Design a Parking Lot

Resources

- 6. Parking Lot, Low Level Design (Hindi) | SDE2 Design interview question | Design Parking Lot
- Code Repo → src/main/java/com/conceptcoding/interviewquestions/parkinglot · main · shrayansh jain / LLD-LowLevelDesign · GitLab

Problem Statement

Design a detailed low-level model for a parking lot system, including all relevant classes and design patterns covered earlier.

Overview

A parking lot is a designated area specifically allocated for vehicles to be parked when they are not in use. To ensure smooth and efficient entry into the parking lot, as well as convenient parking and easy exit, the entire system requires a careful and thoughtful design approach. A well-designed parking lot system should be extensible, meaning it can accommodate future expansions or changes; maintainable, so it can be easily managed and updated over time; and efficient, providing optimal use of space and resources. To achieve these important goals, we apply the principles, patterns, guidelines, and best practices that are outlined in the low-level design series.

A simple parking lot is typically an open area that features a single entry and exit gate, designed to manage the flow of vehicles efficiently. The layout of this parking lot is organised into distinct blocks, each designated for parking different types of vehicles, ensuring optimal use of space and convenience for users. When customers enter the parking lot, they are provided with a parking ticket, which serves as a record of their entry time and vehicle type. Upon exiting, customers pay a fee that is calculated based on the type of vehicle they parked, allowing for a fair and structured payment system.

This section explains how to design a simple parking lot.

Requirement Classification

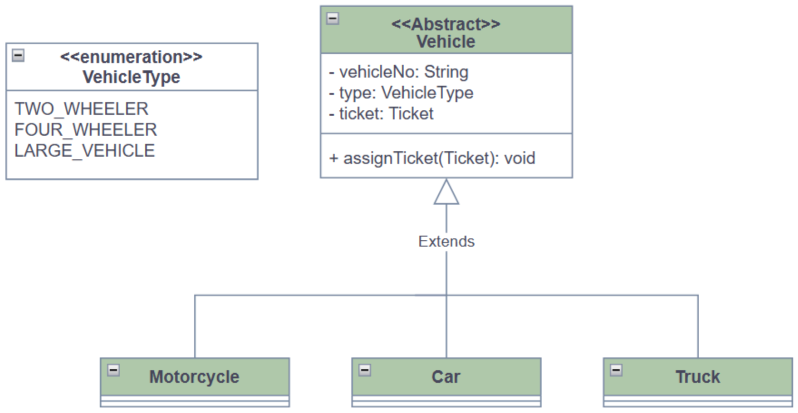
Requirement	Design Choice	Use case
Account → Admin, Agent, and Watchman, etc?	Admin only	Admin controls the entire management of the parking lot system, which includes issuing tickets, collecting payments, adding and removing parking spots and payment methods.
Entrances & Exits → Single entrance and exit or Multiple entrances and exits?	Single Entrance and Single Exit	A customer arrives at the entry gate, collects the parking ticket, and parks their vehicle in the assigned parking spot. A parking spot is assigned based on the vehicle type and is made available when the customer pays the fee and exits the venue.
Types of Parking Spots → 2-wheelers, 4-wheelers, spots for handicapped and large vehicles.	2-wheelers, 4-wheelers, spots for handicapped and large vehicles.	There is a fixed number of parking spots available for each category of vehicle types.

Parking Floors → Single or Multifloor?	Single	The entry and exit belong to a single-floor parking lot.
Cost Calculation → Hourly based charges or minute-based charges?	Mix	Certain parking spots will have hourly based charges, and others will be charged per-minute basis.
Payment Methods → Cash or CreditCard?	Both	The customer can choose to pay via cash or credit card.

Here, we use the Bottom Up Approach to design the Parking Lot System. It's a strategy in which we build the lower-level individual components first and connect them by establishing relationships between them, and gradually build up to a larger solution.

Components

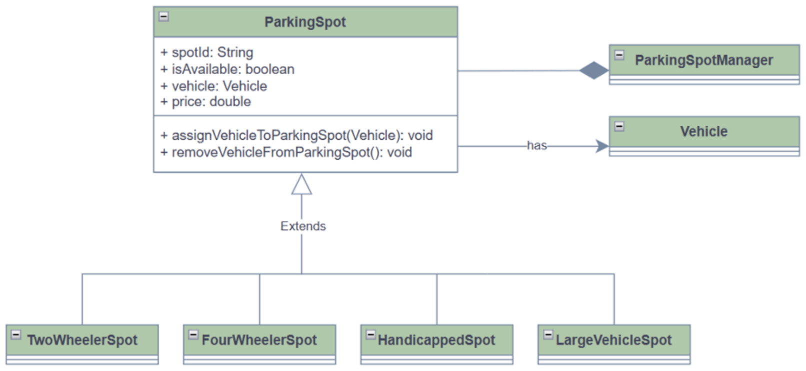
1. `Vehicle` and `VehicleType`



- All `Vehicle`s entering the parking lot can be broadly classified into a certain `VehicleType` (which is an enum).
- Each `Vehicle` has details about itself, such as its vehicle number and its type.

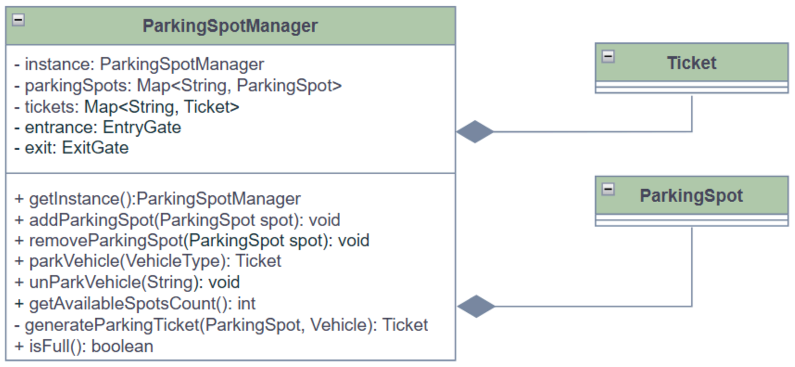
- Each `Vehicle` holds a reference to its parking `Ticket`. The assignment takes place when the `Ticket` is issued at the `EntryGate`.

2. `ParkingSpot`



- A parking lot has a fixed number of spots for various vehicle types. It could vary from 0 to N.
- Each `ParkingSpot` can be one of the following types: `TwoWheelerSpot`, `FourWheelerSpot`, `HandicappedSpot`, or `LargeVehicleSpot`.
- Derived classes will implement the price method specifically, as each parking spot type has a different price for the `VehicleType` parked in the spot.

3. `ParkingSpotManager`

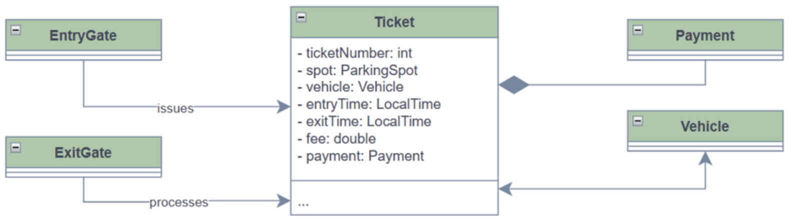


ParkingSpotManager
- instance: ParkingSpotManager
- parkingSpots: Map<String, ParkingSpot>
- tickets: Map<String, Ticket>
- entrance: EntryGate
- exit: ExitGate
+ getInstance():ParkingSpotManager
+ addParkingSpot(ParkingSpot spot): void
+ removeParkingSpot(ParkingSpot spot): void
+ parkVehicle(VehicleType): Ticket
+ unParkVehicle(String): void
+ getAvailableSpotsCount(): int
- generateParkingTicket(ParkingSpot, Vehicle): Ticket
+ isFull(): boolean
Ticket
ParkingSpot

- `ParkingSpotManager` is a singleton class. A single instance will orchestrate the entire parking management system.
- `ParkingSpotManager` maintains a map of `ParkingSpot`s across spot IDs.
- `ParkingSpotManager` maintains a map of parking `Ticket`s across vehicle numbers.

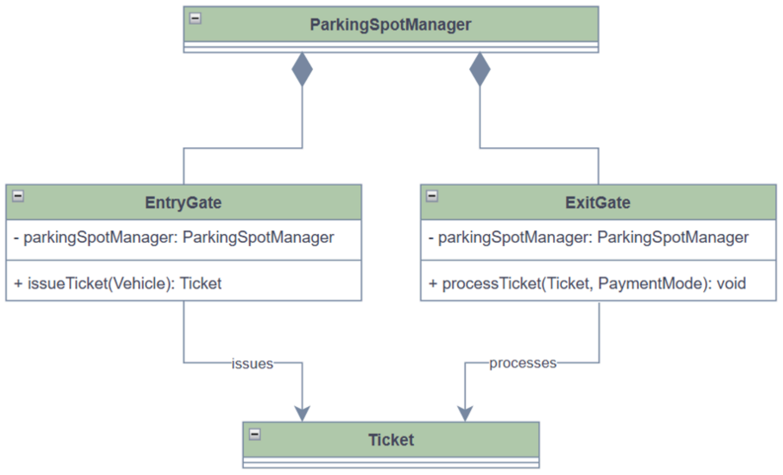
- `ParkingSpotManager` controls the parking of vehicles in the parking lot, which includes adding and removing parking spaces, checking if the parking lot is full before finding parking spaces, parking vehicles in designated spots, and freeing parking spots.

4. `Ticket`



- The `Ticket` includes entry time, exit time, ticket number, vehicle number, parking fee, and assigned parking spot.
- It also holds a reference to the `Payment` object for processing the payment.
- The `EntryGate` issues parking `Ticket`s.
- The ExitGate validates and processes the payment for Parking using the `Ticket` reference.
- A `Vehicle` is associated with a `Ticket`, and every `Ticket` is associated with a `Vehicle`.

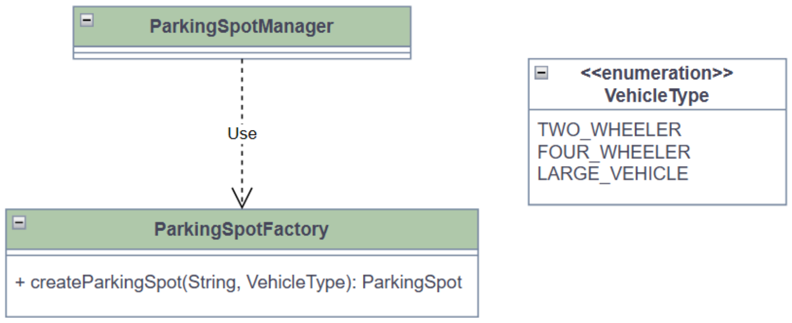
5. `EntryGate` and `ExitGate`


ParkingSpotManager
EntryGate
- parkingSpotManager: ParkingSpotManager
+ issueTicket(Vehicle): Ticket
ExitGate
- parkingSpotManager: ParkingSpotManager
+ processTicket(Ticket, PaymentMode): void
issues
processes
Ticket

- `ParkingSpotManager` is the orchestrator of the parking lot system and hence holds the reference to the entry and exit gates of the area, and if extended in future, it can be configured to add more entrances and exits based on traffic flow.

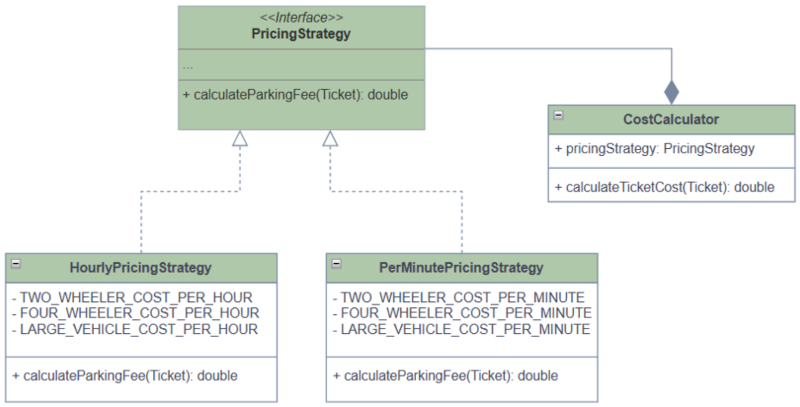
- `EntryGate` issues a parking `Ticket` to the customer, and `ExitGate` validates and processes the `Ticket` during the exit of the vehicle. Hence, the gates have a unidirectional association with the `Ticket` class.

6. `ParkingSpotFactory`



- `ParkingSpotFactory` is used to create parking spots by the Admin with specified IDs for each based on a specific `VehicleType`.
- The `ParkingSpotManager` uses the `ParkingSpot`s created by the `ParkingSpotFactory` and adds them to the internal map reference of all available `ParkingSpot`s.

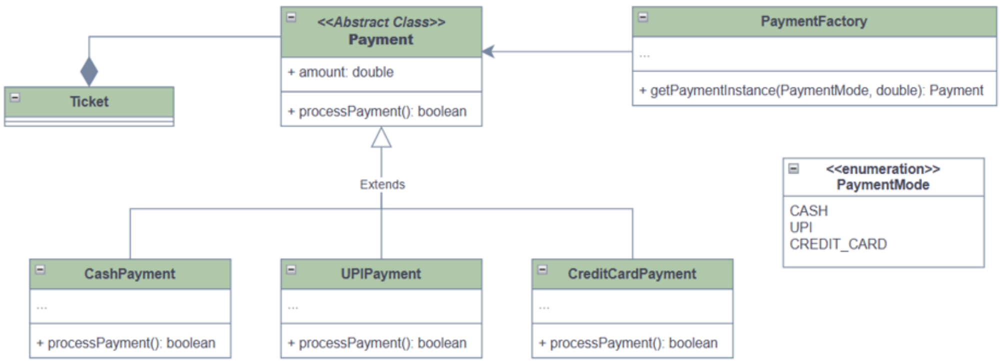
7. `CostCalculator` & `PricingStrategy`


<<Interface>>
PricingStrategy
...
+ calculateParkingFee(Ticket): double
CostCalculator
+ pricingStrategy: PricingStrategy
+ calculateTicketCost(Ticket): double
HourlyPricingStrategy
- TWO_WHEELER_COST_PER_HOUR
- FOUR_WHEELER_COST_PER_HOUR
- LARGE_VEHICLE_COST_PER_HOUR
+ calculateParkingFee(Ticket): double
PerMinutePricingStrategy
- TWO_WHEELER_COST_PER_MINUTE
- FOUR_WHEELER_COST_PER_MINUTE
- LARGE_VEHICLE_COST_PER_MINUTE
+ calculateParkingFee(Ticket): double

- `PricingStrategy` declares a construct `calculateParkingFee`, a method to calculate prices based on the concrete `PricingStrategy` adopted by the `ParkingSpotManager.CostCalculator`.

- `CostCalculator` holds a reference to a concrete `PricingStrategy`. Its method `calculateTicketCost()` calls the corresponding `PricingStrategy.calculateParkingFee()` to compute the total parking fee to be paid by the customer for the total parking duration.

8. **`Payment`, `PaymentMode` and `PaymentFactory`**



- The `Payment` abstract class defines a common construct used to pay parking fees using different modes of payment(Cash, UPI and Card).
- The concrete implementation of the `Payment` class, like `CashPayment`, `UPIPayment`, and `CreditCardPayment`, is responsible for processing the Payment.
- Each `Ticket` has a reference to the type of `Payment` method used to pay the parking fee calculated at the exit.
- `PaymentFactory` creates new payment modes at exit based on the customer's choice of `PaymentMode`.

Additional Features

- `ParkingStrategy`
  - `ParkingStrategy` helps customers choose between a few approaches to achieve a preferred or efficient parking space.
  - The current problem of designing a simple parking lot can be extended to have different Parking strategies, like `FirstAvailableSpotParkingStrategy`, `NearestToEntranceStrategy`, etc.

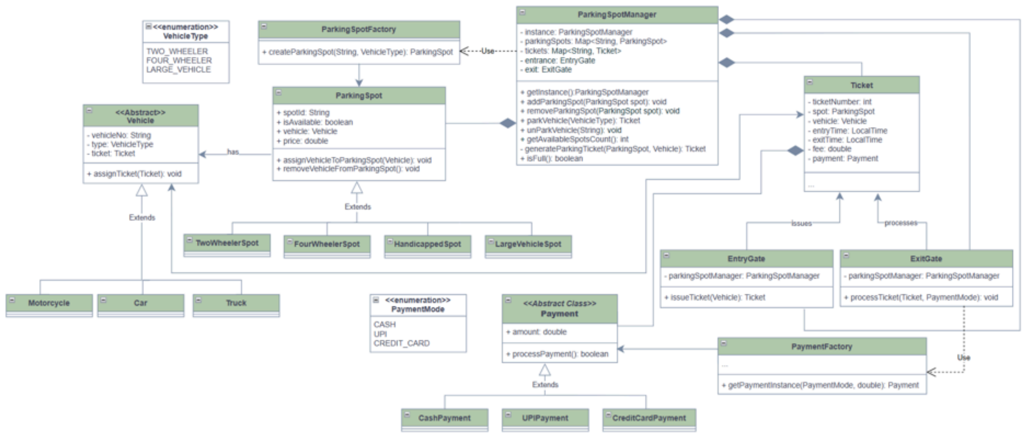
  - Min Heap data structures can be used to calculate the nearest parking spot (least distance from the entrance).
- `ParkingFloor`
  - Adding multiple floors to the parking lot design will require adding multiple entry and exit gates.
  - `ParkingSpotManager` will have to hold references to all gates present.
  - A display board with real-time updates of available parking spots would be required.

Design Patterns

- **Singleton:** `ParkingSpotManager`
- **Factory:** `ParkingSpotFactory` **and** `PaymentFactory`
- **Strategy:** `PricingStrategy`

Class Diagram

We combine the components above to produce a final solution to the Parking Lot design problem.



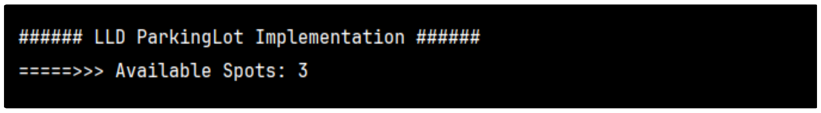
<<enumeration>> VehicleType
TWO_WHEELER
FOUR_WHEELER
LARGE_VEHICLE
ParkingSpotFactory
+ createParkingSpot(String, VehicleType): ParkingSpot
Use
ParkingSpotManager
- instance: ParkingSpotManager
- parkingSpots: Map<String, ParkingSpot>
- tickets: Map<String, Ticket>
- entrance: EntryGate
- exit: ExitGate
+ getInstance():ParkingSpotManager
+ addParkingSpot(ParkingSpot spot): void
+ removeParkingSpot(ParkingSpot spot): void
+ parkVehicle(VehicleType): Ticket
+ unParkVehicle(String): void
+ getAvailableSpotsCount(): int
- generateParkingTicket(ParkingSpot, Vehicle): Ticket
+ isFull(): boolean
<<Abstract>> Vehicle
- vehicleNo: String
- type: VehicleType
- ticket: Ticket
+ assignTicket(Ticket): void
has
ParkingSpot
+ spotId: String
+ isAvailable: boolean
+ vehicle: Vehicle
+ price: double
+ assignVehicleToParkingSpot(Vehicle): void
+ removeVehicleFromParkingSpot(): void
Ticket
- ticketNumber: int
- spot: ParkingSpot
- vehicle: Vehicle
- entryTime: LocalTime
- exitTime: LocalTime
- fee: double
- payment: Payment
...
Extends
Extends
TwoWheelerSpot
FourWheelerSpot
HandicappedSpot
LargeVehicleSpot
issues
processes
EntryGate
- parkingSpotManager: ParkingSpotManager
+ issueTicket(Vehicle): Ticket
ExitGate
- parkingSpotManager: ParkingSpotManager
+ processTicket(Ticket, PaymentMode): void
Motorcycle
Car
Truck
<<enumeration>> PaymentMode
CASH
UPI
CREDIT_CARD
<<Abstract Class>> Payment
+ amount: double
+ processPayment(): boolean
PaymentFactory
...
+ getPaymentInstance(PaymentMode, double): Payment
Use
Extends
CashPayment
UPIPayment
CreditCardPayment

Parking Lot System: Class Diagram

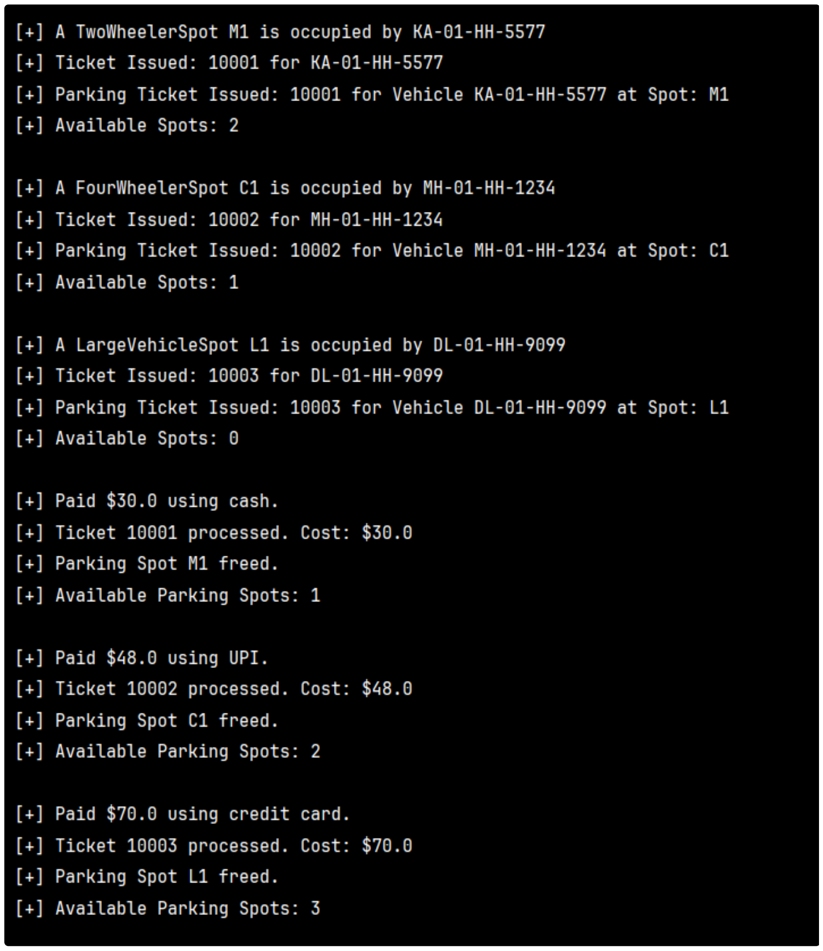
Implementation

Refer Code Repository for Executable Code → src/main/java/com/conceptcoding/interviewquestions/parkinglot · main · shrayansh jain / LLD-LowLevelDesign · GitLab

Output



```
###### LLD ParkingLot Implementation ######
=====>>> Available Spots: 3
```



```
[+] A TwoWheelerSpot M1 is occupied by KA-01-HH-5577
[+] Ticket Issued: 10001 for KA-01-HH-5577
[+] Parking Ticket Issued: 10001 for Vehicle KA-01-HH-5577 at Spot: M1
[+] Available Spots: 2

[+] A FourWheelerSpot C1 is occupied by MH-01-HH-1234
[+] Ticket Issued: 10002 for MH-01-HH-1234
[+] Parking Ticket Issued: 10002 for Vehicle MH-01-HH-1234 at Spot: C1
[+] Available Spots: 1

[+] A LargeVehicleSpot L1 is occupied by DL-01-HH-9099
[+] Ticket Issued: 10003 for DL-01-HH-9099
[+] Parking Ticket Issued: 10003 for Vehicle DL-01-HH-9099 at Spot: L1
[+] Available Spots: 0

[+] Paid $30.0 using cash.
[+] Ticket 10001 processed. Cost: $30.0
[+] Parking Spot M1 freed.
[+] Available Parking Spots: 1

[+] Paid $48.0 using UPI.
[+] Ticket 10002 processed. Cost: $48.0
[+] Parking Spot C1 freed.
[+] Available Parking Spots: 2

[+] Paid $70.0 using credit card.
[+] Ticket 10003 processed. Cost: $70.0
[+] Parking Spot L1 freed.
[+] Available Parking Spots: 3
```