Back To Course Home

Grokking the Object Oriented Design Interview

0% completed

Search Course

Object-Oriented Design and UML

- Object-Oriented Basics
- OO Analysis and Design
- What is UML?
- Use Case Diagrams
- Class Diagram
- Sequence diagram
- Activity Diagrams

Object Oriented Design Case Studies

- Design a Library Management System
- Design a Parking Lot
- Design Amazon - Online Shopping System
- Design Stack Overflow
- Design a Movie Ticket Booking System
- Design an ATM
- Design an Airline Management System
- Design Blackjack and a Deck of Cards
- Design a Hotel Management System
- Design a Restaurant Management system
- Design Chess
- Design an Online Stock Brokerage System
- Design a Car Rental System
- Design LinkedIn
- Design CricInfo
- Design Facebook - a social network

Appendix

- Contact us
- Other courses

Course Certificate

Mark Course as Completed

Design a Parking Lot

Let's make an object-oriented design for a multi-floor parking lot.

We'll cover the following

- System Requirements
- Use case diagram
- Class diagram
- Activity diagrams
- Code

A parking lot or car park is a dedicated cleared area that is intended for parking vehicles. In most countries where cars are a major mode of transportation, parking lots are a feature of every city and suburban area. Shopping malls, sports stadiums, megachurches, and similar venues often feature parking lots over large areas.

A Parking Lot

System Requirements

We will focus on the following set of requirements while designing the parking lot:

1. The parking lot should have multiple floors where customers can park their cars.
2. The parking lot should have multiple entry and exit points.
3. Customers can collect a parking ticket from the entry points and can pay the parking fee at the exit points on their way out.
4. Customers can pay the tickets at the automated exit panel or to the parking attendant.
5. Customers can pay via both cash and credit cards.
6. Customers should also be able to pay the parking fee at the customer's info portal on each floor. If the customer has paid at the info portal, they don't have to pay at the exit.
7. The system should not allow more vehicles than the maximum capacity of the parking lot. If the parking is full, the system should be able to show a message at the entrance panel and on the parking display board on the ground floor.
8. Each parking floor will have many parking spots. The system should support multiple types of parking spots such as Compact, Large, Handicapped, Motorcycle, etc.
9. 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.
10. The system should support parking for different types of vehicles like car, truck, van, motorcycle, etc.
11. Each parking floor should have a display board showing any free parking spot for each spot type.
12. The system should support a per-hour parking fee model. For example, customers have to pay $4 for the first hour; $3.5 for the second and third hours, and $2.5 for all the remaining hours.

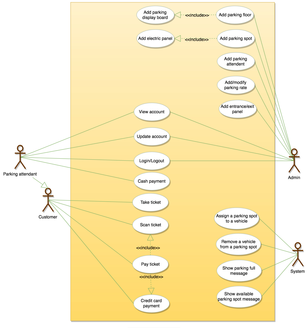
Use case diagram

Here are the main Actors in our system:

- **Admin:** Mainly responsible for adding and modifying parking floors, parking spots, entrance, and exit panels, adding/removing parking attendants, etc.
- **Customer:** All customers can get a parking ticket and pay for it.
- **Parking attendant:** Parking attendants can do all the activities on the customer's behalf, and can take cash for ticket payment.
- **System:** To display messages on different info panels, as well as assigning and removing a vehicle from a parking spot.

Here are the top use cases for Parking Lot:

- **Add/Remove/Edit parking floor:** To add, remove or modify a parking floor from the system. Each floor can have its own display board to show free parking spots.
- **Add/Remove/Edit parking spot:** To add, remove or modify a parking spot on a parking floor.
- **Add/Remove a parking attendant:** To add or remove a parking attendant from the system.
- **Take ticket:** To provide customers with a new parking ticket when entering the parking lot.
- **Scan ticket:** To scan a ticket to find out the total charge.
- **Credit card payment:** To pay the ticket fee with credit card.
- **Cash payment:** To pay the parking ticket through cash.
- **Add/Modify parking rate:** To allow admin to add or modify the hourly parking rate.

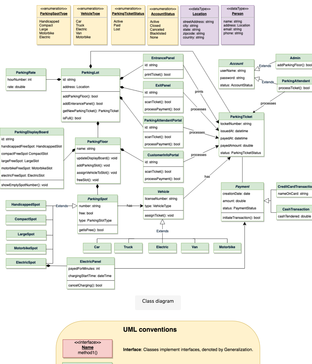


Use case diagram

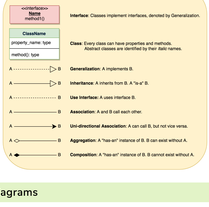
Class diagram

Here are the main classes of our Parking Lot System:

- **ParkingLot:** The central part of the organization for which this software has been designed. It has attributes like 'Name' to distinguish it from any other parking lots and 'Address' to define its location.
- **ParkingFloor:** The parking lot will have many parking floors.
- **ParkingSpot:** Each parking floor will have many parking spots. Our system will support different parking spots 1) Handicapped, 2) Compact, 3) Large, 4) Motorcycle, and 5) Electric.
- **Account:** We will have two types of accounts in the system: one for an Admin, and the other for a parking attendant.
- **Parking ticket:** This class will encapsulate a parking ticket. Customers will take a ticket when they enter the parking lot.
- **Vehicle:** Vehicles will be parked in the parking spots. Our system will support different types of vehicles 1) Car, 2) Truck, 3) Electric, 4) Van and 5) Motorcycle.
- **EntrancePanel and ExitPanel:** EntrancePanel will print tickets, and ExitPanel will facilitate payment of the ticket fee.
- **Payment:** This class will be responsible for making payments. The system will support credit card and cash transactions.
- **ParkingRate:** This class will keep track of the hourly parking rates. It will specify a dollar amount for each hour. For example, for a two hour parking ticket, this class will define the cost for the first and the second hour.
- **ParkingDisplayBoard:** Each parking floor will have a display board to show available parking spots for each spot type. This class will be responsible for displaying the latest availability of free parking spots to the customers.
- **ParkingAttendantPortal:** This class will encapsulate all the operations that an attendant can perform, like scanning tickets and processing payments.
- **CustomerInfoPortal:** This class will encapsulate the info portal that customers use to pay for the parking ticket. Once paid, the info portal will update the ticket to keep track of the payment.
- **ElectricPanel:** Customers will use the electric panels to pay and charge their electric vehicles.

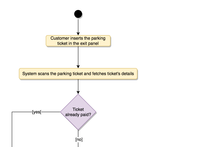


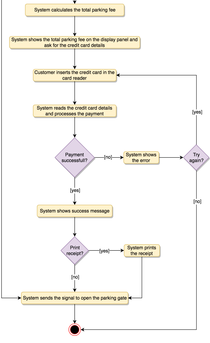
Class diagram



Activity diagrams

Customer paying for parking ticket: Any customer can perform this activity. Here are the set of steps:





Code

Following is the skeleton code for our parking lot system:

Enums and Constants: Here are the required enums, data types, and constants:

Java | Python

```java
public enum VehicleType {
  CAR, TRUCK, ELECTRIC, VAN, MOTORBIKE
}

public enum ParkingSpotType {
  HANDICAPPED, COMPACT, LARGE, MOTORBIKE, ELECTRIC
}

public enum AccountStatus {
  ACTIVE, BLOCKED, BANNED, COMMORMISED, ARCHIVED, UNKNOWN
}

public enum ParkingTicketStatus {
  ACTIVE, PAID, LOST
}

public class Address {
  private String streetAddress;
  private String city;
  private String state;
  private String zipCode;
  private String country;
}

public class Person {
  private String name;
  private Address address;
  private String email;
```

Account, Admin, and ParkingAttendant: These classes represent various people that interact with our system:

Java | Python

```java
// For simplicity, we are not defining getter and setter functions. The reader can
// assume that all class attributes are private and accessed through their respective
// public getter methods and modified only through their public methods function.

public abstract class Account {
  private String userName;
  private String password;
  private AccountStatus status;
  private Person person;

  public boolean resetPassword();
}

public class Admin extends Account {
  public bool addParkingFloor(ParkingFloor floor);
  public bool addParkingSpot(String floorName, ParkingSpot spot);
  public bool addParkingDisplayBoard(String floorName, ParkingDisplayBoard displayBoard);
  public bool addCustomerInfoPanel(String floorName, CustomerInfoPanel infoPanel);

  public bool addEntrancePanel(EntrancePanel entrancePanel);
  public bool addExitPanel(ExitPanel exitPanel);
}

public class ParkingAttendant extends Account {
  public bool processTicket(string TicketNumber);
}
```

ParkingSpot: Here is the definition of ParkingSpot and all of its children classes:

Java | Python

```java
public abstract class ParkingSpot {
  private String number;
  private boolean free;
  private Vehicle vehicle;
  private final ParkingSpotType type;

  public boolean IsFree();

  public ParkingSpot(ParkingSpotType type) {
    this.type = type;
  }

  public boolean assignVehicle(Vehicle vehicle) {
    this.vehicle = vehicle;
    free = false;
  }

  public boolean removeVehicle() {
    this.vehicle = null;
    free = true
  }
}

public class HandicappedSpot extends ParkingSpot {
  public HandicappedSpot() {
    super(ParkingSpotType.HANDICAPPED);
  }
}
```

Vehicle: Here is the definition for Vehicle and all of its child classes:

Java | Python

```java
public abstract class Vehicle {
  private String licenseNumber;
  private final VehicleType type;
  private ParkingTicket ticket;

  public Vehicle(VehicleType type) {
    this.type = type;
  }

  public void assignTicket(ParkingTicket ticket) {
    this.ticket = ticket;
  }
}

public class Car extends Vehicle {
  public Car() {
    super(VehicleType.CAR);
  }
}

public class Van extends Vehicle {
  public Van() {
    super(VehicleType.VAN);
  }
}

public class Truck extends Vehicle {
  public Truck() {
```

ParkingFloor: This class encapsulates a parking floor:

Java | Python

```java
public class ParkingFloor {
  private String name;
  private HashMap<String, HandicappedSpot> handicappedSpots;
  private HashMap<String, CompactSpot> compactSpots;
  private HashMap<String, LargeSpot> largeSpots;
  private HashMap<String, MotorbikeSpot> motorbikeSpots;
  private HashMap<String, ElectricSpot> electricSpots;
  private HashMap<String, CustomerInfoPortal> infoPortals;
  private ParkingDisplayBoard displayBoard;

  public ParkingFloor(String name) {
    this.name = name;
  }

  public void addParkingSpot(ParkingSpot spot) {
    switch (spot.getType()) {
    case ParkingSpotType.HANDICAPPED:
      handicappedSpots.put(spot.getNumber(), spot);
      break;
    case ParkingSpotType.COMPACT:
      compactSpots.put(spot.getNumber(), spot);
      break;
    case ParkingSpotType.LARGE:
      largeSpots.put(spot.getNumber(), spot);
      break;
    case ParkingSpotType.MOTORBIKE:
      motorbikeSpots.put(spot.getNumber(), spot);
      break;
```

ParkingDisplayBoard: This class encapsulates a parking display board:

Java | Python

```java
public class ParkingDisplayBoard {
  private String id;
  private HandicappedSpot handicappedFreeSpot;
  private CompactSpot compactFreeSpot;
  private LargeSpot largeFreeSpot;
  private MotorbikeSpot motorbikeFreeSpot;
  private ElectricSpot electricFreeSpot;

  public void showEmptySpotNumber() {
    String message = "";
    if(handicappedFreeSpot.IsFree()){
      message += "Free Handicapped: " + handicappedFreeSpot.getNumber();
    } else {
      message += "Handicapped is full";
    }
    message += System.lineSeparator();

    if(compactFreeSpot.IsFree()){
      message += "Free Compact: " + compactFreeSpot.getNumber();
    } else {
      message += "Compact is full";
    }
    message += System.lineSeparator();

    if(largeFreeSpot.IsFree()){
      message += "Free Large: " + largeFreeSpot.getNumber();
    } else {
      message += "Large is full";
```

ParkingLot: Our system will have only one object of this class. This can be enforced by using the Singleton pattern. In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to only one object.

Java | Python

```java
public class ParkingLot {
  private String name;
  private Location address;
  private ParkingRate parkingRate;

  private int compactSpotCount;
  private int largeSpotCount;
  private int motorbikeSpotCount;
  private int electricSpotCount;
  private final int maxCompactCount;
  private final int maxLargeCount;
  private final int maxMotorbikeCount;
  private final int maxElectricCount;

  private HashMap<String, EntrancePanel> entrancePanels;
  private HashMap<String, ExitPanel> exitPanels;
  private HashMap<String, ParkingFloor> parkingFloors;

  // all active parking tickets, identified by their ticketNumber
  private HashMap<String, ParkingTicket> activeTickets;

  // singleton ParkingLot to ensure only one object of ParkingLot in the system,
  // all entrance panels will use this object to create new parking ticket: getNewParkingTicket()
  // similarly exit panels will also use this object to close parking tickets
  private static ParkingLot parkingLot = null;

  // private constructor to restrict for singleton
  private ParkingLot() {
```

Back
Design a Library Management System

Next
Design Amazon - Online Shopping Sy...

Mark as Completed

Report an Issue
Ask a Question