OOP Course Project

Project name:
Parking Lot Management System

Dr. Ahmet KAPLAN

Name: Semanur ÇAM
Student ID: 63210108
Department: IE

Project Overview

Project Information:

The Parking Lot Management System is a Java-based app that is meant to make parking easier. It makes it easy for users to handle parking for cars, print tickets, figure out fees, and keep track of which spots are open and which ones are taken. The system aims to automate control of parking lots to make them more accurate and easier for people to use.

Purpose and Objectives:

• Make using an easy-to-understand method in the parking lot easier.
• Cut down on mistakes made by hand by automating the process of making tickets and figuring out fees.
• Show real-time availability of parking spots to get the most out of parking.

Possible Users or Audience:

Parking lot owners and managers, drivers who use parking lots often, and businesses that manage big parking lots (like malls and airports) are all possible users or audiences.

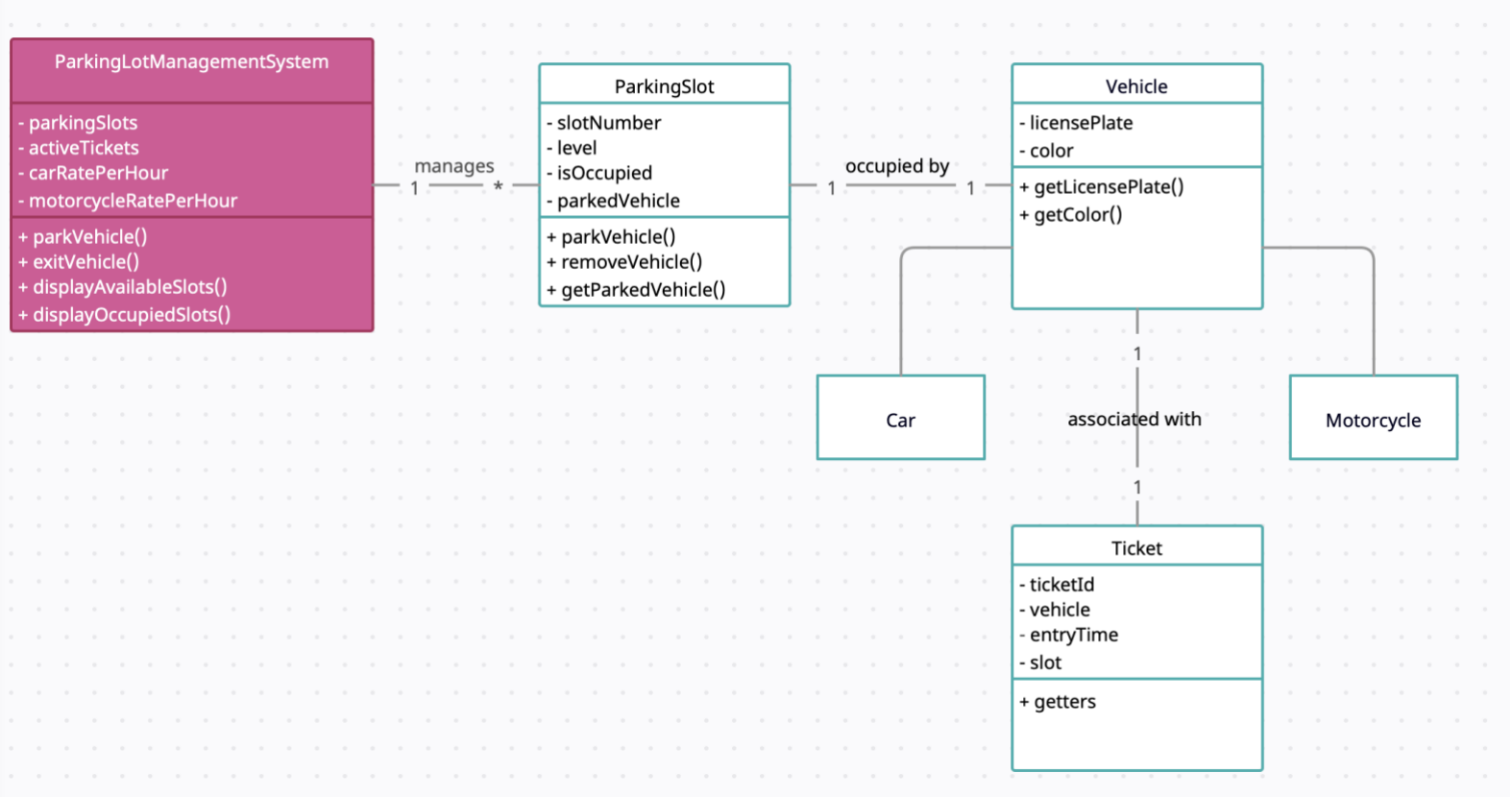
Problem Statement:

Manual management of parking facilities frequently results in inefficiencies, inaccuracies, and consumer discontent stemming from misfiled records, erroneous charge assessments, or lack of spaces.

Why is it Important?

An automated Parking Lot Management System promotes operational efficiency, improves customer experience, and decreases operational costs by offering a reliable, precise, and user-friendly platform for managing parking facilities.

System Architecture



Inheritance:
- Car and Motorcycle **inherit** from the abstract class Vehicle.

Associations:
- ParkingLotManagementSystem is **associated** with multiple ParkingSlot objects (1-to-Many).

- ParkingSlot is **associated** with one Vehicle (1-to-1).

- Ticket is **associated** with one ParkingSlot and one Vehicle (1-to-1).

Composition:
- ParkingLotManagementSystem has a **composition** relationship with ParkingSlot. (Slots cannot exist without the system.)

Key Features

Feature 1: Vehicle Parking and Ticket Management

This system makes parking and citations efficient. Each automobile has a parking space, and a UUID system generates a unique ticket ID to ensure tracking. The system checks slot availability before allowing car parking. The system stores entry time, vehicle details, and slot number in an active ticket database after parking. This mechanism ensures smooth and precise parking operations in large multi-level complexes. Ticket handling is automated, reducing human error.

Users can sell slot-specific tickets and track slot occupancy in real time with this functionality. This helps organize busy parking lots where cars and motorcycles constantly enter and exit. The system speeds ticket retrieval at exit, speeding departure.

Feature 2: Parking Fee Calculation

A detailed parking fee computation algorithm is included. The charge depends on the vehicle type (car or motorcycle) and parking duration. Cars cost $3 per hour, while motorbikes cost $1.50. The system calculates parking length in minutes using arrival and exit timestamps and rounds up to the nearest hour for billing. This ensures fair and accurate fee collection, preventing overcharges and undercharges.

This feature benefits parking facility owners and managers who want to maximize revenue. Automatic billing eliminates tedious computations, reducing administrative burdens and disagreements. The vehicle-type-based charge structure ensures user transparency and helps parking lot management track finances.

Feature 3: Slot Availability Display

The system displays available and occupied parking spaces. Users can view all slots by level and number in real time. The license plate number, color, and ticket ID of parked cars are shown alongside occupied slots. In multi-level parking garages, where vacancies can be hard to see, this information helps consumers find a place quickly.

This feature shows real-time availability, making it easier to find open slots and improving user experience. It also provides administrators with a complete view of slot occupancy, improving parking space management. This function boosts efficiency and customer satisfaction in high-traffic parking lots.

Feature 4: Multi-Level Parking Support

Multi-tiered parking is designed into the system. Each parking space has a level, so users can choose their level. This design keeps the parking lot organized while accommodating several cars. The slot management system ensures vehicles park in their spaces, reducing parking confusion.

This multi-tiered help is crucial for urban areas with limited space and vertical parking facility construction. The technology optimises car distribution by efficiently using all floors of the parking facility. It simplifies navigation by letting users see available spaces on each level and choose.

Feature 5: Vehicle Type Differentiation

The system distinguishes cars and motorcycles. This matters because parking facilities may have different zones or spaces for different car categories and different rates. Motorcycle charge rates are lower than car charging rates since they are smaller. This ensures that the parking facility accommodates a variety of users while maintaining equity and organization.

The system allocates slots by vehicle type to maximize space use. This is especially beneficial for parking facilities with limited space or different customers. This function helps organize parking so vehicles are parked according to their needs.

OOP Concepts Implemented

1. Classes and Objects

Core Classes in the Project:

The Parking Lot Management System is structured around key classes like `ParkingLot`, `ParkingSlot`, `Vehicle`, `Ticket`, and `FeeCalculator`.

- **`ParkingLot`**: Manages the overall parking structure, including levels and slots.
- **`ParkingSlot`**: Represents individual parking slots, with attributes like slot ID, level, and availability.
- **`Vehicle`**: Represents a vehicle with attributes like type, license plate, and color.
- **`Ticket`**: Manages ticket-related data, such as entry time, exit time, and associated vehicle.
- **`FeeCalculator`**: Responsible for calculating parking fees based on vehicle type and duration.

Objects are instantiated from these classes to manage real-world entities like vehicles parked in the lot or tickets generated for users.

2. Encapsulation

Encapsulation is implemented using private attributes and methods to ensure data security and proper access control. For example:

- In the `ParkingSlot` class, the `availability` attribute is private (`__availability`). Public getter and setter methods (`is_available()` and `set_availability(status)`) control access to this attribute.
- In the `Ticket` class, the entry and exit timestamps are private, and methods like `get_entry_time()` and `get_exit_time()` allow controlled access.

Encapsulation ensures that sensitive data, such as slot availability or ticket details, cannot be directly accessed or modified without proper validation.

3. Inheritance

The project uses inheritance to create a hierarchy of related classes.

- **Example**: The `Car` and `Motorcycle` classes inherit from the base `Vehicle` class.
  - The `Vehicle` class contains common attributes like license plate, color, and type.
  - `Car` and `Motorcycle` extend `Vehicle` by including specific methods or attributes, such as parking slot size requirements.

This hierarchy ensures code reusability and reduces redundancy, as shared properties and methods are implemented in the parent class.

4. Polymorphism

Polymorphism is demonstrated through method overriding and overloading:

- **Method Overriding**:
  The `calculate_fee()` method is overridden in the `FeeCalculator` class for different vehicle types. For instance:
  - Cars are charged $3 per hour.
  - Motorcycles are charged $1.5 per hour.
- **Method Overloading** (if the programming language supports it):
  In the `ParkingSlot` class, a method `assign_vehicle()` may be overloaded to accept different parameters, such as vehicle type or vehicle object.

Polymorphism allows flexibility in how methods are called and executed based on the object type, enhancing the scalability of the project.

5. Abstraction

Abstraction is implemented using abstract classes or interfaces to define the blueprint for specific functionality.

- **Example**: The `FeeCalculator` class can be implemented as an abstract base class with an abstract method `calculate_fee()`.
    - Concrete classes for `CarFeeCalculator` and `MotorcycleFeeCalculator` implement the `calculate_fee()` method with specific logic.

By using abstraction, the system ensures that essential methods are implemented in derived classes, while hiding unnecessary details from external users.

Evaluation of OOP Concepts

1. Classes & Objects

The class structure is well-defined, with each class representing a real-world entity or functionality. For instance, the `Vehicle` class models vehicle attributes, while the `Ticket` class handles ticketing operations. These objects interact seamlessly to represent the parking lot ecosystem.

2. Encapsulation

Getters and setters are used extensively to control access to private attributes. For example:

- `ParkingSlot` uses private attributes like `__slot_id` and `__availability`, ensuring data security.
- Controlled access prevents unauthorized modification of sensitive attributes.

3. Inheritance

The class hierarchy includes base and derived classes like `Vehicle`, `Car`, and `Motorcycle`. This structure eliminates redundancy and promotes code reuse. It also allows future extension, such as adding new vehicle types.

4. Polymorphism

Overriding the `calculate_fee()` method in `FeeCalculator` demonstrates polymorphism effectively. Different implementations for cars and motorcycles showcase how the system adapts behavior dynamically based on the object type.

5. Abstraction

Abstract classes like `FeeCalculator` define essential methods, ensuring a consistent interface across different implementations. This provides a clear structure for developers while hiding internal details from the end-user.

Overall, the implementation of OOP concepts ensures modularity, scalability, and maintainability in the Parking Lot Management System.

Challenges Faced

Difficulties Encountered During Development

System Design:
Designing a modular system with proper class relationships was initially tricky. For example, establishing clear relationships between ParkingLot, ParkingSlot, and Vehicle while avoiding circular dependencies required multiple iterations of the UML diagram.

Fee Calculation Logic:
Implementing a flexible fee calculation mechanism that could adapt to different pricing models for various vehicle types was complex. Managing edge cases, such as partial hours or vehicles overstaying their time, required additional logic.

Concurrency Management:
Handling simultaneous parking slot assignments in a multi-threaded environment presented challenges. Ensuring that two vehicles couldn't occupy the same slot required implementing thread-safe mechanisms.

Overcoming Challenges

For System Design:
Iterative prototyping helped refine the UML diagrams. Peer feedback and consulting online resources on object-oriented design patterns provided clarity in creating efficient class relationships.

For Fee Calculation Logic:
A strategy design pattern was implemented for fee calculation. This allowed dynamic assignment of fee calculation strategies (e.g., per hour, flat rate) to different vehicle types. Unit testing helped validate edge cases.

For Concurrency Management:
A locking mechanism was introduced for slot assignments, ensuring only one thread could modify a slot's status at any time.

Code Generator Tools Used

Eclipse IDE enhanced productivity with intelligent code suggestions, quick error detection, and built-in refactoring tools, which helped maintain clean and error-free code.

PlantUML was particularly useful for real-time updates to the UML diagrams, ensuring that design changes were consistently reflected in the documentation.

ScreenShots: