CASE STUDY 1: PARKING LOT EXPLAINING DESIGN AND DECISIONS OF OUR CODE

DESIGN :

We have created a building consisting of three floors where each floor consists of 100

parking lot spaces. We allotted a specific number of lot spaces for motorbikes, cars, electric bikes & cars and PWD bikes, respectively, as shown in the image.

The ground floor exclusively consists of only trucks and PWD bike parking slots. The remaining floors have cars & bikes, including electric vehicle parking spaces.

A charging option is given to electric vehicles.

The slots are allotted according to the size of the vehicle, compact vehicles are allotted one space, and large cars are allocated two slots.

A display board shows the availability of parking spaces on the required floor chosen. The user can pick whichever room is comfortable from the availability board.

Each floor has a Customer's info portal, where the user can pay the parking fees before leaving. User has the option to pay through Net banking and UPI.

The fare is collected according to the time the vehicle is parked and the type of vehicle. Fare is calculated using the "Per hour" parking fee model.

The user can either pay the fare at the exit point of the building instead of paying at the departure point of the floor.

Users paying through the customer info portal and paying at the exit gate have different exits.

Users are given the option to pay fees at the exit gate, where they can choose the mode of payment, either cash or card.

EXPLANATION OF OUR CODE

USER CLASS-

We created the class “User”, in which we stored the necessary details of the user. To keep the details of all users, we created an array of objects of the type “User”.

All the user's information is stored in “private” fields to hide the user's personal data.

The class has getters and setters of private fields as methods.

The in-time and out-time of the user are stored under the “Instant” data.

The class has the “print ticket” method, which prints the user's ticket.

FLOOR CLASS-

Each floor has a Boolean array “flr”, which keeps track of the vacant or occupied slots.

The class has the method “availability”, which checks for the availability of the user's vehicle in the floor and returns the position of the empty slot.

The method “SlotAllotment” changes the array “flr” elements from false to true for the allotted slots.

The method “SlotDeAllotment” changes the array “flr” elements from true to false for the emptied slots.

VEHICLE CLASS-

It is an abstract class that stores a particular vehicle's start and end index in a Boolean array and the base amount for payment during departure.

The classes “Car”, “Truck”, “ElectricCar”, “Handicapped”, and “MotorCycle” inherit the class vehicle to store the details of their respective vehicles.

PAYMENT CLASS-

The calculation of duration and amount and payment at exit is done in this class.

The method “calculate duration” calculates the duration of the vehicle in the parking lot using in-Time and out-Time and returns time in Minutes.

The “calculate amount” method calculates the amount used to pay according to the duration of stay and type of vehicle using the "per-hour parking fee model".

The methods “cash payment” and “credit card payment” displays the prototype of payment through cash and credit card.

COSTUMERINFOPORTAL CLASS-

The user gets an option for payment of fees through the customer-info portal.

Users can pay fees through net banking or UPI using methods “NetBanking” and “UPI”, which display the respective prototype.

In the main code, the user enters their details and opts for a floor to park their vehicle.

The ground floor is exclusively given for Trucks and Handicapped people. Every floor has slots for Electric cars. After the vehicle is parked, the user gets the ticket displayed on the screen. When the user wants to exit, the amount is calculated according to the duration and type of vehicle, and the user gets different payment options.

WHY HAVE WE USED ABSTRACT CLASSES AND OTHER CLASSES?

We have used classes for creating objects, which store properties/data of objects.

We have used the abstract class “Vehicle” as it allows different objects to use standard functionalities and the categories “Car”, “Truck”, etc. inherit the class Vehicle.

HAVE YOU USED ANY OTHER CONCEPTS FROM JAVA?

We have used private fields in the user class for security purposes.

We have used encapsulation and inheritance.

We have learnt to use “instant” inbuilt class to access the time at that particular instant.

We have learnt to create an array of objects to store multiple users’ data.

HOW HAVE OOPS BEEN INTEGRATED INTO YOUR SOLUTION?

Inheritance helped us to prevent the repetition of many standard functionalities in classes. For example, the class “CostumerInfoPortal” inherited the class “User” to access the user details during payment.

Abstraction helped us to show the critical information and hide the rest from the user.

We have created classes and objects to the types, which made us access all course functionalities just by completing the thing; this made us store a large number of attributes of each user by just creating the array of objects.

Oops, concepts helped us to distinguish and assign the attributes to the different classes according to their respective functionalities.

Using OOPs concepts increased the writability and readability of the code.

HOW OOPS CONCEPTS SUPPORTED YOUR DESIGN DECISION?

OOPs helped us in presenting a structured code which is easily understandable.

It helped us empower and restrict different classes' functionalities depending upon their use.

The concept helped us execute the code part-wise and made it simple to rectify any possible errors as the execution is decentralised.

TEAM -4 MEMBERS :

CS21B054 V. SWETHA REDDY

CS21B052 V.NIKHIL CHOWDARY

CS21B042 R.SAI KRISHNA

CS21B029 K. BHARGAVA CHAITANYA

CS21B024 J.PRANAY BHASKER REDDY

CS21B016 C.DAKSHAYANI