Parking Lot System Design (original code)(main.java)

1. Code Explanation

1.1. Overall Structure

This code represents a parking lot management system with customers, floors for parking, and payment methods. Here's an overview of the key components:

- Class `Customer`:

- Represents a customer with attributes such as ID, vehicle ID, entry time, exit time, vehicle type, and mobile number.

- Provides a method `amount` to calculate the parking fee based on entry and exit times.

- Class `Floor`:

- Represents a parking floor with attributes like floor number, vehicle type, size, and available slots.

- Provides methods to display the floor, handle vehicle entry, and exit.

- Class `Electric` (Unused):

- An extra class that seems unused in the current code.

- Class `Payment`:

- Represents payment methods with options for cash, credit card, and an online portal.

- Provides methods to simulate payment processes.

- Main Class `Main`:

- Contains the main logic for managing customer entries, exits, and payments.

- Utilizes objects of the above classes to implement parking lot functionality.

1.2. Object-Oriented Principles Used

1.2.1. Abstraction

- The `Customer`, `Floor`, and `Payment` classes abstract away the complexities of their respective functionalities.

1.2.2. Encapsulation

- Data related to customers and parking floors is encapsulated within the `Customer` and `Floor` classes, respectively, providing controlled access to the data.

1.2.3. Polymorphism

- Polymorphism is used through method overriding. For example, different payment methods (`cash`, `creditcard`, `onlinePortal`) implement their own behavior.

1.3. Java Concepts Used

- Java Time API: `java.time.LocalDateTime` and `java.time.Duration` are used for handling entry and exit times and calculating the duration between them.

- Exception Handling: The code uses try-catch blocks to handle exceptions related to thread interruption (`InterruptedException`).

2. Design Decisions

2.1. Use of Abstract Class, Interfaces, and Classes

- Class `Customer`:

- Represents common attributes and behaviors of customers.

- Provides a method for calculating parking fees.

- Class `Floor`:

- Represents parking floors with specific attributes and methods for handling entries and exits.

- Class `Payment`:

- Represents payment methods, providing flexibility to extend with new payment options if needed.

2.2. Other Java Concepts Used

- Java Time API:

- Utilized to handle time-related operations accurately.

- Exception Handling:

- Proper exception handling is implemented for scenarios like thread interruption during payment simulation.

2.3. Integration of OOPs

- Abstraction and Encapsulation:

- Abstraction is employed to hide complex calculations behind a simple `amount` method in the `Customer` class.

- Encapsulation is used to keep customer data and floor details encapsulated within their respective classes.

- Polymorphism:

- Polymorphism is used in the `Payment` class, where different payment methods provide their specific implementations.

2.4. OOPs Support for Design

- Modularity and Flexibility:

- Each class has a specific responsibility, promoting modularity and allowing easy extension of functionality.

- Payment methods can be added or modified independently.

- Code Reusability:

- Common behaviors are encapsulated within classes, promoting code reuse and reducing redundancy.

# Parking Lot System Design(Modified code)(2nd code)(kase.java)

1. Code Explanation

**1.1. Overall Structure**

This code represents a simple parking lot system with multiple floors and different types of vehicles. Here's an overview of the key components:

- **Abstract Class `Vehicle` and Its Subclasses**:

- `Vehicle` is an abstract class representing common attributes and behaviors of vehicles.

- Subclasses like `Bike`, `Car`, `ElectricCar`, and `Truck` inherit from `Vehicle` and implement specific behaviors for entering and exiting the parking lot.

- **Class `Floor`:**

- Represents individual parking floors with information about floor number, available slots, and methods for entering and exiting.

- **Class `Electric`:**

- Handles the charging process for electric vehicles, calculating charging costs, and offering payment options.

- **Class `Customer`:**

- Represents information about a customer's vehicle entry and exit times, vehicle type, and mobile number.

- Provides a method to calculate and display parking fees.

- **Class `Payment\_Method`:**

- Provides different payment methods such as cash, credit card, and online portal.

- **Class `Kase`(Main class):**

- Contains the main logic for taking customer input, managing parking, and calculating fees.

- Utilizes object-oriented principles to create instances of classes and invoke their methods.

**1.2. Object-Oriented Principles Used**

**1.2.1. Inheritance**

- Inheritance is used in this code to create a hierarchy of vehicle types (`Bike`, `Car`, `ElectricCar`, `Truck`) that inherit common attributes and methods from the `Vehicle` abstract class.

**1.2.2. Abstraction**

- The `Vehicle` class is an abstract class that defines the abstract methods `enter` and `exit`, allowing concrete subclasses to provide their implementations.

**1.2.3. Polymorphism**

- Polymorphism is exhibited through method overriding, where subclasses (`Bike`, `Car`, etc.) provide their specific implementations for the `enter` and `exit` methods.

**1.2.4. Encapsulation**

- Encapsulation is used to encapsulate data within classes and provide access through getters and setters. For example, the `Customer` class encapsulates customer data.

**2. Design Decisions**

**2.1. Use of Abstract Class, Interfaces, and Classes**

- Abstract Class `Vehicle`:

- It's used to define common attributes and methods for all vehicle types.

- The abstract methods `enter` and `exit` provide a blueprint for different vehicle types to implement their own entry and exit behaviors.

- Interfaces and Classes:

- An `interface` named `Payment` defines a contract for payment methods.

- The `Payment\_Method` class provides concrete implementations for different payment methods (`cash`, `creditcard`, `onlinePortal`).

**2.2. Other Java Concepts Used**

- Java Time API:

- `java.time.LocalDateTime` is used to handle entry and exit times of customers.

- `java.time.Duration` is used to calculate the time duration between entry and exit.

- Exception Handling:

- The code uses try-catch blocks to handle exceptions related to thread interruption (`InterruptedException`).

**2.3. Integration of OOPs**

- Inheritance and Abstraction:

- Inheritance is used to create a hierarchy of vehicle types.

- Abstraction is used through the `Vehicle` abstract class and abstract methods.

- Polymorphism:

- Polymorphism is achieved through method overriding, where different vehicle types provide their own behavior for entering and exiting.

- Encapsulation:

- Customer data is encapsulated within the `Customer` class, providing controlled access to the data.

**2.4. OOPs Support for Design**

- Flexibility and Extensibility:

- The use of abstract classes and subclasses allows for easy addition of new vehicle types with their unique behaviors.

- Payment methods can be extended without modifying existing code.

- Modularity:

- Each class has a specific responsibility, promoting modularity and separation of concerns.

- Code Reusability:

- Common vehicle attributes and behaviors are defined in the `Vehicle` abstract class, promoting code reuse.

**2.5 Modifications with respect to first code:**

In the modified code, several changes were made to improve the structure and organization of the code. Here are the key differences between the original and modified versions:

1. Class Structure and Organization:

- Original: The original code used a single large block with all the classes, including `Customer`, `Floor`, `Electric`, and `Payment`, defined in a single file. This made the code less modular and harder to maintain.

- Modified: The modified code introduced a more organized class structure. Each class was moved into its separate file, improving code maintainability and readability.

2. Vehicle and Customer Classes:

- Original: In the original code, the `Vehicle` and `Customer` classes had many fields and methods, including payment-related functionality, which made them less focused on their main responsibilities.

- Modified: In the modified code, the `Vehicle` and `Customer` classes were simplified. The `Customer` class calculates the parking fee, and the payment-related functionality was moved to the `Payment` class, following the Single Responsibility Principle.

3. Payment Handling:

- Original: Payment handling was tightly integrated into the `Customer` class, making it responsible for both parking fee calculation and payment processing.

- Modified: Payment handling was separated into the `Payment` class, which now contains methods for cash, credit card, and online portal payments. This separation adheres to the Single Responsibility Principle, making the code more maintainable and extensible.

4. Floor Class:

- Original: The `Floor` class had fields for `floorno`, `vehicletype`, `size`, and `availableSlots`. The `displayflo`, `enter`, and `exit` methods were responsible for displaying information and managing slot availability.

- Modified: The `Floor` class was improved in the modified code to focus solely on managing slot availability. It retains `floorno`, `size`, and `availableSlots` fields but eliminates the `vehicletype` and `displayflo` methods. This simplification aligns the class with the Single Responsibility Principle.

5. Electric Class:

- Original: The `Electric` class was introduced in the original code but wasn't used effectively.

- Modified: In the modified code, the `Electric` class was removed since it wasn't serving a clear purpose.

6. Main Method:

- Original: In the original code, the `main` method handled user input and interactions for parking and payment within a single large structure.

- Modified: In the modified code, the `main` method was improved to be more modular and organized. It separates the initialization of floors, customer data input, parking, fee calculation, and payment processing into different sections.

7. Latest Exit Times:

- Original: The original code tracked the latest exit times for different vehicle types within the `main` method.

- Modified: In the modified code, the tracking of latest exit times was retained but updated to consider the exit times of specific vehicle types.

Overall, the modified code improves code organization and adheres to object-oriented principles, such as the Single Responsibility Principle, by separating concerns into different classes. It also enhances code readability and maintainability.

Parking Lot(3rd solution idea);

1. Classes

* Creation of class such as Parking Lot for managing floors,slots and vehicle allocation.
* Declaration of Payment as an interface which I have done in my second code but with more methods.

1. Customer class:

* As in the above codes,Customer class will hold the details of the customer.
* As we have made a payment interface,Customer class will implement it and all the payment methods will be defined in the customer class.

1. Parking Lot class:

* In this class,I have declared an array of floors rather than the floor object.
* All the vehicle and slot allotment methods are described in this class.
* In this code,In Payment methods ,I have assigned basepay for all the vehicles other than hourly pay
* For e.g.Rs 50 for truck and other heavy vehicles,Rs 30 for cars and electric cars and Rs 20 for bikes and scooters.These should be paid compulsorily with some amount added on basepay on the basis of the hours it stays there.

1. Main Class:

* There is not much change in main class than the above two codes but the number of objects declared in this code would be less as we have less classes.