interactions.
- Where it's Applied:
    - Subsystems Managed
        - Ticket Management
        - Payment Processing
        - Parking Spot Management
- Implementation Details: A ParkingFacade class serves as the central interface for ticket generation, payment processing, and managing parking spots. It interacts with subsystem classes including Ticket, Payment, and Garage to streamline ticket issuance, parking space updates, and fee calculations.
- ParkingFacde interacts with:
    - Ticket: For generating new tickets and managing entry and exit times.
    - Payment: For processing payments and associating them with specific tickets.
    - Garage: For managing real-time parking spot availability.
    - Fee: To handle fee calculations based on parking duration.

Wireframe Design



Login Screen

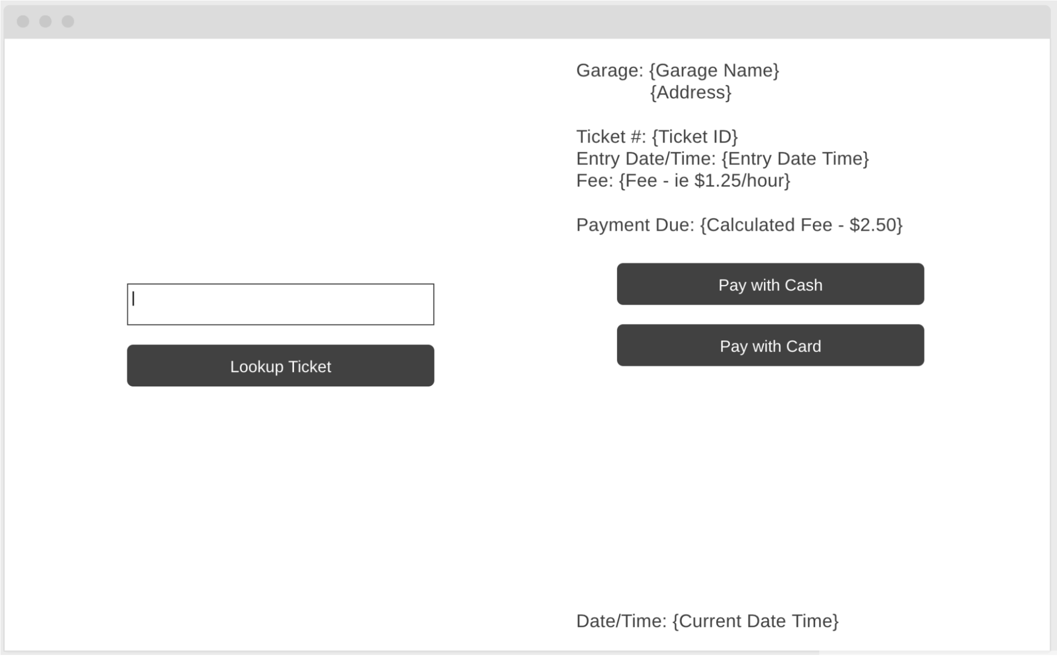


Garage: {Garage Name}
{Address}

Ticket #: {Ticket ID}
Entry Date/Time: {Entry Date Time}
Fee: {Fee - ie $1.25/hour}

Date/Time: {Current Date Time}

Enter Ticket



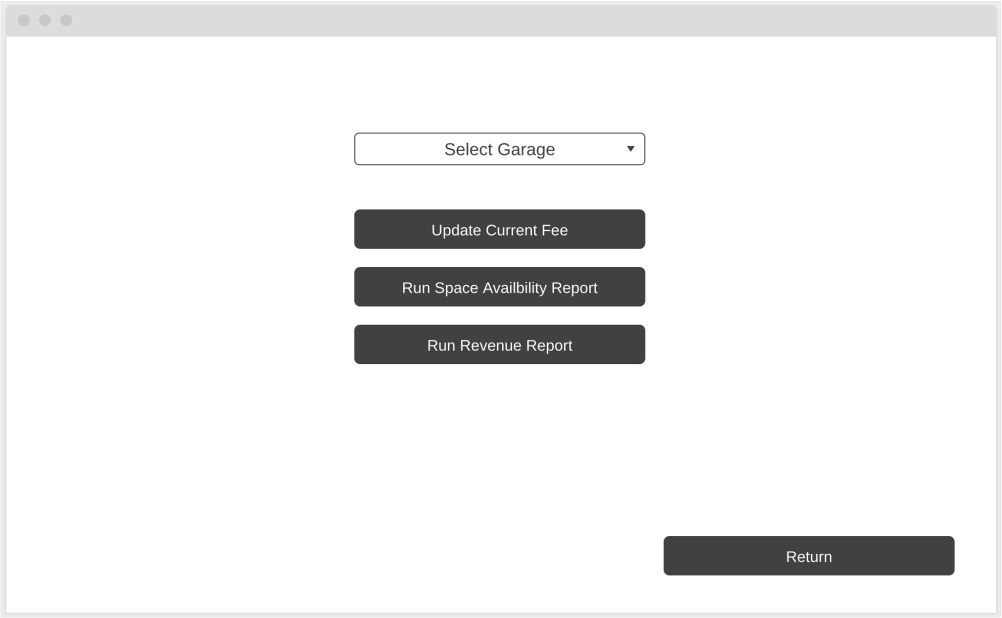
Garage: {Garage Name}
{Address}

Ticket #: {Ticket ID}
Entry Date/Time: {Entry Date Time}
Fee: {Fee - ie $1.25/hour}

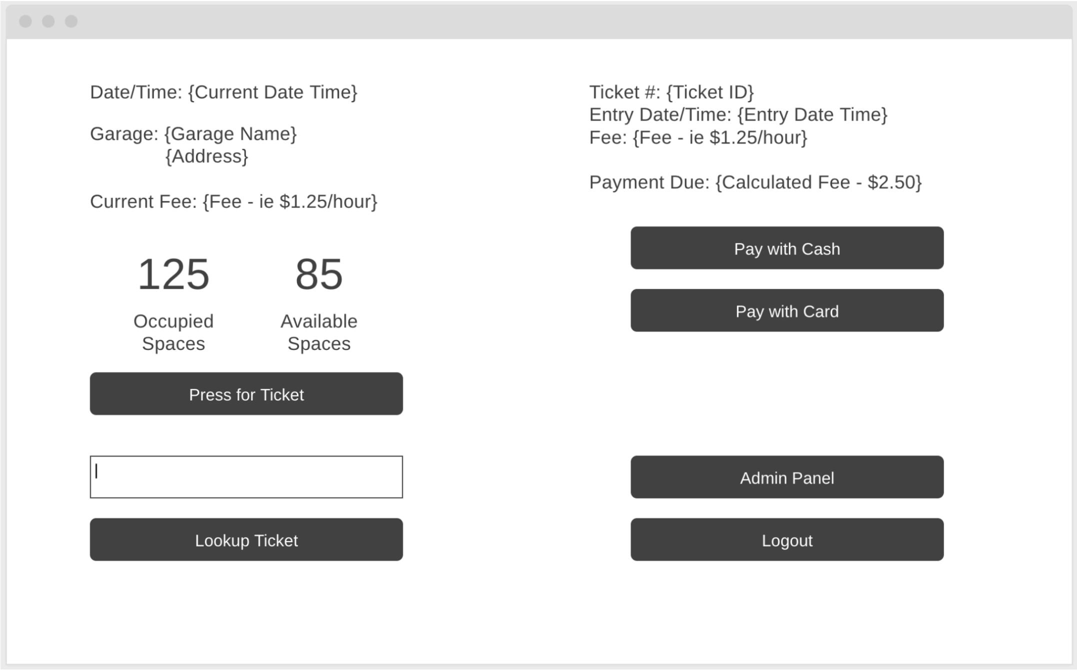
Payment Due: {Calculated Fee - $2.50}

Date/Time: {Current Date Time}

Exit Ticket



Garage Selection


Date/Time: {Current Date Time}

Garage: {Garage Name}
{Address}

Current Fee: {Fee - ie $1.25/hour}

125 Occupied Spaces

85 Available Spaces

Ticket #: {Ticket ID}
Entry Date/Time: {Entry Date Time}
Fee: {Fee - ie $1.25/hour}

Payment Due: {Calculated Fee - $2.50}

Dashboard