Low-Level Design (LLD) for Parking Lot System

I'll explain the complete low-level design for a parking lot system with detailed Java implementation.

---

1. Requirements Gathering

Functional Requirements:

1. **Multiple floors with multiple parking spots per floor**
2. **Different types of parking spots (Compact, Large, Handicapped, Motorcycle)**
3. **Different types of vehicles (Car, Truck, Van, Motorcycle)**
4. **Entry and exit panels at each floor**
5. **Payment system with hourly rates**
6. **Parking spot allocation strategy**
7. **Parking ticket generation**
8. **Display available spots**

Non-Functional Requirements:

1. **System should be scalable**
2. **High availability**
3. **Thread-safe for concurrent operations**

---

2. Core Entities and Classes

Class Diagram Overview:

ParkingLot (Singleton)

├── ParkingFloor (multiple)

│ ├── ParkingSpot (multiple)

│ ├── EntryPanel

│ └── ExitPanel

├── Vehicle (abstract)

│ ├── Car

│ ├── Truck

│ ├── Van

│ └── Motorcycle

├── ParkingTicket

├── Payment (abstract)

│ ├── CashPayment

│ └── CardPayment

└── ParkingSpotManager

---

3. Detailed Implementation

Step 1: Enums and Constants

java

```
// Enum for Vehicle Types
enum VehicleType {
    CAR,
    TRUCK,
    VAN,
    MOTORCYCLE
}
```

```
// Enum for Parking Spot Types
enum ParkingSpotType {
    COMPACT,
    LARGE,
    HANDICAPPED,
    MOTORCYCLE
}

// Enum for Parking Spot Status
enum ParkingSpotStatus {
    AVAILABLE,
    OCCUPIED,
    RESERVED
}

// Enum for Payment Status
enum PaymentStatus {
    PENDING,
    COMPLETED,
    FAILED,
    REFUNDED
}
```

```
// Enum for Payment Method
enum PaymentMethod {
    CASH,
    CREDIT_CARD,
    DEBIT_CARD,
    UPI
}
Step 2: Vehicle Classes
java
// Abstract Vehicle class
abstract class Vehicle {
    private String licensePlate;
    private VehicleType vehicleType;

    public Vehicle(String licensePlate, VehicleType vehicleType) {
        this.licensePlate = licensePlate;
        this.vehicleType = vehicleType;
    }

    public String getLicensePlate() {
        return licensePlate;
    }
```

```
    public VehicleType getVehicleType() {
        return vehicleType;
    }
}

// Concrete Vehicle classes
class Car extends Vehicle {
    public Car(String licensePlate) {
        super(licensePlate, VehicleType.CAR);
    }
}

class Truck extends Vehicle {
    public Truck(String licensePlate) {
        super(licensePlate, VehicleType.TRUCK);
    }
}

class Van extends Vehicle {
    public Van(String licensePlate) {
        super(licensePlate, VehicleType.VAN);
    }
```

```
}

class Motorcycle extends Vehicle {
    public Motorcycle(String licensePlate) {
        super(licensePlate, VehicleType.MOTORCYCLE);
    }
}
```

Step 3: Parking Spot Classes

java

```
// Abstract ParkingSpot class
abstract class ParkingSpot {
    private String spotId;
    private ParkingSpotType spotType;
    private ParkingSpotStatus status;
    private Vehicle vehicle;

    public ParkingSpot(String spotId, ParkingSpotType spotType) {
        this.spotId = spotId;
        this.spotType = spotType;
        this.status = ParkingSpotStatus.AVAILABLE;
        this.vehicle = null;
    }
```

```
public synchronized boolean isAvailable() {
    return status == ParkingSpotStatus.AVAILABLE;
}

public synchronized boolean assignVehicle(Vehicle vehicle) {
    if (isAvailable() && canFitVehicle(vehicle)) {
        this.vehicle = vehicle;
        this.status = ParkingSpotStatus.OCCUPIED;
        return true;
    }
    return false;
}

public synchronized boolean removeVehicle() {
    if (status == ParkingSpotStatus.OCCUPIED) {
        this.vehicle = null;
        this.status = ParkingSpotStatus.AVAILABLE;
        return true;
    }
    return false;
}

// Abstract method to check if vehicle can fit
```

```
    protected abstract boolean canFitVehicle(Vehicle vehicle);

    // Getters
    public String getSpotId() { return spotId; }
    public ParkingSpotType getSpotType() { return spotType; }
    public ParkingSpotStatus getStatus() { return status; }
    public Vehicle getVehicle() { return vehicle; }
}

// Concrete Parking Spot classes
class CompactSpot extends ParkingSpot {
    public CompactSpot(String spotId) {
        super(spotId, ParkingSpotType.COMPACT);
    }

    @Override
    protected boolean canFitVehicle(Vehicle vehicle) {
        // Compact spots can fit motorcycles and cars
        return vehicle.getVehicleType() == VehicleType.MOTORCYCLE ||
               vehicle.getVehicleType() == VehicleType.CAR;
    }
}
```

```
class LargeSpot extends ParkingSpot {
    public LargeSpot(String spotId) {
        super(spotId, ParkingSpotType.LARGE);
    }

    @Override
    protected boolean canFitVehicle(Vehicle vehicle) {
        // Large spots can fit any vehicle
        return true;
    }
}

class HandicappedSpot extends ParkingSpot {
    public HandicappedSpot(String spotId) {
        super(spotId, ParkingSpotType.HANDICAPPED);
    }

    @Override
    protected boolean canFitVehicle(Vehicle vehicle) {
        // Handicapped spots can fit cars
        return vehicle.getVehicleType() == VehicleType.CAR;
    }
}
```

```
class MotorcycleSpot extends ParkingSpot {
    public MotorcycleSpot(String spotId) {
        super(spotId, ParkingSpotType.MOTORCYCLE);
    }

    @Override
    protected boolean canFitVehicle(Vehicle vehicle) {
        // Motorcycle spots only fit motorcycles
        return vehicle.getVehicleType() == VehicleType.MOTORCYCLE;
    }
}
```

Step 4: Parking Ticket

```
java
import java.time.LocalDateTime;
import java.time.Duration;

class ParkingTicket {
    private String ticketId;
    private Vehicle vehicle;
    private ParkingSpot parkingSpot;
    private LocalDateTime entryTime;
    private LocalDateTime exitTime;
```

```
private double amount;
private PaymentStatus paymentStatus;

public ParkingTicket(String ticketId, Vehicle vehicle, ParkingSpot parkingSpot) {
    this.ticketId = ticketId;
    this.vehicle = vehicle;
    this.parkingSpot = parkingSpot;
    this.entryTime = LocalDateTime.now();
    this.paymentStatus = PaymentStatus.PENDING;
}

public void markExit() {
    this.exitTime = LocalDateTime.now();
    this.amount = calculateAmount();
}

private double calculateAmount() {
    if (exitTime == null) {
        return 0.0;
    }

    // Calculate parking duration in hours
    Duration duration = Duration.between(entryTime, exitTime);
```

```
    long hours = duration.toHours();

    // If less than 1 hour, charge for 1 hour
    if (hours < 1) {
        hours = 1;
    }

    // Rate based on vehicle type
    double hourlyRate = getHourlyRate();
    return hours * hourlyRate;
}

private double getHourlyRate() {
    switch (vehicle.getVehicleType()) {
        case MOTORCYCLE:
            return 5.0;
        case CAR:
            return 10.0;
        case VAN:
            return 15.0;
        case TRUCK:
            return 20.0;
        default:
```

```
            return 10.0;
        }
    }

    public void updatePaymentStatus(PaymentStatus status) {
        this.paymentStatus = status;
    }

    // Getters
    public String getTicketId() { return ticketId; }
    public Vehicle getVehicle() { return vehicle; }
    public ParkingSpot getParkingSpot() { return parkingSpot; }
    public LocalDateTime getEntryTime() { return entryTime; }
    public LocalDateTime getExitTime() { return exitTime; }
    public double getAmount() { return amount; }
    public PaymentStatus getPaymentStatus() { return paymentStatus; }
}
```

Step 5: Payment Classes

java

```
// Abstract Payment class
abstract class Payment {
    private String paymentId;
    private double amount;
```

```
private PaymentStatus status;
private PaymentMethod method;
private LocalDateTime paymentTime;

public Payment(String paymentId, double amount, PaymentMethod method) {
    this.paymentId = paymentId;
    this.amount = amount;
    this.method = method;
    this.status = PaymentStatus.PENDING;
}

public abstract boolean processPayment();

protected void markCompleted() {
    this.status = PaymentStatus.COMPLETED;
    this.paymentTime = LocalDateTime.now();
}

protected void markFailed() {
    this.status = PaymentStatus.FAILED;
}

// Getters
```

```
    public String getPaymentId() { return paymentId; }
    public double getAmount() { return amount; }
    public PaymentStatus getStatus() { return status; }
    public PaymentMethod getMethod() { return method; }
}

// Concrete Payment classes
class CashPayment extends Payment {
    private double cashReceived;

    public CashPayment(String paymentId, double amount, double cashReceived) {
        super(paymentId, amount, PaymentMethod.CASH);
        this.cashReceived = cashReceived;
    }

    @Override
    public boolean processPayment() {
        if (cashReceived >= getAmount()) {
            markCompleted();
            System.out.println("Cash payment processed. Change: $" +
                               (cashReceived - getAmount()));
            return true;
        }
```

```
            markFailed();
            System.out.println("Insufficient cash!");
            return false;
        }
    }

class CardPayment extends Payment {
    private String cardNumber;
    private String cvv;

    public CardPayment(String paymentId, double amount,
                       PaymentMethod method, String cardNumber, String cvv) {
        super(paymentId, amount, method);
        this.cardNumber = cardNumber;
        this.cvv = cvv;
    }

    @Override
    public boolean processPayment() {
        // Simulate card processing
        if (validateCard()) {
            markCompleted();
            System.out.println("Card payment processed successfully!");
```

```
            return true;
        }
        markFailed();
        System.out.println("Card payment failed!");
        return false;
    }

    private boolean validateCard() {
        // Simulate card validation
        return cardNumber != null && cvv != null && cvv.length() == 3;
    }
}
```

Step 6: Parking Floor

```
java
import java.util.*;

class ParkingFloor {
    private int floorNumber;
    private List<ParkingSpot> parkingSpots;
    private Map<ParkingSpotType, List<ParkingSpot>> spotsByType;

    public ParkingFloor(int floorNumber) {
        this.floorNumber = floorNumber;
```

```
    this.parkingSpots = new ArrayList<>();
    this.spotsByType = new HashMap<>();

    // Initialize spotsByType map
    for (ParkingSpotType type : ParkingSpotType.values()) {
        spotsByType.put(type, new ArrayList<>());
    }
}

public void addParkingSpot(ParkingSpot spot) {
    parkingSpots.add(spot);
    spotsByType.get(spot.getSpotType()).add(spot);
}

public synchronized ParkingSpot findAvailableSpot(Vehicle vehicle) {
    // Strategy: Try to find the most appropriate spot type first
    List<ParkingSpotType> preferredTypes = getPreferredSpotTypes(vehicle);

    for (ParkingSpotType type : preferredTypes) {
        for (ParkingSpot spot : spotsByType.get(type)) {
            if (spot.isAvailable() && spot.assignVehicle(vehicle)) {
                return spot;
            }
```

```
        }
    }
    return null;
}

private List<ParkingSpotType> getPreferredSpotTypes(Vehicle vehicle) {
    List<ParkingSpotType> types = new ArrayList<>();

    switch (vehicle.getVehicleType()) {
        case MOTORCYCLE:
            types.add(ParkingSpotType.MOTORCYCLE);
            types.add(ParkingSpotType.COMPACT);
            types.add(ParkingSpotType.LARGE);
            break;
        case CAR:
            types.add(ParkingSpotType.COMPACT);
            types.add(ParkingSpotType.LARGE);
            break;
        case VAN:
        case TRUCK:
            types.add(ParkingSpotType.LARGE);
            break;
    }
```

```
    return types;
}

public Map<ParkingSpotType, Integer> getAvailableSpotCount() {
    Map<ParkingSpotType, Integer> counts = new HashMap<>();

    for (ParkingSpotType type : ParkingSpotType.values()) {
        int count = 0;
        for (ParkingSpot spot : spotsByType.get(type)) {
            if (spot.isAvailable()) {
                count++;
            }
        }
        counts.put(type, count);
    }

    return counts;
}

public int getFloorNumber() {
    return floorNumber;
}
```

```
    public List<ParkingSpot> getParkingSpots() {
        return parkingSpots;
    }
}
```

Step 7: Entry and Exit Panels

java

```
class EntryPanel {
    private String panelId;
    private int floorNumber;

    public EntryPanel(String panelId, int floorNumber) {
        this.panelId = panelId;
        this.floorNumber = floorNumber;
    }

    public ParkingTicket generateTicket(Vehicle vehicle, ParkingSpot spot) {
        String ticketId = "TKT-" + System.currentTimeMillis();
        ParkingTicket ticket = new ParkingTicket(ticketId, vehicle, spot);

        System.out.println("=== PARKING TICKET ===");
        System.out.println("Ticket ID: " + ticketId);
        System.out.println("Vehicle: " + vehicle.getLicensePlate());
```

```
        System.out.println("Spot: " + spot.getSpotId());
        System.out.println("Entry Time: " + ticket.getEntryTime());
        System.out.println("======================");

        return ticket;
    }

    public String getPanelId() { return panelId; }
    public int getFloorNumber() { return floorNumber; }
}

class ExitPanel {
    private String panelId;
    private int floorNumber;

    public ExitPanel(String panelId, int floorNumber) {
        this.panelId = panelId;
        this.floorNumber = floorNumber;
    }

    public boolean processExit(ParkingTicket ticket) {
        ticket.markExit();
```

```
    System.out.println("\n=== EXIT SUMMARY ===");
    System.out.println("Ticket ID: " + ticket.getTicketId());
    System.out.println("Vehicle: " + ticket.getVehicle().getLicensePlate());
    System.out.println("Entry Time: " + ticket.getEntryTime());
    System.out.println("Exit Time: " + ticket.getExitTime());
    System.out.println("Amount Due: $" + ticket.getAmount());
    System.out.println("====================\n");

    return true;
}

public boolean processPayment(ParkingTicket ticket, Payment payment) {
    if (payment.processPayment()) {
        ticket.updatePaymentStatus(PaymentStatus.COMPLETED);
        ticket.getParkingSpot().removeVehicle();
        System.out.println("Payment successful! You may exit now.");
        return true;
    }
    System.out.println("Payment failed! Please try again.");
    return false;
}

public String getPanelId() { return panelId; }
```

```
    public int getFloorNumber() { return floorNumber; }
}
```

Step 8: Parking Lot (Singleton)

```
java
import java.util.*;
import java.util.concurrent.ConcurrentHashMap;

class ParkingLot {
    private static ParkingLot instance;
    private String name;
    private String address;
    private List<ParkingFloor> floors;
    private Map<String, ParkingTicket> activeTickets;

    // Private constructor for Singleton
    private ParkingLot(String name, String address) {
        this.name = name;
        this.address = address;
        this.floors = new ArrayList<>();
        this.activeTickets = new ConcurrentHashMap<>();
    }

    // Singleton getInstance method
```

```
public static synchronized ParkingLot getInstance(String name, String address) {
    if (instance == null) {
        instance = new ParkingLot(name, address);
    }
    return instance;
}

public void addFloor(ParkingFloor floor) {
    floors.add(floor);
}

public synchronized ParkingTicket parkVehicle(Vehicle vehicle) {
    // Find available spot across all floors
    for (ParkingFloor floor : floors) {
        ParkingSpot spot = floor.findAvailableSpot(vehicle);
        if (spot != null) {
            // Generate ticket from entry panel
            EntryPanel entryPanel = new EntryPanel(
                "ENTRY-" + floor.getFloorNumber(),
                floor.getFloorNumber()
            );
            ParkingTicket ticket = entryPanel.generateTicket(vehicle, spot);
            activeTickets.put(ticket.getTicketId(), ticket);
```

```
            System.out.println("Vehicle parked successfully on Floor " +
                    floor.getFloorNumber());
            return ticket;
        }
    }

    System.out.println("Sorry! Parking lot is full.");
    return null;
}

public synchronized boolean unparkVehicle(String ticketId, Payment payment) {
    ParkingTicket ticket = activeTickets.get(ticketId);

    if (ticket == null) {
        System.out.println("Invalid ticket!");
        return false;
    }

    // Process exit
    ExitPanel exitPanel = new ExitPanel("EXIT-1", 1);
    exitPanel.processExit(ticket);
```

```
    // Process payment
    if (exitPanel.processPayment(ticket, payment)) {
        activeTickets.remove(ticketId);
        return true;
    }

    return false;
}

public void displayAvailability() {
    System.out.println("\n=== PARKING LOT AVAILABILITY ===");
    System.out.println("Parking Lot: " + name);

    for (ParkingFloor floor : floors) {
        System.out.println("\nFloor " + floor.getFloorNumber() + ":");
        Map<ParkingSpotType, Integer> counts = floor.getAvailableSpotCount();

        for (Map.Entry<ParkingSpotType, Integer> entry : counts.entrySet()) {
            System.out.println("  " + entry.getKey() + ": " +
                               entry.getValue() + " spots available");
        }
    }
    System.out.println("================================\n");
```

```
    }

    public ParkingTicket getTicket(String ticketId) {
        return activeTickets.get(ticketId);
    }
}
```

Step 9: Demo and Testing

java

```
public class ParkingLotDemo {
    public static void main(String[] args) {
        // Initialize Parking Lot
        ParkingLot parkingLot = ParkingLot.getInstance(
            "Downtown Parking",
            "123 Main Street"
        );

        // Create floors and spots
        setupParkingLot(parkingLot);

        // Display initial availability
        parkingLot.displayAvailability();

        // Test Case 1: Park a car
```

```
System.out.println("--- Test Case 1: Park a Car ---");
Vehicle car1 = new Car("ABC-1234");
ParkingTicket ticket1 = parkingLot.parkVehicle(car1);

// Test Case 2: Park a motorcycle
System.out.println("\n--- Test Case 2: Park a Motorcycle ---");
Vehicle bike1 = new Motorcycle("XYZ-5678");
ParkingTicket ticket2 = parkingLot.parkVehicle(bike1);

// Test Case 3: Park a truck
System.out.println("\n--- Test Case 3: Park a Truck ---");
Vehicle truck1 = new Truck("TRK-9999");
ParkingTicket ticket3 = parkingLot.parkVehicle(truck1);

// Display availability after parking
parkingLot.displayAvailability();

// Simulate some time passing (for testing payment calculation)
simulateTimeDelay();

// Test Case 4: Unpark the car with cash payment
System.out.println("\n--- Test Case 4: Unpark Car (Cash) ---");
if (ticket1 != null) {
```

```
    Payment cashPayment = new CashPayment(
        "PAY-" + System.currentTimeMillis(),
        ticket1.getAmount(),
        50.0
    );
    parkingLot.unparkVehicle(ticket1.getTicketId(), cashPayment);
}

// Test Case 5: Unpark motorcycle with card payment
System.out.println("\n--- Test Case 5: Unpark Motorcycle (Card) ---");
if (ticket2 != null) {
    Payment cardPayment = new CardPayment(
        "PAY-" + System.currentTimeMillis(),
        ticket2.getAmount(),
        PaymentMethod.CREDIT_CARD,
        "1234-5678-9012-3456",
        "123"
    );
    parkingLot.unparkVehicle(ticket2.getTicketId(), cardPayment);
}

// Display final availability
parkingLot.displayAvailability();
```

```
}

private static void setupParkingLot(ParkingLot parkingLot) {
    // Create 3 floors
    for (int i = 1; i <= 3; i++) {
        ParkingFloor floor = new ParkingFloor(i);

        // Add different types of spots to each floor
        // 10 Compact spots
        for (int j = 1; j <= 10; j++) {
            floor.addParkingSpot(
                new CompactSpot("F" + i + "-C" + j)
            );
        }

        // 10 Large spots
        for (int j = 1; j <= 10; j++) {
            floor.addParkingSpot(
                new LargeSpot("F" + i + "-L" + j)
            );
        }

        // 5 Motorcycle spots
```

```
        for (int j = 1; j <= 5; j++) {
            floor.addParkingSpot(
                new MotorcycleSpot("F" + i + "-M" + j)
            );
        }

        // 2 Handicapped spots
        for (int j = 1; j <= 2; j++) {
            floor.addParkingSpot(
                new HandicappedSpot("F" + i + "-H" + j)
            );
        }

        parkingLot.addFloor(floor);
    }
}

private static void simulateTimeDelay() {
    try {
        Thread.sleep(2000); // Wait 2 seconds
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

```
    }
}
```

4. Key Design Principles Used

1. SOLID Principles

- **Single Responsibility**: Each class has one reason to change
- **Open/Closed**: Open for extension (new vehicle/spot types) but closed for modification
- **Liskov Substitution**: Subclasses can replace parent classes
- **Interface Segregation**: Specific interfaces for specific needs
- **Dependency Inversion**: Depend on abstractions, not concrete classes

2. Design Patterns

- **Singleton Pattern**: ParkingLot has only one instance
- **Factory Pattern**: Could be added for creating different spot/vehicle types
- **Strategy Pattern**: Different payment methods
- **Template Method**: Payment processing

3. Thread Safety

- Synchronized methods for concurrent access
- ConcurrentHashMap for active tickets

---

5. Extensibility Points

The design can be easily extended to add:

1. **Reservation System**: Add a reservation class
2. **Electric Vehicle Charging**: Add EVChargingSpot
3. **Monthly Pass**: Add subscription-based parking

4. **Admin Dashboard**: Add admin operations
5. **Notification System**: SMS/Email notifications
6. **Parking History**: Track all parking events