14. Elevator System

Here's a comprehensive elevator system design following the structured approach you requested:

Step 1: Outline Use Cases and Constraints

Use Cases (In-Scope)

1. Call Elevator from a Floor: A user presses an up or down button to call the elevator.
2. Select Destination Floor: Once inside, a user selects a destination floor.
3. Elevator Movement: The elevator moves to the requested floors in an optimized manner.
4. Door Operation: The doors open and close at the requested floors.
5. Emergency Handling: Handle power failures, emergency stops, and overload conditions.

Out of Scope

- Multi-elevator coordination
- Advanced scheduling algorithms (e.g., AI-based optimizations)

Constraints and Assumptions

- The elevator serves a single building with N floors.
- The elevator follows a FIFO (First-In, First-Out) request processing strategy.
- Doors remain open for 5 seconds before closing.
- The elevator will not move if the door is open.
- In case of power failure, the elevator stops at the nearest floor.

Step 2: High-Level Design

Core Components

1. Elevator System
    - Manages elevator requests and assigns them to an available elevator.
2. Elevator
    - Moves up/down based on requests.
    - Controls doors and handles emergencies.
3. User Interface
    - Users request an elevator or select a floor inside the cabin.
4. Sensors & Actuators
    - Floor sensors, weight sensors, door sensors.
    - Actuators for motor and door control.

Step 3: Design Core Components (According to Use Cases)

1. **Elevator System Class**
    - Manages multiple elevators.
    - Dispatches requests to an idle or best-fit elevator.
2. **Elevator Class**
    - Represents a single elevator.
    - Moves to requested floors.
    - Controls door operations and direction.
3. **Request Handling**
    - Manages requests from inside and outside the elevator.
4. **Emergency Handling**
    - Handles overload, power failure, and emergency stops.

Step 4: Scale the Design

- Multiple Elevators: Implement a scheduling algorithm to assign requests to elevators.
- Optimized Request Handling: Implement Nearest-Car Scheduling or AI-based routing.
- Event-Based System: Use message queues (e.g., Kafka) for event-driven communication.

Java Implementation

```
import java.util.*;

// Enum representing the possible directions of the elevator
enum Direction { UP, DOWN, IDLE }

// Elevator class representing an individual elevator
class Elevator {
  private int currentFloor; // Tracks the current floor of the elevator
  private Direction direction; // Tracks the movement direction of the elevator
  private boolean doorOpen; // Indicates if the elevator doors are open
  private PriorityQueue<Integer> upRequests; // Stores upward floor requests
  private PriorityQueue<Integer> downRequests; // Stores downward floor requests

  public Elevator() {
    this.currentFloor = 0;
    this.direction = Direction.IDLE;
    this.doorOpen = false;
    this.upRequests = new PriorityQueue<>(); // Min-heap for ascending order requests
    this.downRequests = new PriorityQueue<>(Collections.reverseOrder());
    // Max-heap for descending order requests
  }
```

```java
// Handles an external request to call the elevator
public void callElevator(int floor) {
    if (floor > currentFloor) {
        upRequests.add(floor);
    } else {
        downRequests.add(floor);
    }
    updateDirection();
}

// Handles an internal request to go to a specific floor
public void selectFloor(int floor) {
    if (floor > currentFloor) {
        upRequests.add(floor);
    } else {
        downRequests.add(floor);
    }
    updateDirection();
}

// Moves the elevator based on the pending requests
public void moveElevator() {
    while (!upRequests.isEmpty() || !downRequests.isEmpty()) {
        if (direction == Direction.UP && !upRequests.isEmpty()) {
            moveToFloor(upRequests.poll()); // Process upward requests
        } else if (direction == Direction.DOWN && !downRequests.isEmpty()) {
            moveToFloor(downRequests.poll()); // Process downward requests
        }
        updateDirection();
    }
    direction = Direction.IDLE; // Set to idle when no requests remain
}
```

```java
// Moves the elevator to the specified floor
private void moveToFloor(int floor) {
    while (currentFloor != floor) {
        if (currentFloor < floor) {
            currentFloor++;
        } else {
            currentFloor--;
        }
        System.out.println("Elevator at floor: " + currentFloor);
    }
    openDoors(); // Open doors when the destination floor is reached
}

// Opens the elevator doors and waits before closing them
private void openDoors() {
    doorOpen = true;
    System.out.println("Doors opening at floor: " + currentFloor);
    try { Thread.sleep(3000); } catch (InterruptedException e) {} // Simulates door open time
    closeDoors();
}

// Closes the elevator doors
private void closeDoors() {
    doorOpen = false;
    System.out.println("Doors closing at floor: " + currentFloor);
}

// Updates the elevator's movement direction based on pending requests
private void updateDirection() {
    if (!upRequests.isEmpty()) {
        direction = Direction.UP;
    } else if (!downRequests.isEmpty()) {
        direction = Direction.DOWN;
    } else {
```

```java
            direction = Direction.IDLE;
        }
    }
}

// Elevator system class to manage multiple elevators
class ElevatorSystem {
    private List<Elevator> elevators;
    public ElevatorSystem(int numElevators) {
        elevators = new ArrayList<>();
        for (int i = 0; i < numElevators; i++) {
            elevators.add(new Elevator()); // Initialize elevators
        }
    }

    // Handles an elevator request from a specific floor
    public void requestElevator(int floor) {
        Elevator bestElevator = findBestElevator(floor);
        if (bestElevator != null) {
            bestElevator.callElevator(floor);
        }
    }

    // Finds the best available elevator (currently assigns to the first one)
    private Elevator findBestElevator(int floor) {
        return elevators.get(0); // Simple logic: Assign to first elevator
    }

    // Starts the elevators' operation by running them in separate threads
    public void runElevators() {
        for (Elevator elevator : elevators) {
            new Thread(() -> elevator.moveElevator()).start();
        }
    }
```

```java
}

// Main application to test the elevator system
public class ElevatorSystemApp {
  public static void main(String[] args) {
    ElevatorSystem system = new ElevatorSystem(1); // Initialize with 1 elevator
    system.requestElevator(3); // Request elevator at floor 3
    system.requestElevator(7); // Request elevator at floor 7
    system.runElevators(); // Start elevator operations
  }
}
```

Explanation of Code

1. **Elevator Class**
   - Handles requests, movement, and door operations.
   - Uses PriorityQueue for up/down direction sorting.
   - Moves to requested floors and handles door opening/closing.
2. **ElevatorSystem Class**
   - Manages multiple elevators.
   - Uses a simple scheduler to assign requests.
3. **Multi-Threading**
   - Runs elevators as separate threads to simulate real-time movement.

Additional Features for Scaling

- Optimized Scheduling: Assign requests using Nearest-Available Elevator logic.
- Event-Driven System: Use Kafka/MQTT for request handling.
- Maintenance & Monitoring: Implement logging for failures and maintenance needs.