Class Diagram for the Parking Lot

Learn to create a class diagram for the parking lot system using the bottom-up approach.

We'll cover the following

In this lesson, we will identify and design classes, abstract classes, and interfaces based on the requirements we have previously gathered from the interviewer in our parking lot system.

Components of a parking lot system

As mentioned earlier, we should design the parking lot system using a bottom-up approach. Therefore, we will first identify and design the classes of the smaller components like vehicles and parking spots. Then, we will create the class of the entire parking lot system, including these smaller components.

Vehicle

Our parking lot system should have a vehicle object according to the requirements. The vehicle can be a car, a truck, a van, and a motorcycle. There are two ways to represent a vehicle in our system:

- Enumeration
- Abstract class

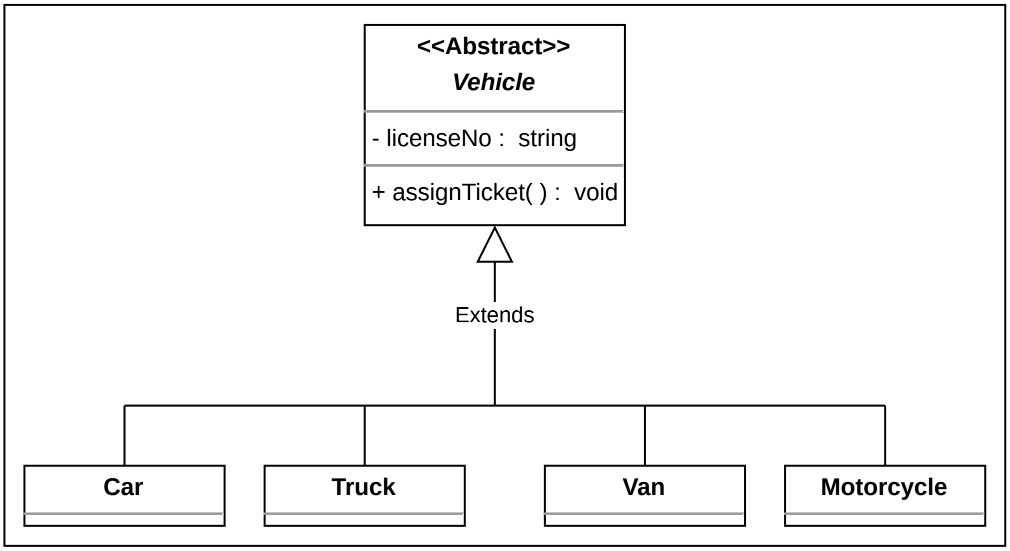
Enumeration vs. abstract class

The **enumeration** class creates a user-defined data type that has the four vehicle types as values.

This approach is not proficient for object-oriented design because if we want to add one more vehicle type later in our system, then we would need to update the code in multiple places in our code, and this would violate the Open Closed principle of the SOLID design principle. This is because the Open Closed principle states that classes can be extended but not modified. Therefore, it is recommended not to use the enumeration data type as it is not a scalable approach.

> **Note:** Using enums isn’t prohibited, but just not recommended. Later, we will use the `PaymentStatus` enum in our parking lot design as it won’t require further modifications.

An **abstract class** cannot instantiate the object and can only be used as a base class. The abstract class for `Vehicle` is the best approach. It allows us to create derived child classes for the `Vehicle` class. It can be extended easily in case the vehicle type changes in the future.



The class diagram of Vehicle and its derived classes

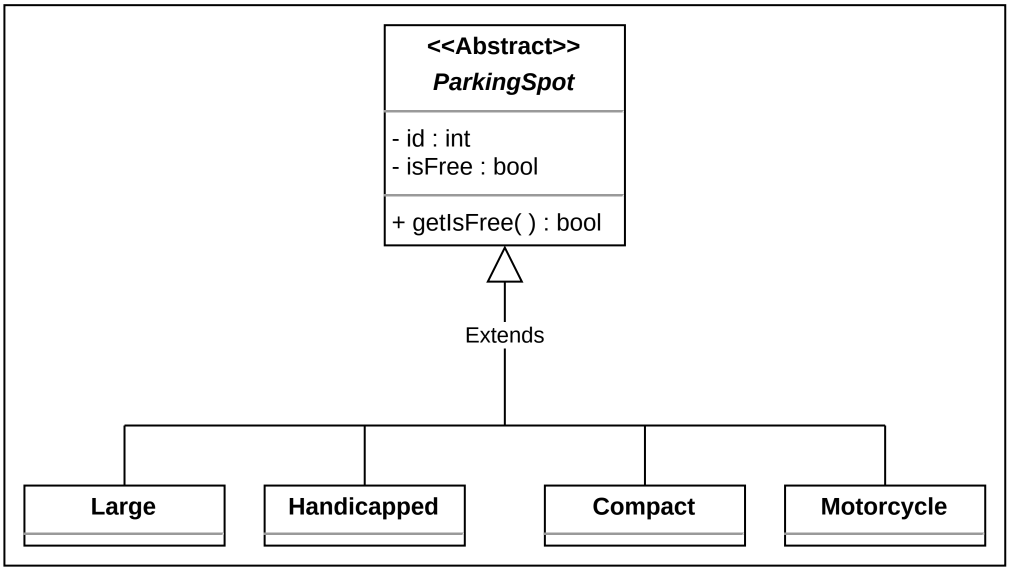
R4: Parking Lot System

R4: Four types of vehicles should be allowed to park in the parking lot, which are as follows:

- Car
- Truck
- Van
- Motorcycle

Parking spot

Similar to the `Vehicle` class, the `ParkingSpot` should also be an abstract class. There are four types of parking spots: handicapped, compact, large, and motorcycle. These classes can be derived from the parking spot abstract class.



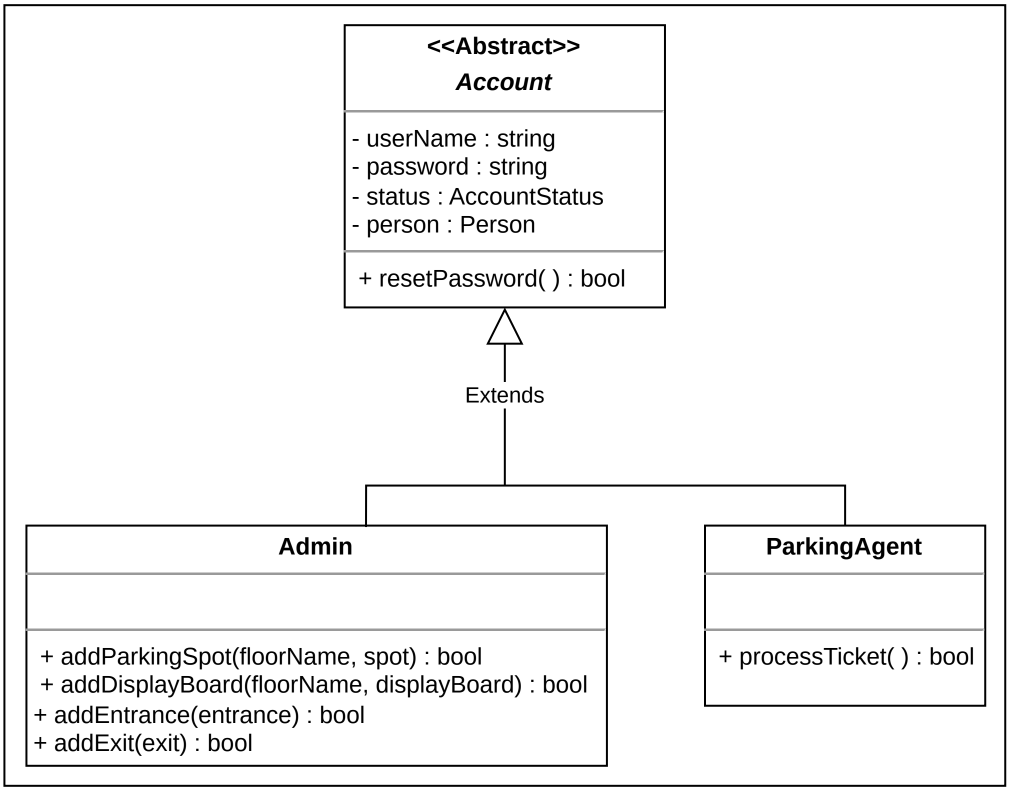
The class diagram of the ParkingSpot and its derived classes

R2: Parking Lot System

R2: There are four different types of parking spots: handicapped, compact, large, and motorcycle.

Account

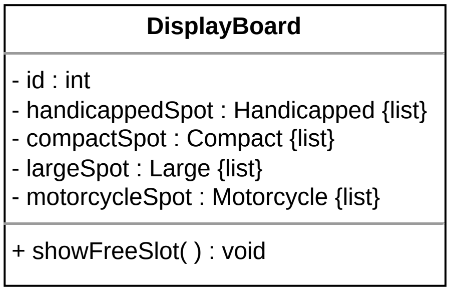
Similar to the `Vehicle` and `ParkingSpot` classes, `Account` should also be an abstract class. There are two child classes: `Admin` and `ParkingAgent`. These classes can be derived from the account abstract class.


<<Abstract>>
Account
- userName : string
- password : string
- status : AccountStatus
- person : Person
+ resetPassword( ) : bool
Extends
Admin
+ addParkingSpot(floorName, spot) : bool
+ addDisplayBoard(floorName, displayBoard) : bool
+ addEntrance(entrance) : bool
+ addExit(exit) : bool
ParkingAgent
+ processTicket( ) : bool

The class diagram of Account and its derived classes

Display board

This class represents the free parking spot types and the number of empty slots.


DisplayBoard
- id : int
- handicappedSpot : Handicapped {list}
- compactSpot : Compact {list}
- largeSpot : Large {list}
- motorcycleSpot : Motorcycle {list}
+ showFreeSlot( ) : void

The class diagram of the DisplayBoard class

R5, and R7: Parking Lot System

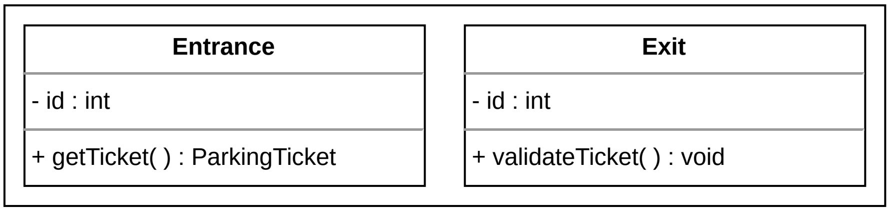
R5: The parking lot should have a display board that shows free parking spots for each parking spot type.

R7: If the parking lot is completely occupied, the system should show a message on the entrance and on the parking lot display board.

Entrance and exit

The `Entrance` class is responsible for returning the parking ticket whenever a vehicle arrives. It contains the ID attribute, since there are multiple entrances to the parking lot. It also has the `getTicket()` method.

The `Exit` class is responsible for validating the parking ticket's payment status before allowing the vehicle to exit the parking lot. It contains the ID attribute, since there are multiple exits to the parking lot. It also has the `validateTicket()` method.



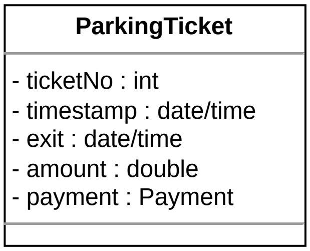
The class diagram of the Entrance and Exit classes

R8: Parking Lot System

R8: Customers should be able to collect a parking ticket from the entrance and pay at the exit.

Parking ticket

The `ParkingTicket` class is one of the central classes of the system. It keeps track of the entrance and exit times of the vehicles, the amount, and the payment status.



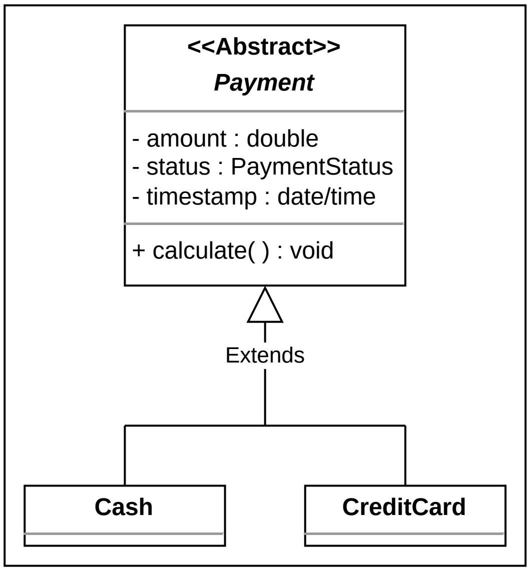
The class diagram of the ParkingTicket class

R8: Parking Lot System

R8: Customers should be able to collect a parking ticket from the entrance and pay at the exit.

Payment

The `Payment` class will be an abstract class and will have two child classes, `card` and `cash`, since these are two payment methods of the parking lot system.



The class diagram of the Payment class

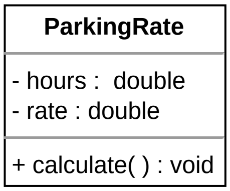
R9 and R10: Parking Lot System

R9: The customer can pay for the ticket either with an automated exit panel or pay the parking agent at the exit.

R10: The payment can be made using either a credit/debit card or cash.

Parking rate

The ParkingRate class is responsible for calculating the final payment based on the time spent in the parking lot.



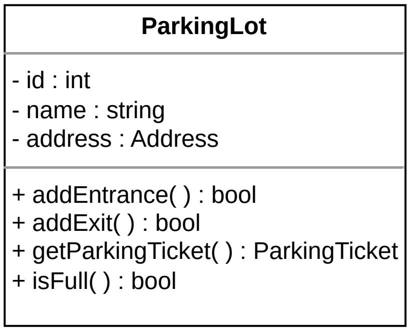
The class diagram of the ParkingRate class

R11: Parking Lot System

R11: The payment should be calculated at an hourly rate.

Parking lot

Now, we will discuss the design of the whole ParkingLot system class. This parking lot system is composed of smaller objects we have already designed, like entrance/exit, parking spots, parking rates, etc.



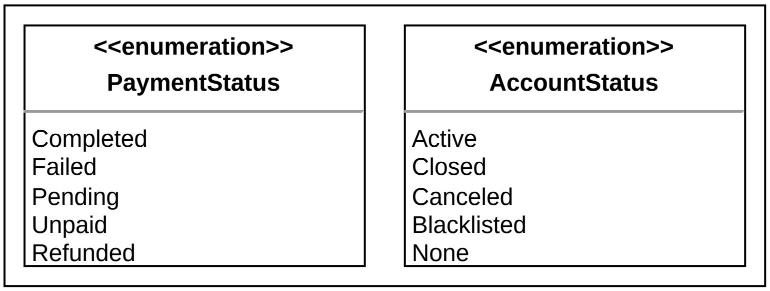
The class diagram for the ParkingLot class

The enumerations and custom data types

The following provides an overview of the enumerations and custom data types used in this problem:

- PaymentStatus: We need to create an enumeration to keep track of the payment status of the parking ticket, whether it is paid, unpaid, canceled, refunded, and so on.

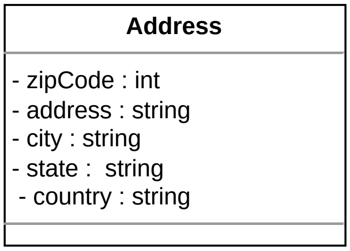
- AccountStatus: We need to create an enumeration to keep track of the status of the account, whether it is active, canceled, closed, and so on.



Enums in the parking lot system

Address

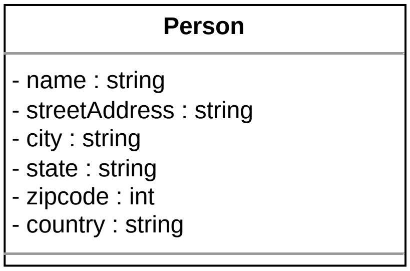
We also need to create a custom data type, Address, that will store the location of the parking lot.



The class diagram of the Address custom datatype

Person

The Person class is used to store information related to a person like a name, street address, country, etc.



The class diagram of the Person class custom datatype

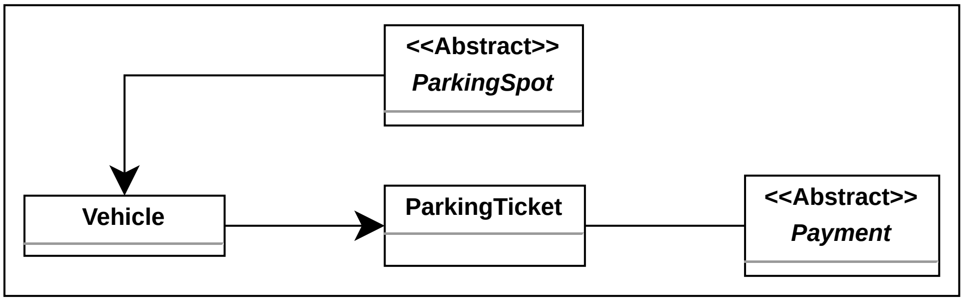
Relationship between the classes

Now, we'll discuss the relationships between the classes we have defined above in our parking lot system.

Association

The class diagram has the following association relationships:

- The `ParkingSpot` has a one-way association with `Vehicle`.
- The `Vehicle` has a one-way association with `ParkingTicket`.
- The `Payment` has a two-way association with `ParkingTicket`.

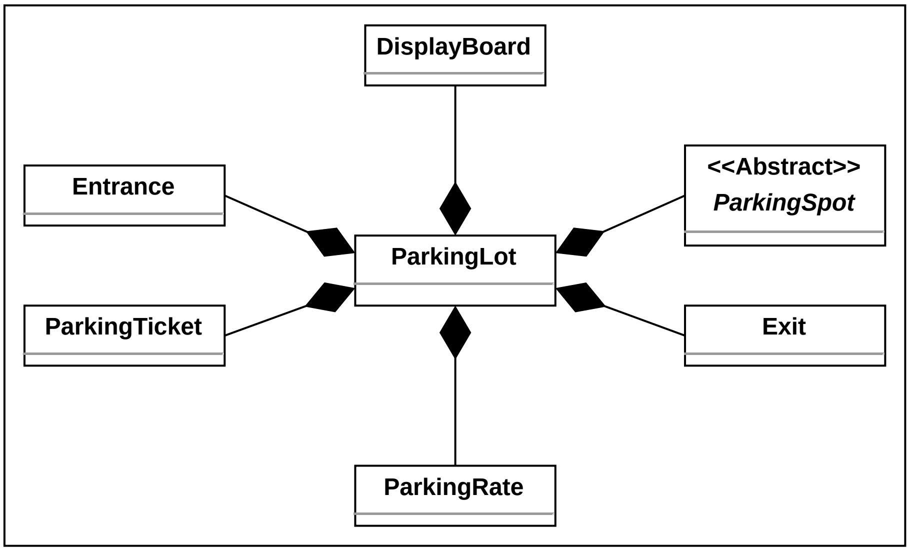


The association relationship between classes

Composition

The class diagram has the following composition relationships.

- The `ParkingLot` class includes `Entrance`, `Exit`, `ParkingRate`, `DisplayBoard`, `ParkingTicket`, and `ParkingSpot`.



The composition relationship between classes

Inheritance

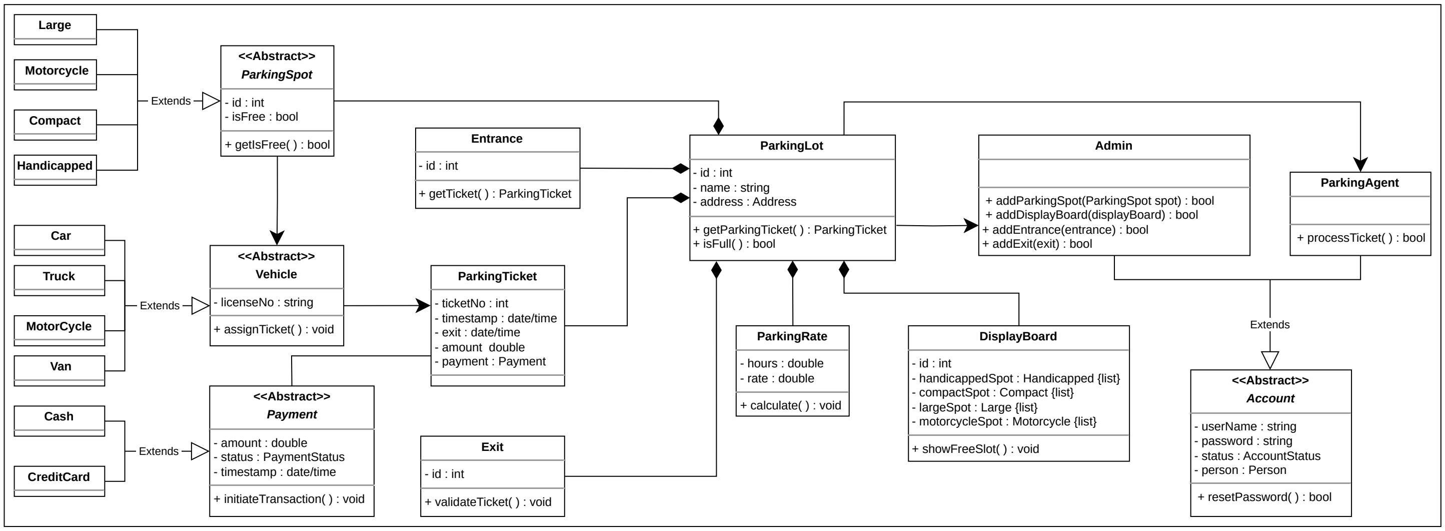
The following classes show an inheritance relationship:

- The `Vehicle` class includes `Car`, `Truck`, `Van`, and `MotorCycle` subclasses.
- The `ParkingSpot` class includes `handicapped`, `compact`, `large`, and `motorcycle` subclasses.
- The `Payment` class includes the `Cash` and `CreditCard` subclasses.

> **Note:** We have already discussed the inheritance relationship between classes in the component section above.

Class diagram of the parking lot system

Here is the complete class diagram for our parking lot system:



The class diagram of the parking lot system

Design pattern

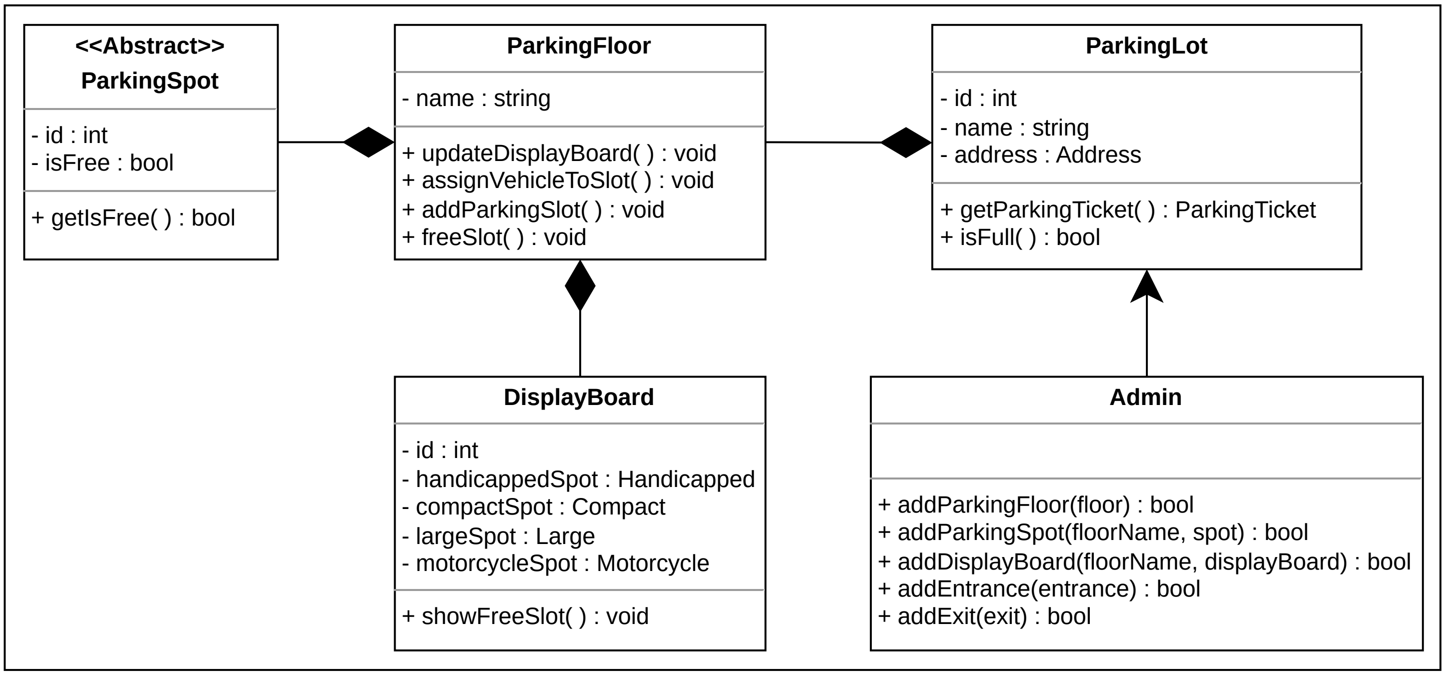
The system itself will have a `ParkingLot` class. It will use the Singleton design pattern, because there will only be a single instance of the parking lot system.

This parking lot system is also composed of smaller objects that we have already designed, like entrance, exit, parking spots, parking rates, etc. Therefore, it will be a good practice to use the Abstract Factory and Factory design pattern to instantiate all those objects.

Additional requirements

The interviewer can introduce some additional requirements in the parking lot system, or they can ask some follow-up questions. Let's see some examples of additional requirements:

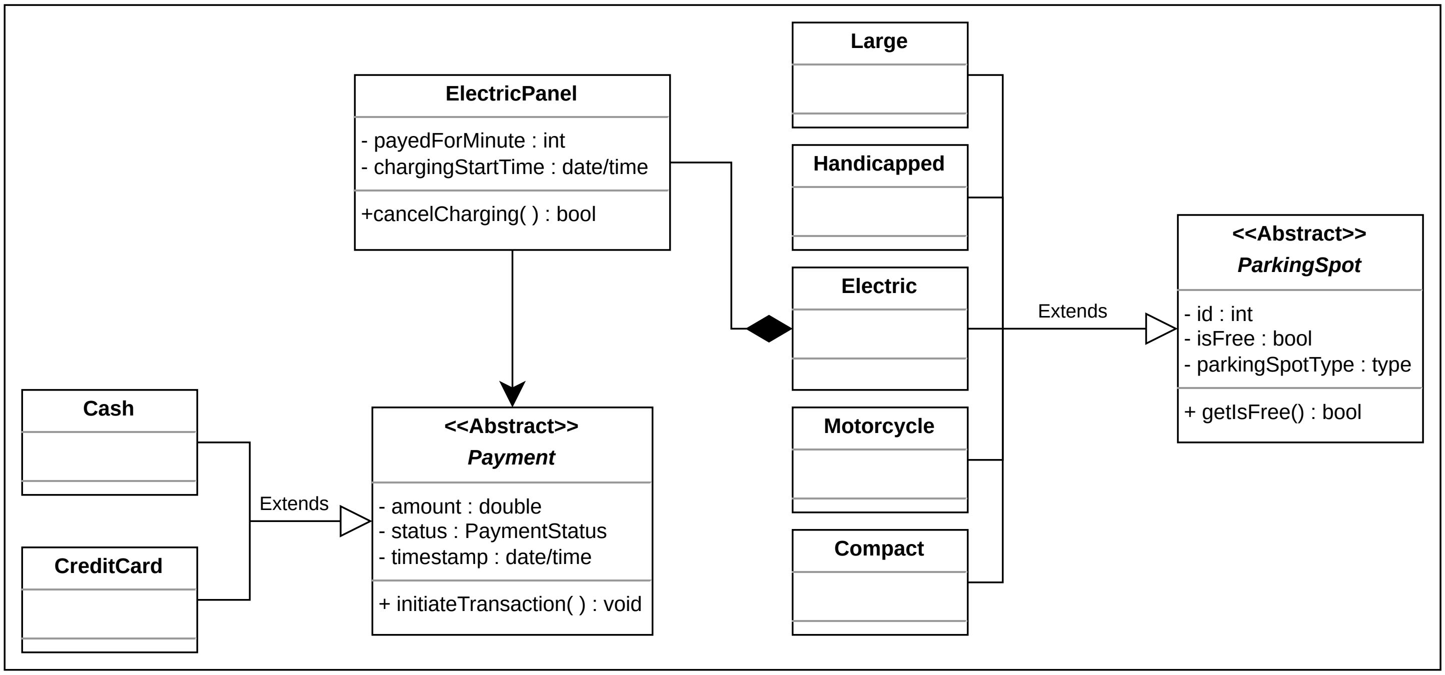
Parking floor: The parking lot should have multiple floors where customers can park their cars. The class diagram provided below shows the relationship of `ParkingFloor` with other classes:



<<Abstract>>
ParkingSpot
- id : int
- isFree : bool
+ getIsFree( ) : bool
ParkingFloor
- name : string
+ updateDisplayBoard( ) : void
+ assignVehicleToSlot( ) : void
+ addParkingSlot( ) : void
+ freeSlot( ) : void
ParkingLot
- id : int
- name : string
- address : Address
+ getParkingTicket( ) : ParkingTicket
+ isFull( ) : bool
DisplayBoard
- id : int
- handicappedSpot : Handicapped
- compactSpot : Compact
- largeSpot : Large
- motorcycleSpot : Motorcycle
+ showFreeSlot( ) : void
Admin
+ addParkingFloor(floor) : bool
+ addParkingSpot(floorName, spot) : bool
+ addDisplayBoard(floorName, displayBoard) : bool
+ addEntrance(entrance) : bool
+ addExit(exit) : bool

Relationship of the ParkingFloor class with other classes

Electric: The parking lot should have some parking spots specified for electric cars. These spots should have an electric panel through which customers can pay and charge their vehicles. The class diagram provided below shows the relationship of `Electric` and `ElectricPanel` with other classes:



Relationship of the Electric and ElectricPanel class with other classes

Question

Let's say that the interviewer asks you that the parking lot should assign a parking spot closest to the entrance. How do you go about solving this requirement?

Hide Answer

This requirement is more about how you implement this parking assignment strategy rather than designing it. The interviewer is really looking at your data structures and algorithms skills in this requirement.

In this scenario, let's say we have four entrances and would like to return to the parking spot which is nearest to the entrance from where the customer is entering the parking lot. The best approach is to implement it using a **min-heap.**

- We will declare four min-heaps. We will add all parking spots to these min-heaps, so there will be a min-heap for each entrance. These min-heaps will store the parking spots in the order of the shortest distance from the entrance.

- We will also declare the following two sets of parking spots:
    - A set of available parking spots
    - A set of reserved parking spots
- We have a map of min-heaps where the key is the entrance ID, and the value is a min-heap. When the user calls the `getParkingSpot` method, we get the entrance ID which gives us the min-heap for that entrance and allows us to pop the top element to get the parking spot.