ELEVATOR SYSTEM

```
// Enums
enum Direction { UP, DOWN }
enum StateType { IDLE, MOVING_UP, MOVING_DOWN }

class Utils {
    static String getStateName(StateType state) {
        switch (state) {
            case IDLE: return "IDLE";
            case MOVING_UP: return "MOVING_UP";
            case MOVING_DOWN: return "MOVING_DOWN";
            default: return "UNKNOWN";
        }
    }
}

// Observer Interface
interface ElevatorObserver {
    void update(int floor, StateType state);
}

// State Pattern
abstract class State {
    protected Elevator elevator;
    public State(Elevator e) { this.elevator = e; }
    abstract void moveUp();
    abstract void moveDown();
    abstract void stop();
    abstract StateType getType();
}

// Forward declare concrete states
class IdleState extends State {
    public IdleState(Elevator e) { super(e); }
    public void moveUp() { elevator.setState(new MovingUpState(elevator)); }
    public void moveDown() { elevator.setState(new MovingDownState(elevator)); }
    public void stop() { /* Already idle */ }
    public StateType getType() { return StateType.IDLE; }
}

class MovingUpState extends State {
    public MovingUpState(Elevator e) { super(e); }
    public void moveUp() { /* Continue moving up */ }
    public void moveDown() { elevator.setState(new MovingDownState(elevator)); }
    public void stop() { elevator.setState(new IdleState(elevator)); }
    public StateType getType() { return StateType.MOVING_UP; }
}
```

```
class MovingDownState extends State {
    public MovingDownState(Elevator e) { super(e); }
    public void moveUp() { elevator.setState(new MovingUpState(elevator)); }
    public void moveDown() { /* Continue moving down */ }
    public void stop() { elevator.setState(new IdleState(elevator)); }
    public StateType getType() { return StateType.MOVING_DOWN; }
}

// Core Elevator Class
class Elevator {
    private int id;
    private int currentFloor;
    private State state;
    private Queue<Integer> floorQueue;
    private ElevatorManager manager;

    public Elevator(int id, ElevatorManager mgr) {
        this.id = id;
        this.currentFloor = 1;
        this.manager = mgr;
        this.state = new IdleState(this);
        this.floorQueue = new LinkedList<>();
        System.out.println("Elevator " + id + " created at floor " + currentFloor);
    }

    public void setState(State newState) {
        this.state = newState;
        System.out.println("Elevator " + id + " changed state to " +
Utils.getStateName(newState.getType()));
    }

    public void addToQueue(int floor) {
        floorQueue.add(floor);
        System.out.println("Elevator " + id + " received request for floor " + floor);
        processQueue();
    }

    public void processQueue() {
        if (floorQueue.isEmpty()) return;
        int targetFloor = floorQueue.peek();
        System.out.println("Elevator " + id + " processing request for floor " + targetFloor);
        if (targetFloor > currentFloor) {
            state.moveUp();
            simulateMovement(targetFloor);
        } else if (targetFloor < currentFloor) {
            state.moveDown();
            simulateMovement(targetFloor);
        }
    }
```

```
    public void simulateMovement(int targetFloor) {
        while (currentFloor != targetFloor) {
            if (targetFloor > currentFloor) currentFloor++;
            else currentFloor--;
            System.out.println("Elevator " + id + " is now at floor " + currentFloor);
        }
        floorQueue.poll();
        state.stop();
    }

    public int getCurrentFloor() { return currentFloor; }
    public int getId() { return id; }
    public State getState() { return state; }
}

// Strategy Pattern
interface ElevatorSelectionStrategy {
    Elevator selectElevator(int floor, Direction direction, List<Elevator> elevators);
}

class NearestElevatorStrategy implements ElevatorSelectionStrategy {
    public Elevator selectElevator(int floor, Direction direction, List<Elevator> elevators) {
        if (elevators.isEmpty()) return null;
        Elevator nearestElevator = elevators.get(0);
        int shortestDistance = Math.abs(floor - nearestElevator.getCurrentFloor());

        for (Elevator elevator : elevators) {
            int distance = Math.abs(floor - elevator.getCurrentFloor());
            StateType currentState = elevator.getState().getType();

            if (currentState == StateType.IDLE && distance < shortestDistance) {
                shortestDistance = distance;
                nearestElevator = elevator;
                continue;
            }
            if (direction == Direction.UP && currentState == StateType.MOVING_UP &&
                elevator.getCurrentFloor() < floor && distance < shortestDistance) {
                shortestDistance = distance;
                nearestElevator = elevator;
            } else if (direction == Direction.DOWN && currentState == StateType.MOVING_DOWN
&&
                    elevator.getCurrentFloor() > floor && distance < shortestDistance) {
                shortestDistance = distance;
                nearestElevator = elevator;
            }
        }
        return nearestElevator;
    }
}
```

```
// Manager Class
class ElevatorManager {
    private List<Elevator> elevators = new ArrayList<>();
    private List<OuterPanel> panels = new ArrayList<>();
    private List<ElevatorObserver> observers = new ArrayList<>();
    private ElevatorSelectionStrategy selectionStrategy;

    public ElevatorManager() {
        this.selectionStrategy = new NearestElevatorStrategy();
        System.out.println("Elevator Manager created");
    }

    public void setSelectionStrategy(ElevatorSelectionStrategy strategy) {
        this.selectionStrategy = strategy;
    }

    public void addToQueue(int floor, Direction direction) {
        System.out.println("Request received for floor " + floor);
        Elevator selectedElevator = selectionStrategy.selectElevator(floor, direction, elevators);
        if (selectedElevator != null) {
            selectedElevator.addToQueue(floor);
        }
    }

    public void notifyObservers(int floor, StateType state) {
        for (ElevatorObserver observer : observers) {
            observer.update(floor, state);
        }
    }

    public void addPanel(OuterPanel panel) {
        panels.add(panel);
        observers.add(panel);
    }

    public void addElevator(Elevator elevator) {
        elevators.add(elevator);
    }
}
```

```
// Outer Panel Class
class OuterPanel implements ElevatorObserver {
    private int floor;
    private ElevatorManager manager;
    private int currentDisplayFloor;
    private Direction currentDirection;

    public OuterPanel(int floorNum, ElevatorManager mgr) {
        this.floor = floorNum;
        this.manager = mgr;
        this.currentDisplayFloor = 1;
        System.out.println("Panel created at floor " + floorNum);
    }

    public void requestElevator(Direction direction) {
        System.out.println("Panel at floor " + floor + " requesting elevator");
        manager.addToQueue(floor, direction);
    }

    public void update(int floor, StateType state) {
        this.currentDisplayFloor = floor;
        System.out.println("Panel at floor " + this.floor + " updated: Elevator at floor " +
                floor + " (" + Utils.getStateName(state) + ")");
    }
}

// Main Simulation
public class ElevatorSimulation {
    public static void main(String[] args) {
        System.out.println("Starting Elevator System Simulation");
        System.out.println("==================================");
        ElevatorManager manager = new ElevatorManager();

        Elevator elevator1 = new Elevator(1, manager);
        Elevator elevator2 = new Elevator(2, manager);
        manager.addElevator(elevator1);
        manager.addElevator(elevator2);

        OuterPanel panel1 = new OuterPanel(1, manager);
        OuterPanel panel2 = new OuterPanel(2, manager);
        OuterPanel panel3 = new OuterPanel(3, manager);
        manager.addPanel(panel1);
        manager.addPanel(panel2);
        manager.addPanel(panel3);
        System.out.println("\nSimulating elevator requests");
        System.out.println("==================================");

        panel3.requestElevator(Direction.DOWN);
        panel1.requestElevator(Direction.UP);
        panel2.requestElevator(Direction.UP);
    }
}
```