# Iteration 1

Project Team: Naveen Prabakar, Ozair Nurani, Nicholas Wang, Brendan Kraft, Pranava Sai Maganti

## Inception Paragraph

The purpose of the project is to develop a system that manages hotels that records interactions between the guest and the hotel staff. Users will be able to book rooms, modify reservations and cancel them. During the booking process, the front-end should be able to check if rooms are available Users will be able to check in and out of their stay at the hotel. Users will receive a card that lets them access their room. They are also given the choice of being able to call room service. Room Service will clean the room and attend to the guests needs. After a user leaves they can leave reviews that the HR staff can look at and take feedback. Once users leave the cleaning staff will clean up the room and prepare it for availability. The overall purpose of the program is to have an organized layout of the transactions between clients and the hotel.

## Use Cases Names & BrainStorm

1. ~~Guest checks in → Name: Checking in~~
2. ~~Guest checks out → Name: Checking out~~
3. ~~Guest books a room → Name: Booking~~
4. Guest calls room service → Name: Calling Room service
5. Guest modifies booking → Name: Booking Modification
6. Guest cancels booking → Name: Booking cancellation
7. ~~Cleaning staff cleans the room → Name: Cleaning the room~~
8. Front desk checks for room availability → Name: Validation with Front Desk
9. ~~Guest uses card to authenticate room entry → Name: Authenticating Room Card~~
10. HR reviews guest ratings per hotel → Name: Reviewing Hotel Reviews

## Use Cases Cards

| Use Case: Checking in (Naveen) | |
|---|---|
| Primary Actor | User |
| Triggering Event | User enters the hotel and starts the check-in |

| Use Case: Checking in (Naveen) | |
|---|---|
| | process with front desk |
| Success Guarantee | User is given the key card to the room |
| Preconditions | 1. User needs to have a room reserved prior to check-in<br>2. User needs to have their ID to verify their Identity |
| Main Success Scenario | 1. User provides ID and details about reservation<br>2. Front Desk confirms reservation and Identity<br>3. Front Desk gives User the Key to their room |
| Alternative/Extensions | 1. User did not have prior reservation, they will not be given key to any room<br>2. User did not bring their ID verification, they will not be given key until provided<br>3. User checks in way before/after their check in time, their reservation will be invalid |

| Use Case: Checking out (Pranava) | |
|---|---|
| Primary Actor | User |
| Triggering Event | User initiates the check-out process at the end of their stay (via front desk, mobile app, website) |
| Success Guarantee | 1) The guest is successfully checked out of the system<br>2) Call the cleaning staff to clean the room and update the room status to available |
| Preconditions | 1) The guest has an active room in the |

| Use Case: Checking out (Pranava) | |
|---|---|
| | system<br>2) The guest stay period has been reached or passed the check out date |
| Main Success Scenario | 1) Guest initiates the check-out process<br>2) System issues a receipt of confirmation of the guests checkout |
| Alternative/Extensions | 1) System error when checkout, admin manually updates the room availability<br>2) Additional charge, staff reviews and deems necessary charges before processing the checkout |

| Use Case: Cleaning the Room (Nick) | |
|---|---|
| Primary Actor | Cleaner |
| Triggering Event | User checks out of the room and the room needs cleaning |
| Success Guarantee | Room is set to available if cleaning was successful with no damages |
| Preconditions | User must have checked out of the room |
| Main Success Scenario | 1. User checks out of room, and cleaning staff enters the room<br>2. Cleaning staff enters room and cleans it up for next reservation<br>3. Cleaning staff exits room and logs the room has been cleaned |
| Alternative/Extensions | 1. The user is still in the room/not checked out. Cleaning staff should go clean another room<br>2. There is damage in the room, cleaners need to report it in their logs |

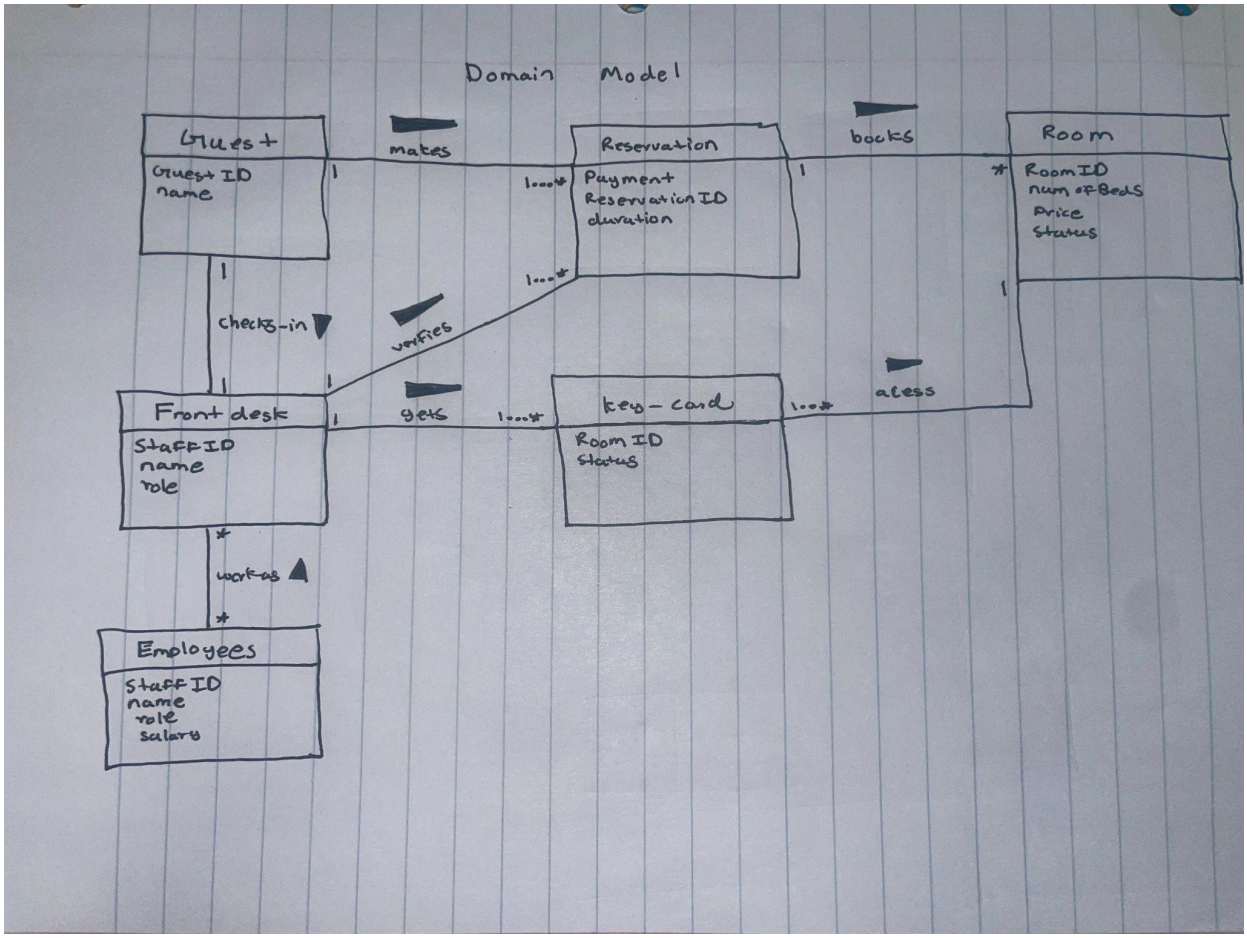
| Use Case: Booking a room (Ozair) | |
|---|---|
| Primary Actor | User |
| Triggering Event | User selects the option to book a room on the hotels website or a mobile app |
| Success Guarantee | 1) The booking is successfully confirmed and recorded in the system<br>2) A booking confirmation is sent to the user |
| Preconditions | 1) The user has access to the hotel booking system<br>2) Room for desired dates should be available |
| Main Success Scenario | 1) User enters preferences like check-in/checkout dates, number of guests, preferences and select room<br>2) User enters personal information and payment details<br>3) User receives a confirmation for booking |
| Alternative/Extensions | 1) User cancels before payment, process is aborted and no reservation is made<br>2) Payment fails, system prompts user to re-enter payment info or choose another method |

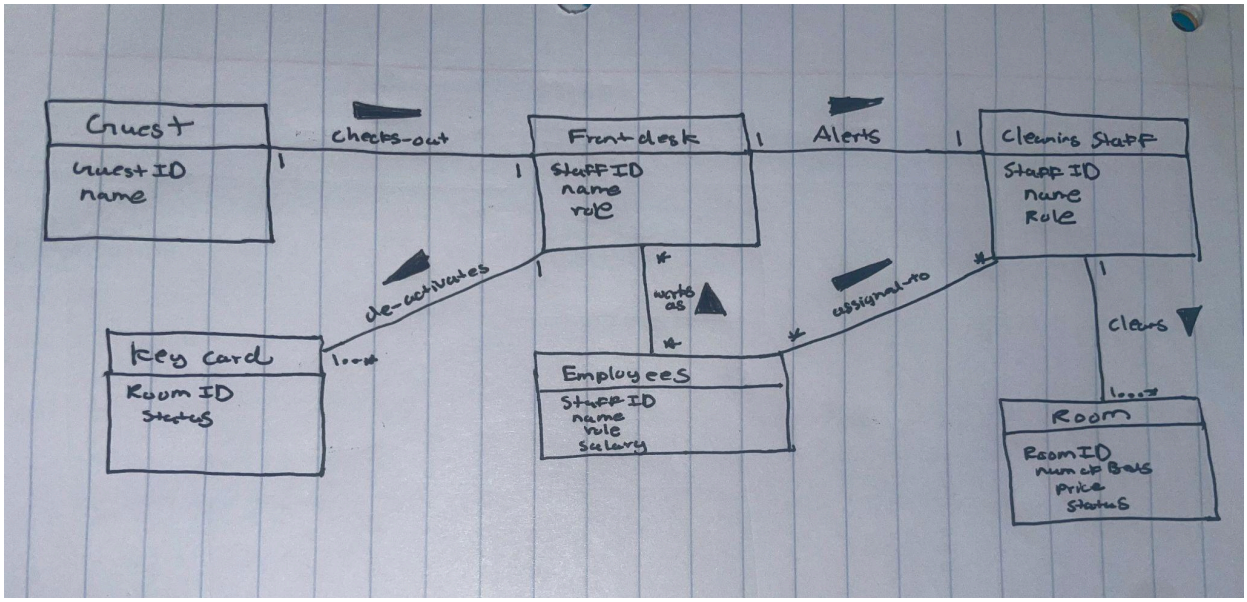
| Use Case: Authenticating with Room Card (Brendan) | |
|---|---|
| Primary Actor | User |
| Triggering Event | Guests swipe or tap their room card at the door's card reader to gain access to their assigned hotel room. |
| Success Guarantee | 1) The system validates the card and grants room access.<br>2) The door unlocks for the guest to enter. |
| Preconditions | 1) The guest has successfully checked in and has been assigned a valid room key card. |

| Use Case: Authenticating with Room Card (Brendan) | |
|---|---|
| | 2) The key card is active and associated with the correct room in the system. |
| Main Success Scenario | 1) The guest approaches the room door and presents the card to the reader.<br>2) The system verifies the card's data (guest identity, room number, validity period).<br>3) The system authenticates the card and unlocks the room door.<br>4) Guests enter the room successfully. |
| Alternative/Extensions | 1) Invalid Card: If the card is expired, deactivated, or does not match the room, access is denied, and the system flashes a red indicator.<br>2) System Error: If the reader malfunctions or cannot connect to the hotel system, maintenance is notified, and guests may request assistance at the front desk.<br>3) Lost Card: If a guest loses the card, the front desk issues a replacement and deactivates the old one. |

## Domain Model for the Iteration

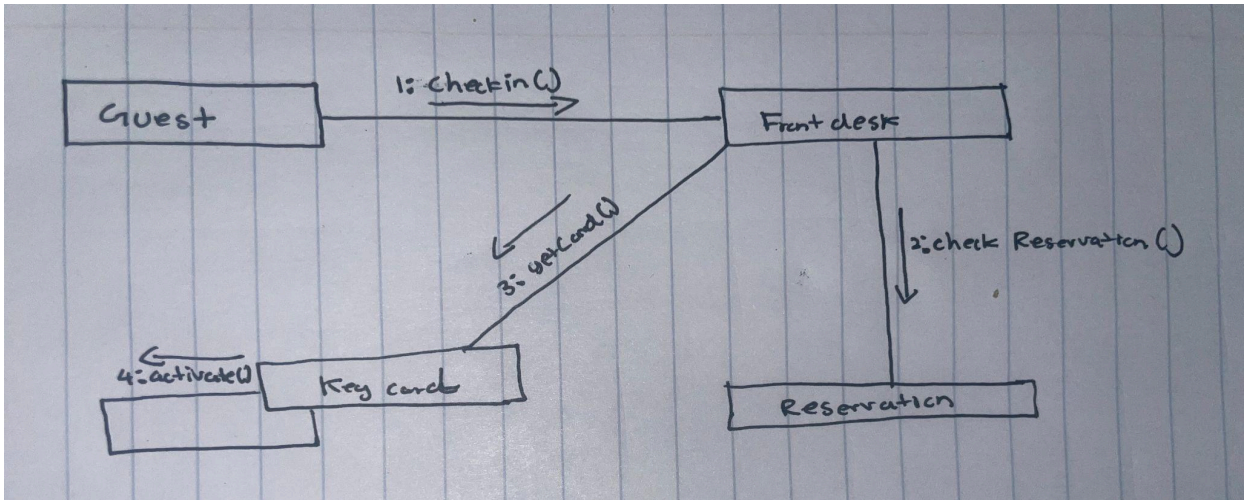
Two Domain Models were created to represent this iteration. This is because it involves two processes between User and Frontdesk: Check in and Check out. Having both of these in the domain model will make it break the domain model syntax, so two were created to support the domain model.



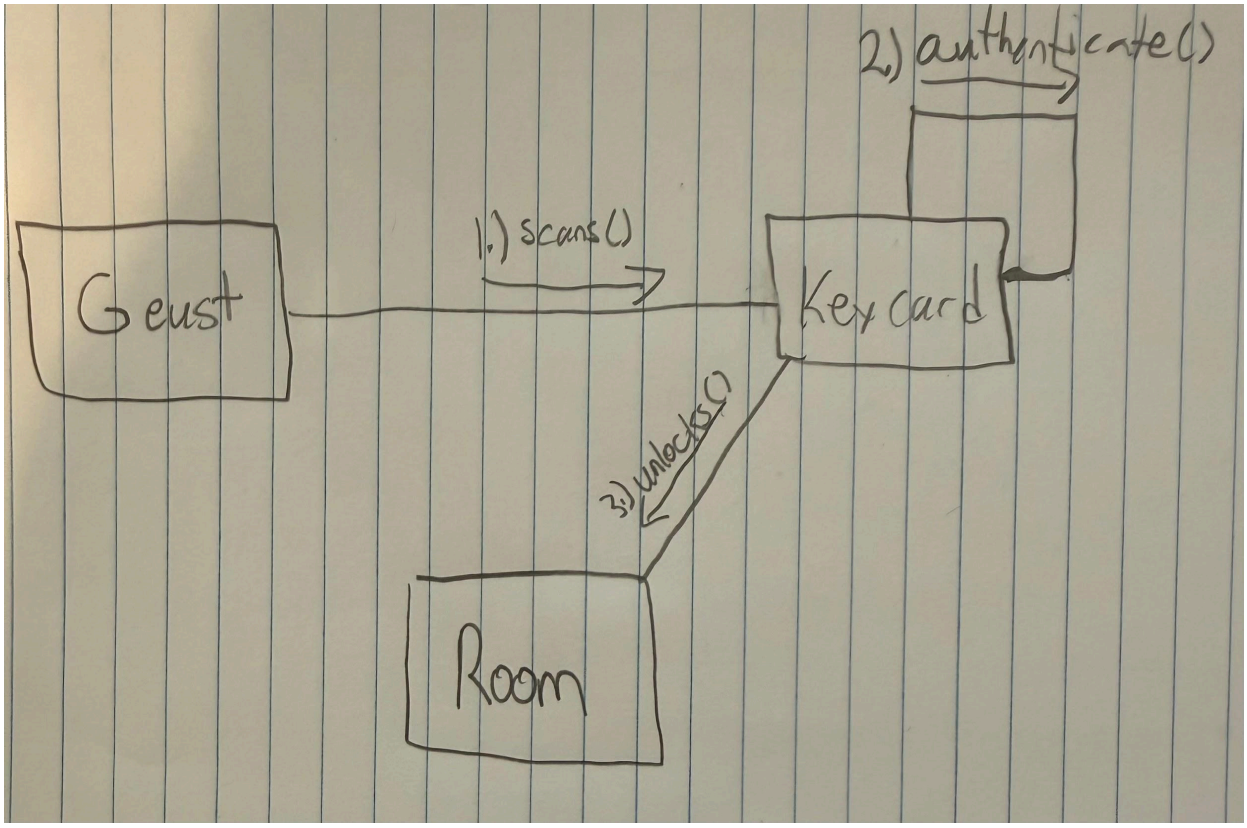




## Partial Communication Diagrams for Each Use Case

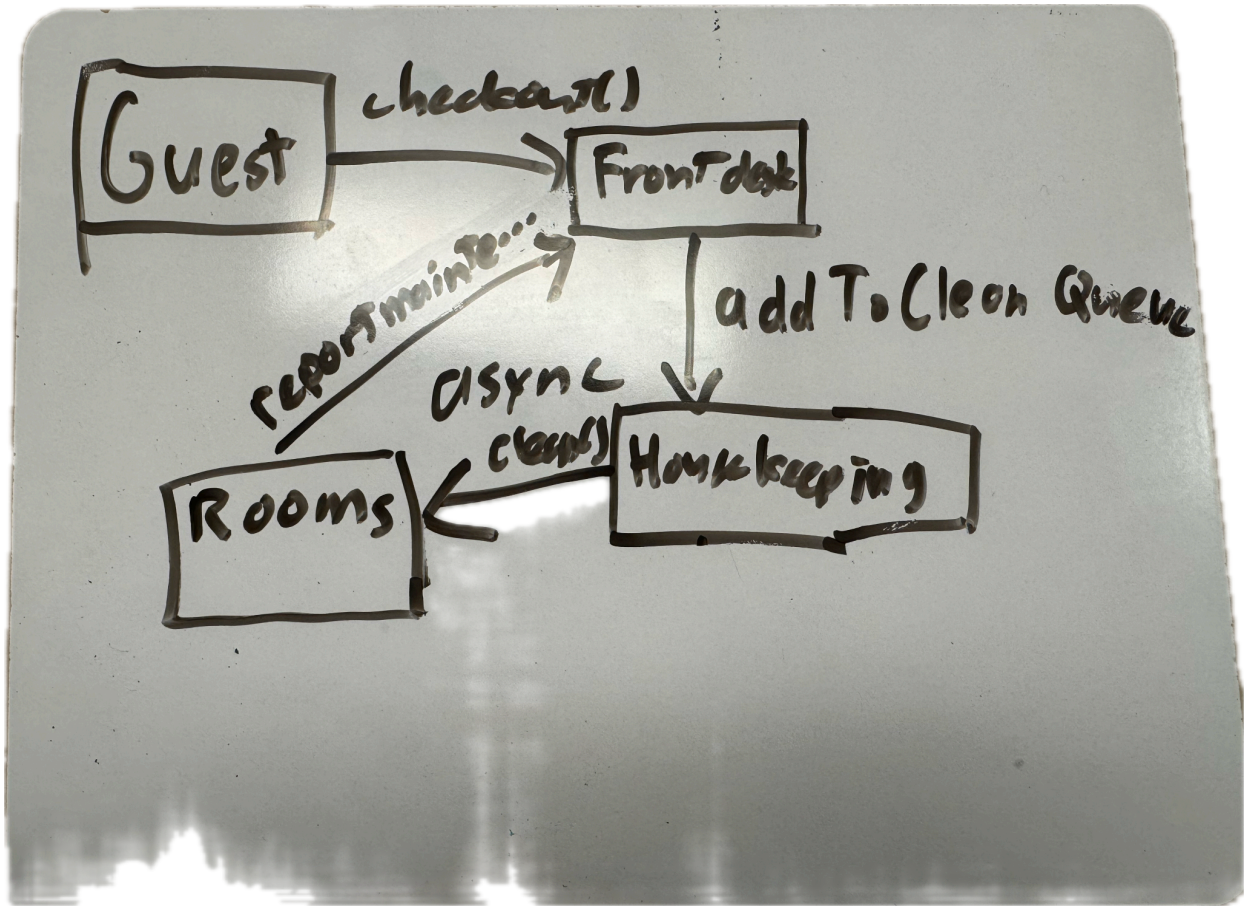
**Communication Diagram for the Checking in Use case (Naveen):**



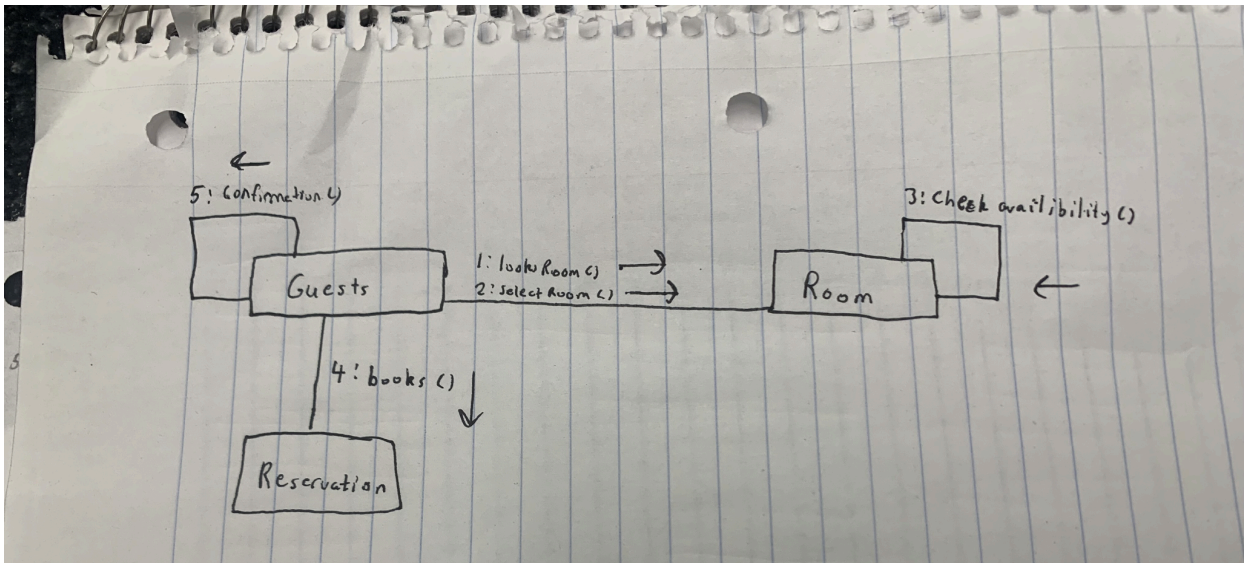

**Communication Diagram for Auth with Key Card Use case (Brendan):**



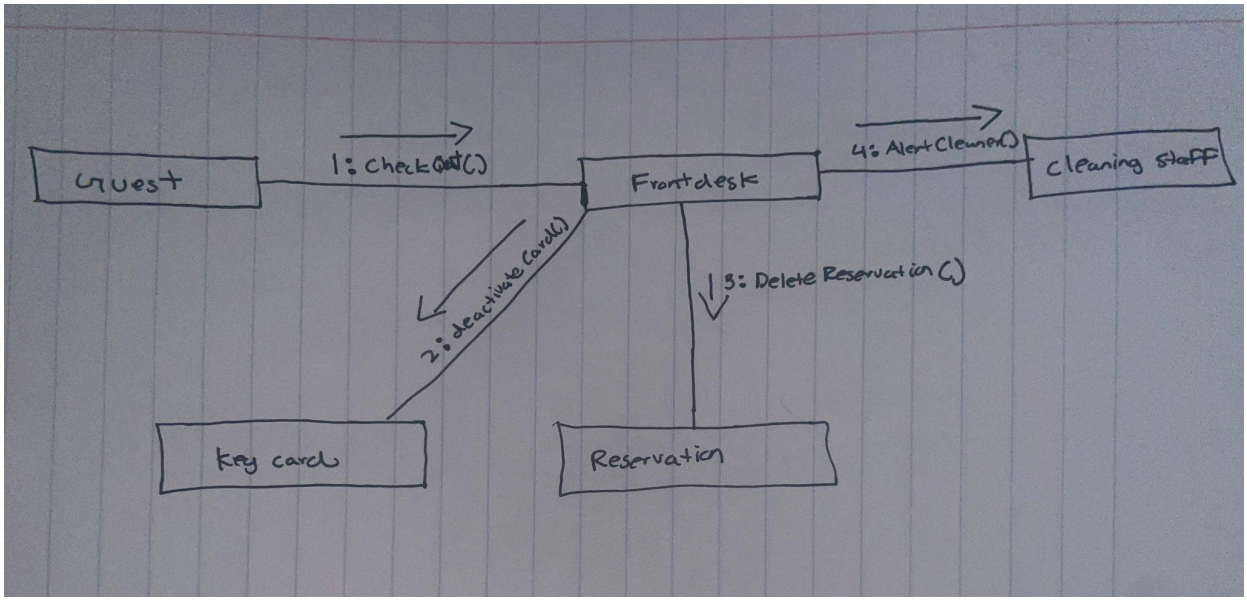

**Communication Diagram for Cleaning the room (Nick):**




**Communication Diagram for Booking in Use case (Ozair):**

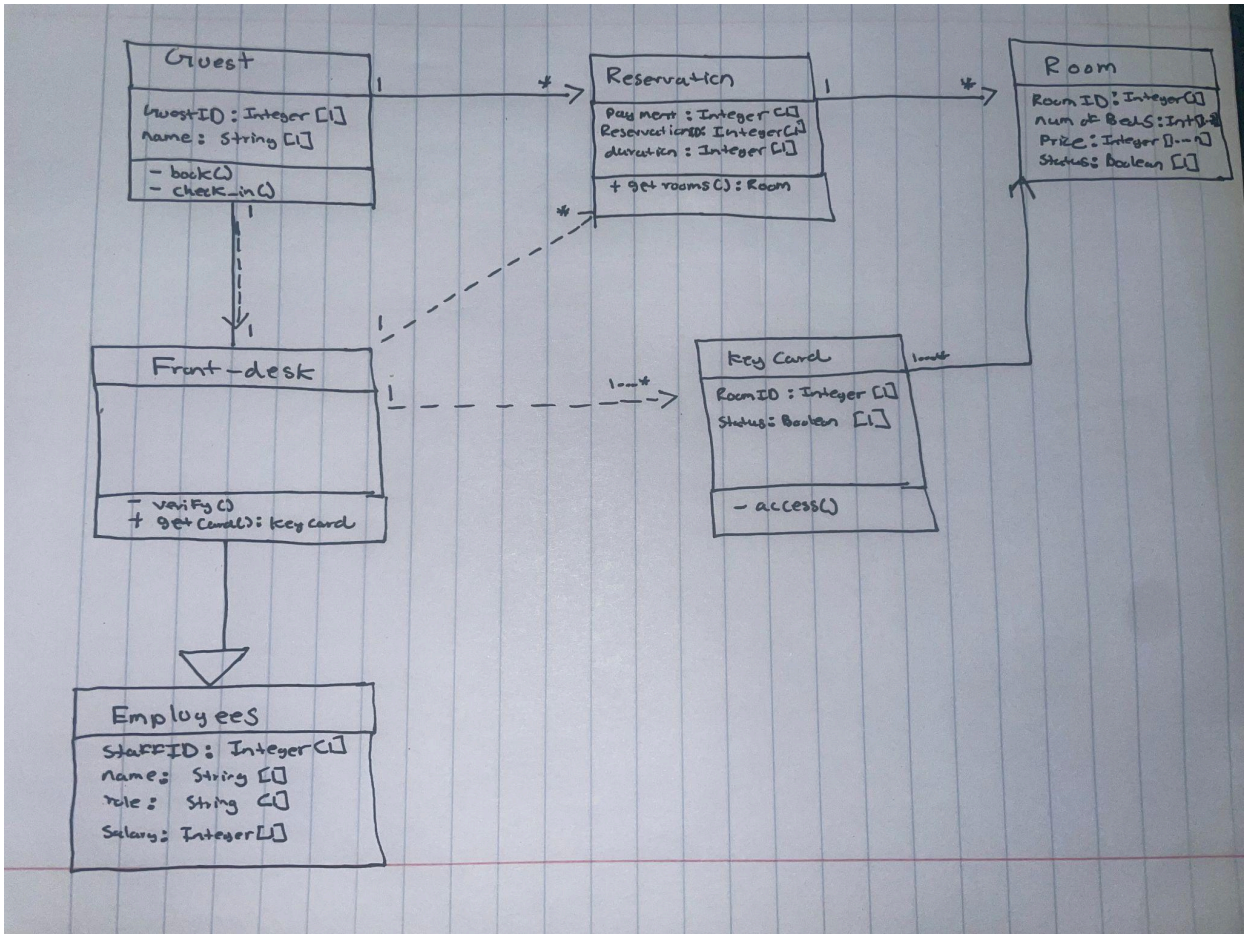


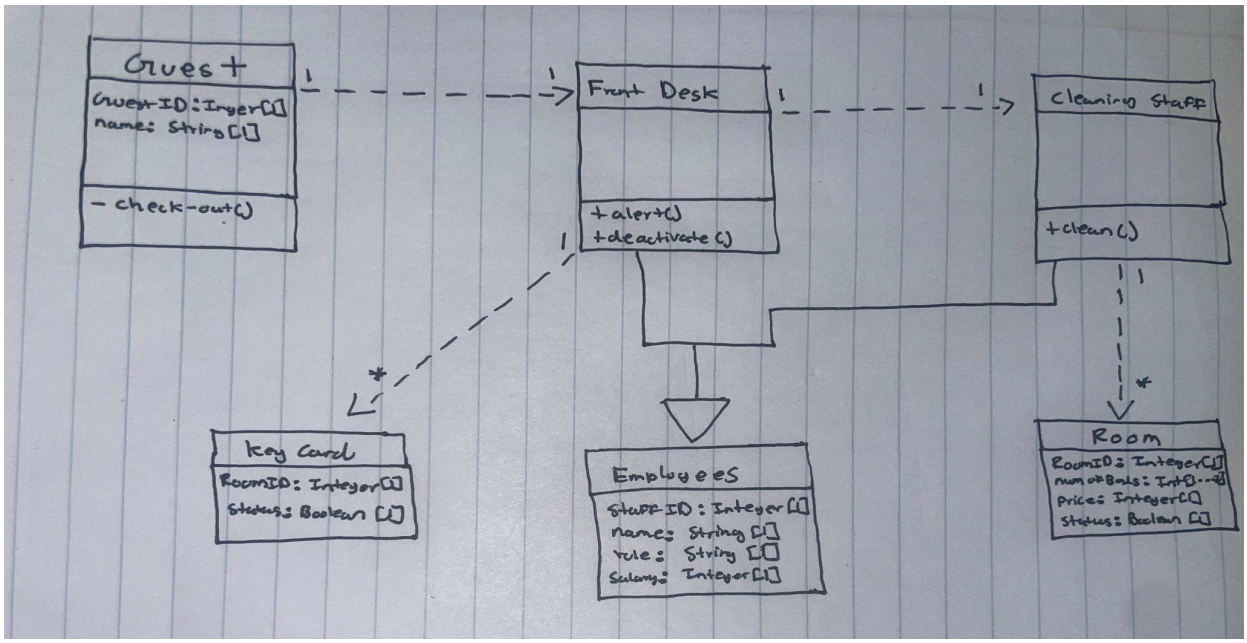

**Communication Diagram for Check out Use case (Pranava):**




## Partial Class Diagrams

Two Class Diagrams to express the overall workflow of this iteration. It covers the process of the user booking a room, checking in with frontdesk, getting a keycard and using it for the room. It also covers Users checking out with frontdesk, frontdesk alerts the cleaners, and cleaners clean the room.


Guest
GuestID : Integer [1]
name : String [1]
- book()
- check_in()
Reservation
Payment : Integer [1]
ReservationID: Integer [1]
duration : Integer [1]
+ get rooms() : Room
Room
RoomID : Integer [1]
num of Beds : Int [1]
Price : Integer [1..*]
Status : Boolean [1]
Front-desk
- verify()
+ getCard() : keycard
Key Card
RoomID : Integer [1]
Status : Boolean [1]
- access()
Employees
StaffID : Integer [1]
name : String [1]
role : String [1]
Salary : Integer [1]
1
*
1..*




## Database (Text files)

1. Reservation.txt → Keep track of the current reservations that have not ended yet
2. Guest.txt → Keep track of all users that have booked at the hotel
3. Keycard.txt → Have a storage of all issued key cards
4. Past_Reservation.txt → When guests check out, their reservation is archived (for future analytics branch use case)
5. Employee.txt → List of working employees in the company
6. Room.txt → List of rooms in the hotel

## Interface List

### Booking Interface:

```
Public interface BookingInterface {
    boolean createReservation(Reservation reservation);
    boolean modifyReservation(Reservation update);
    boolean cancelReservation(Reservation cancel);
    boolean checkAvailability(Room check);
}
```

### FrontDesk Interface:

```
Public interface FrontDesk {
    Reservation verifyCheckIn(Guest guest);
```

```
    boolean verifyIdentity(Guest guest);
    void provideKeyCard(int roomnumber, Guest owner);
    boolean verifyCheckOut();
    void revokeKeyCard();
    room alertCleaningStaff(int roomNumber);
}
```

### Cleaning Interface:

```
public interface CleaningStaff {
    void addToCleanQueue(room roomToClean);
}
```

### Room Interface:

```
public interface RoomInterface {
    boolean setStatus(String status);
    boolean assignToGuest(Guest guest);
}
```

#### CheckIn/CheckOut Interface:

```
public interface CheckingProcess {
    void checkin(Employee frontdesk);
    void checkout(Employee frontdesk);
}
```

#### Keycard Interface:

```
public interface KeyCardInterface {
    RoomAccessResult authenticateAndUnlock(Guest guest, room
targetRoom);
    int getRoomnumber();
    Boolean getStatus();
    Guest getOwner();
}
```

## Implementation

### Booking Use Case:

```
package Main.Booking;
```

```
import Main.Room.room;
import java.io.*;
import java.time.LocalDate;
import java.util.*;

public class Booking implements BookingInterface {

    private static final String ROOM_FILE = "Room.txt";
    private static final String RESERVATION_FILE = "Reservation.txt";
    private static final String GUEST_FILE = "Guest.txt";

    // ==================== MAIN MENU ====================
    public static void handleBooking() {
        Booking bookingSystem = new Booking();
        Scanner sc = new Scanner(System.in);

        while (true) {
            System.out.println("\n===== HOTEL BOOKING SYSTEM =====");
            System.out.println("1. Book a Room");
            System.out.println("2. Modify a Reservation");
            System.out.println("3. Cancel a Reservation");
            System.out.println("4. Exit");
            System.out.print("Select an option: ");

            String choice = sc.nextLine();
            switch (choice) {
                case "1" -> bookingSystem.bookRoomFlow();
                case "2" -> bookingSystem.modifyReservationFlow();
                case "3" -> bookingSystem.cancelReservationFlow();
                case "4" -> {
                    System.out.println("Goodbye!");
                    return;
                }
                default -> System.out.println("Invalid choice. Please try
again.");
            }
        }
    }

    // ==================== BOOK FLOW ====================
    private void bookRoomFlow() {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter Guest Name: ");
        String guestName = sc.nextLine();

        System.out.print("Enter Guest ID: ");
        int guestId = Integer.parseInt(sc.nextLine());
```

```
        System.out.print("Enter Check-in Date (YYYY-MM-DD): ");
        LocalDate checkIn = LocalDate.parse(sc.nextLine());

        System.out.print("Enter Check-out Date (YYYY-MM-DD): ");
        LocalDate checkOut = LocalDate.parse(sc.nextLine());

        List<room> availableRooms = getAvailableRooms(checkIn, checkOut);
        if (availableRooms.isEmpty()) {
            System.out.println("No rooms available for the selected dates.");
            return;
        }

        System.out.println("\nAvailable Rooms:");
        for (room r : availableRooms) {
            System.out.println(r.getRoomNumber() + " (" + r.getType() + ")");
        }

        System.out.print("\nSelect Room Number to book: ");
        String roomNumber = sc.nextLine();

        room selectedRoom = availableRooms.stream()
                .filter(r -> r.getRoomNumber() == Integer.parseInt(roomNumber))
                .findFirst().orElse(null);

        if (selectedRoom == null) {
            System.out.println("Invalid room selection.");
            return;
        }

        selectedRoom.setStartDate(checkIn);
        selectedRoom.setEndDate(checkOut);


        if (createReservation(guestName, guestId, selectedRoom)) {
            System.out.println("\nReservation created successfully!");
        } else {
            System.out.println("Could not create reservation.");
        }
    }

    // ==================== MODIFY FLOW ====================
    private void modifyReservationFlow() {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Reservation ID to modify: ");
        String reservationId = sc.nextLine();

        if (!modifyReservation(reservationId)) {
            System.out.println("Reservation not found or could not be
modified.");
```

```
        } else {
            System.out.println("Reservation updated successfully!");
        }
    }

    // ==================== CANCEL FLOW ====================
    private void cancelReservationFlow() {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Reservation ID to cancel: ");
        String reservationId = sc.nextLine();

        if (!cancelReservation(reservationId)) {
            System.out.println("Reservation not found or could not be
cancelled.");
        } else {
            System.out.println("Reservation cancelled successfully!");
        }
    }

    // ==================== INTERFACE IMPLEMENTATION ====================
    @Override
    public boolean createReservation(String guestName, int guestId, room room)
{
        try {
            // Generate a unique reservation ID
            String reservationId = UUID.randomUUID().toString();

            // Append reservation to Reservation.txt (without guestId)
            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(RESERVATION_FILE, true))) {
                bw.write(reservationId + "," + guestName + "," +
                        room.getRoomNumber() + "," + room.getStartDate() + ","
+ room.getEndDate());
                bw.newLine();
            }

            // Add guest to Guest.txt if they are new
            addGuestIfNew(guestName, guestId);

            return true;
        } catch (IOException e) {
            System.out.println("Error creating reservation: " +
e.getMessage());
            return false;
        }
    }


    @Override
```

```
    public boolean modifyReservation(String reservationId) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter new Check-in Date (YYYY-MM-DD): ");
        LocalDate newCheckIn = LocalDate.parse(sc.nextLine());
        System.out.print("Enter new Check-out Date (YYYY-MM-DD): ");
        LocalDate newCheckOut = LocalDate.parse(sc.nextLine());

        List<String> updatedReservations = new ArrayList<>();
        boolean found = false;

        try (BufferedReader br = new BufferedReader(new
FileReader(RESERVATION_FILE))) {
            String header = br.readLine();
            updatedReservations.add(header);

            String line;
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (parts[0].equals(reservationId)) {
                    line = parts[0] + "," + parts[1] + "," + parts[2] + "," +
                            newCheckIn + "," + newCheckOut;
                    found = true;
                }
                updatedReservations.add(line);
            }
        } catch (IOException e) {
            System.out.println("Error modifying reservation: " +
e.getMessage());
            return false;
        }

        if (found) {
            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(RESERVATION_FILE))) {
                for (String r : updatedReservations) {
                    bw.write(r);
                    bw.newLine();
                }
            } catch (IOException e) {
                System.out.println("Error saving modified reservation: " +
e.getMessage());
                return false;
            }
        }

        return found;
    }

    @Override
```

```
    public boolean cancelReservation(String reservationId) {
        List<String> updatedReservations = new ArrayList<>();
        boolean found = false;

        try (BufferedReader br = new BufferedReader(new
FileReader(RESERVATION_FILE))) {
            String header = br.readLine();
            updatedReservations.add(header);

            String line;
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (parts[0].equals(reservationId)) {
                    found = true;
                    continue; // skip
                }
                updatedReservations.add(line);
            }
        } catch (IOException e) {
            System.out.println("Error cancelling reservation: " +
e.getMessage());
            return false;
        }

        if (found) {
            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(RESERVATION_FILE))) {
                for (String r : updatedReservations) {
                    bw.write(r);
                    bw.newLine();
                }
            } catch (IOException e) {
                System.out.println("Error updating reservations: " +
e.getMessage());
                return false;
            }
        }

        return found;
    }

    @Override
    public boolean checkAvailability(room room) {
        LocalDate start = room.getStartDate();
        LocalDate end = room.getEndDate();
        return isRoomAvailable(room.getRoomNumber(), start, end);
    }

    // ==================== HELPER METHODS ====================
```

```
    private List<room> getAvailableRooms(LocalDate requestedStart, LocalDate
requestedEnd) {
        List<room> availableRooms = new ArrayList<>();
        try (BufferedReader br = new BufferedReader(new FileReader(ROOM_FILE)))
{
            br.readLine(); // skip header
            String line;
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                int roomNumber = Integer.parseInt(parts[0]);
                String type = parts[1];
                if (isRoomAvailable(roomNumber, requestedStart, requestedEnd))
{
                    availableRooms.add(new room(roomNumber, type, "Available",
null, null));
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading rooms: " + e.getMessage());
        }
        return availableRooms;
    }

    private boolean isRoomAvailable(int roomNumber, LocalDate requestedStart,
LocalDate requestedEnd) {
        try (BufferedReader br = new BufferedReader(new
FileReader(RESERVATION_FILE))) {
            br.readLine(); // skip header
            String line;
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (Integer.parseInt(parts[2]) == roomNumber) {
                    LocalDate existingStart = LocalDate.parse(parts[3]);
                    LocalDate existingEnd = LocalDate.parse(parts[4]);
                    if (!requestedEnd.isBefore(existingStart) &&
!requestedStart.isAfter(existingEnd)) {
                        return false;
                    }
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading reservations: " +
e.getMessage());
        }
        return true;
    }

    private void addGuestIfNew(String guestName, int guestId) {
```

```
        boolean exists = false;
        try (BufferedReader br = new BufferedReader(new
FileReader(GUEST_FILE))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] parts = line.split(",");
                if (parts.length >= 2 &&
parts[0].trim().equalsIgnoreCase(guestName)
                        && Integer.parseInt(parts[1].trim()) == guestId) {
                    exists = true;
                    break;
                }
            }
        } catch (IOException ignored) {
        }

        if (!exists) {
            try (BufferedWriter bw = new BufferedWriter(new
FileWriter(GUEST_FILE, true))) {
                bw.write(guestName + "," + guestId);
                bw.newLine();
            } catch (IOException e) {
                System.out.println("Error writing guest: " + e.getMessage());
            }
        }
    }
}

package Main.Booking;

import java.time.LocalDate;

public class Reservation {
    private String reservationId;
    private String guestName;
    private int  roomNumber;
    private LocalDate  startDate;
    private LocalDate endDate;

    public Reservation(String reservationId, String guestName, int roomNumber,
LocalDate  startDate, LocalDate  endDate) {
        this.reservationId = reservationId;
        this.guestName = guestName;
        this.roomNumber = roomNumber;
        this.startDate = startDate;
        this.endDate = endDate;
    }

    public String getReservationId() {
```

```
        return reservationId;
    }

    public String getGuestName() {
        return guestName;
    }

    public int getRoomNumber() {
        return roomNumber;
    }

    public LocalDate getStartDate() {
        return startDate;
    }

    public LocalDate getEndDate() {
        return endDate;
    }

    @Override
    public String toString() {
        return reservationId + "," + guestName + "," + roomNumber + "," +
startDate + "," + endDate;
    }
}
```

### Check In Use case & Check Out Use case (2 use cases)

```
package Main.Employee;

public class Employee {
    private int StaffID;
    private String name;
    private String role;

    public Employee(int StaffID, String name, String role){
        this.StaffID = StaffID;
        this.name = name;
        this.role = role;
    }

    private int getId(){
        return StaffID;
    }

    public String getName(){
        return name;
```

```
    }

    public String role() {
        return role;
    }
}

package Main.Employee;

import Main.Guest.*;
import Main.Booking.*;
import Main.Room.room;

import java.io.*;
import java.time.LocalDate;


import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.io.FileWriter;


public class frontdeskteam extends Employee implements FrontDesk{

    //access to the Guest Database
    private static String GUESTS = "Guest.txt";

    //access to reservation Database
    private static final String RESERVATION = "Reservation.txt";

    //access to keycard Database
    private static final String KEYCARD = "Keycard.txt";

    //access to past reservations
    private static final String PAST_RESERVATIONS = "Past_Reservation.txt";

    private static Housekeeping HousekeepingManager;

    private static final BlockingQueue<Runnable> requestQueue = new
LinkedBlockingQueue<Runnable>();
    private static String frontDeskPerson = null;

    public frontdeskteam(int id, String name) {
        super(id, name, "FrontDesk");
    }
```

```
    public void addHouseKeepingManager(Housekeeping manager){
        this.HousekeepingManager = manager;
    }

    @Override
    public Reservation verifyCheckIn(Guest guest) {

        File file = new File(RESERVATION);

        try{
            Scanner scnr = new Scanner(file);
            scnr.nextLine();

            while(scnr.hasNextLine()){
                String[] reserve = scnr.nextLine().split(",");

                String revid = reserve[0];
                String name = reserve[1];
                String roomnumber = reserve[2];
                String startDate = reserve[3];
                String endDate = reserve[4];

                if(name.equals(guest.getName())){
                    LocalDate actualCheckIn = LocalDate.now();
                    if(actualCheckIn.compareTo(LocalDate.parse(startDate)) <
0){
                        System.out.println("Sorry sir, you've arrvied to early
of your check in on " + startDate);
                        return null;
                    }
                    else if(actualCheckIn.compareTo((LocalDate.parse(endDate)))
> 0){
                        System.out.println("Sorry, you arrived past the end of
your booking");
                        return null;
                    }
                    return new Reservation(revid, name,
Integer.parseInt(roomnumber), LocalDate.parse(startDate),
LocalDate.parse(endDate));

                }
            }
        }
        catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }
        return null;
    }
```

```
@Override
public boolean verifyIdentity(Guest guest) {

    File file = new File(GUESTS);

    try {
        Scanner scnr = new Scanner(file);

        while(scnr.hasNextLine()){
            String[] profile = scnr.nextLine().split(",");
            String name = profile[0];
            int id = Integer.parseInt(profile[1]);

            if(guest.getName().equals(name) && guest.getid() == id){
                return true;
            }
        }

    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
    return false;
}


@Override
public void provideKeyCard(int roomnumber, Guest owner) {
    //Create they keycard
    KeyCard keycard = new KeyCard(roomnumber, true, owner);
    File log = new File(KEYCARD);

    //Record the card has been issued
    try{
        FileWriter f = new FileWriter(log, true);
        PrintWriter write = new PrintWriter(f);
        write.println(keycard.toString());
        write.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

@Override
public boolean verifyCheckOut(Guest guest) {
    File reservationFile = new File(RESERVATION);
    File pastFile = new File(PAST_RESERVATIONS);

    List<String> updatedReservations = new ArrayList<>();
    boolean found = false;
```

```
        int roomNumber = -1;

        try (Scanner reader = new Scanner(reservationFile)) {
            while (reader.hasNextLine()) {
                String line = reader.nextLine().trim();
                if (line.isEmpty() || line.startsWith("reservationId"))
continue; // skip header

                String[] parts = line.split(",");
                String resId = parts[0];
                String guestName = parts[1];
                int currentRoom = Integer.parseInt(parts[2]);
                String startDate = parts[3];
                String endDate = parts[4];

                if (guestName.equals(guest.getName())) {
                    found = true;
                    roomNumber = currentRoom;

                    // Record actual checkout date (today)
                    LocalDate actualCheckOut = LocalDate.now();

                    // Append to Past_Reservation.txt with actual checkout date
                    try (PrintWriter pastWriter = new PrintWriter(new
FileWriter(pastFile, true))) {
                        if (pastFile.length() == 0) {

pastWriter.println("reservationId,guestName,roomNumber,startDate,endDate,actual
CheckOutDate");
                        }
                        pastWriter.printf("%s,%s,%d,%s,%s,%s%n",
                                resId, guestName, currentRoom, startDate,
endDate, actualCheckOut);
                    }
                } else {
                    updatedReservations.add(line);
                }
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        if (!found) return false;

        // Rewrite the Reservation.txt with remaining active reservations
        try (PrintWriter writer = new PrintWriter(reservationFile)) {

writer.println("reservationId,guestName,roomNumber,startDate,endDate");
            for (String record : updatedReservations) {
```

```
                writer.println(record);
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        // Revoke keycard for this guest
        revokeKeyCard(roomNumber, guest);
        room roomToClean = alertCleaningStaff(roomNumber);


        HousekeepingManager.addToCleanQueue(roomToClean);
        System.out.println();
        System.out.println(roomToClean.getRoomNumber() + " " +
roomToClean.getType());
        System.out.println();

        return true;
    }

    @Override
    public void revokeKeyCard(int roomNumber, Guest guest) {
        File keycardFile = new File(KEYCARD);
        List<String> updatedRecords = new ArrayList<>();

        try (Scanner reader = new Scanner(keycardFile)) {
            while (reader.hasNextLine()) {
                String line = reader.nextLine().trim();
                if (line.isEmpty()) continue;

                String[] record = line.split("\\s+");
                int recordedRoom = Integer.parseInt(record[0]);
                String recordedGuest = record[2] + " " + record[3];

                // Remove only the matching keycard
                if (!(recordedRoom == roomNumber &&
guest.getName().equals(recordedGuest))) {
                    updatedRecords.add(line);
                }
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }

        // Rewrite keycard file without revoked card
        try (PrintWriter writer = new PrintWriter(keycardFile)) {
            for (String record : updatedRecords) {
                writer.println(record);
            }
```

```
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public room alertCleaningStaff(int roomNumber) {
        try {
            Scanner scnr = new Scanner(new File("Room.txt"));
            boolean roomMatch = false;
            String typeOfRoom = null;
            while (!roomMatch) {
                String roomData = scnr.nextLine();
                if (roomData.startsWith("roomNumber")) {
                    continue;
                }
                String[] roomInfo = roomData.split(",");
                int roomNum = Integer.parseInt(roomInfo[0]);
                if (roomNum == roomNumber) {
                    typeOfRoom = roomInfo[1];
                    roomMatch = true;
                }
            }
            room room = new room(roomNumber,typeOfRoom,null,null,null);
            System.out.println();
            System.out.println("Alerting the cleaning staff to clean room " +
room.getRoomNumber());
            System.out.println();
            return room;
        }
        catch (FileNotFoundException e) {
            throw new RuntimeException(e);
        }
    }

    //Ticket System
    public static void addToQueue(Runnable task) {
        requestQueue.add(task);
    }

    //Process Tickets
    public static void processQueue(String staffName) {
        frontDeskPerson = staffName;
        System.out.println();
        System.out.println(staffName + " logged in at the front desk.");
        System.out.println();

        while (!requestQueue.isEmpty() && frontDeskPerson != null) {
            try {
```

```
                Runnable task = requestQueue.take();
                System.out.println(staffName + " is now handling a guest
request");
                System.out.println();
                task.run();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                break;
            }
        }

        frontDeskPerson = null;
    }
}

package Main.Guest;

import Main.Booking.Reservation;
import Main.Employee.Employee;
import Main.Employee.FrontDesk;

import java.time.LocalTime;
import java.util.Scanner;

public class GuestSession implements CheckingProcess {

    @Override
    public void checkin(Employee frontDesk) {
        boolean process = true;
        Scanner scnr = new Scanner(System.in);

        while(process){
            String name = scnr.nextLine();
            System.out.println("Okay, and could you provide your id please?");
            int id = scnr.nextInt();

            Guest guest = new Guest(name, id);
            //Downcast to use FrontDesk
            boolean check = ((FrontDesk) frontDesk).verifyIdentity(guest);

            if(check){
                System.out.println("Thank you. Give me a moment to verify your
reservation");
                Reservation reserve = ((FrontDesk)
frontDesk).verifyCheckIn(guest);

                if(reserve == null){
                    break;
                }
```

```
                System.out.println();
                System.out.println("Thanks for waiting, you have a reservation
from " + reserve.getStartDate() + " to " + reserve.getEndDate());
                System.out.println();
                ((FrontDesk) frontDesk).provideKeyCard(reserve.getRoomNumber(),
guest);
                System.out.println();
                System.out.println("Here is your keycard for room " +
reserve.getRoomNumber() );
                break;
            }
            System.out.println("Sorry sir, you can't verify your identity");
            break;
        }
    }

    @Override
    public void checkout(Employee frontDesk) {
        Scanner scnr = new Scanner(System.in);
        LocalTime now = LocalTime.now();

        String greeting;
        if (now.isBefore(java.time.LocalTime.NOON)) {
            greeting = "Good morning";
        } else if (now.isBefore(java.time.LocalTime.of(17, 0))) {
            greeting = "Good afternoon";
        } else if (now.isBefore(java.time.LocalTime.of(21, 0))) {
            greeting = "Good evening";
        } else {
            greeting = "Good night";
        }

        System.out.println(greeting + "! Checking out today?");
        System.out.print("Can I get the name on the reservation? ");
        String name = scnr.nextLine();
        System.out.print("And the ID you used during check-in? ");
        int id = scnr.nextInt();
        scnr.nextLine(); // consume leftover newline

        Guest guest = new Guest(name, id);

        // Verify identity
        boolean verified = ((FrontDesk) frontDesk).verifyIdentity(guest);
        if (!verified) {
            System.out.println("Hmm, I’m not finding a match with that name and
ID. Could you double-check them for me?");
            return;
        }
```

```
        // Attempt checkout
        boolean checkedOut = ((FrontDesk) frontDesk).verifyCheckOut(guest);
        if (checkedOut) {
            System.out.println("Alright, " + guest.getName() + ", I found your
reservation.");
            System.out.println("Your checkout is all set. I'll take your
keycard—thank you!");
            System.out.println("We hope you had a pleasant stay. Safe
travels!");
        } else {
            System.out.println("It looks like you don't have any active
reservations right now.");
            System.out.println("If you've already checked out earlier, you're
all good.");
        }
    }
}

package Main.Guest;

import Main.Employee.Employee;
import Main.Employee.frontdeskteam;

public class CheckSystemController {
    public static void RunCheckin(Employee frontDesk){

        System.out.println("Waiting for FrontDesk Employee");

        frontdeskteam.addToQueue(() -> {
            System.out.println("===== HOTEL CHECK-IN SYSTEM =====");
            System.out.println();
            System.out.println("Hi, I am " + frontDesk.getName() + ", Can you
provide your name?");

            GuestSession check_in = new GuestSession();
            check_in.checkin(frontDesk);
        });
    }

    public static void RunCheckOut(Employee frontDesk){

        System.out.println("Waiting for FrontDesk Employee");

        frontdeskteam.addToQueue(() -> {
            System.out.println();
            System.out.println("===== HOTEL CHECK-OUT SYSTEM =====");
            System.out.println();

            GuestSession check_out = new GuestSession();
```

```
            check_out.checkout(frontDesk);
        });
    }
}
```

### Cleaning Use Case:

```
package Main.Employee;
import Main.Room.room;
import java.util.concurrent.LinkedBlockingQueue;

public class Housekeeping extends Employee implements CleaningStaff {
    private static final LinkedBlockingQueue<room> toCleanQueue = new
LinkedBlockingQueue<>();
    private static String currentCleaner = null;
    private static boolean active = true;

    public Housekeeping(int id, String name) {
        super(id, name, "CleaningStaff");
    }

    public void addToCleanQueue(room roomToClean) {
        try {
            toCleanQueue.put(roomToClean);
            System.out.println("A new cleaning request has been added to the
housekeeping queue.");
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }


    public static void processCleaning(String cleanerName) {
        if (!active) {
            System.out.println("Housekeeping is currently inactive.");
            return;
        }

        currentCleaner = cleanerName;
        System.out.println(cleanerName + " logged in to housekeeping.");

        while (!toCleanQueue.isEmpty()) {
            try {
                room roomToClean = toCleanQueue.take();
                System.out.println(cleanerName + " is now cleaning room " +
roomToClean.getRoomNumber());
                clean(roomToClean);
            } catch (InterruptedException e) {
```

```
                Thread.currentThread().interrupt();
                break;
            }
        }

        System.out.println("All pending cleaning tasks completed.");
        System.out.println(cleanerName + " logged out of housekeeping.");
        currentCleaner = null;
    }

    private static void clean(room roomToClean) {
        synchronized (roomToClean) {
            System.out.println("Cleaning Room: " +
roomToClean.getRoomNumber());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            System.out.println("Cleaning Room Done: " +
roomToClean.getRoomNumber());
        }
    }

    private static void fireAllWorkers() {
        active = false;
        System.out.println("Everyone is fired.");
    }
}

package Main.Guest;

import Main.Room.room;

public class Guest {
    private String name;
    private int id;

    public Guest(String name, int id){
        this.name = name;
        this.id = id;
    }

    public String getName(){
        return name;
    }

    public int getid(){
        return id;
```

```
    }

    public RoomAccessResult scanKeyCard(KeyCardInterface keyCard, room
targetRoom) {
        if (keyCard == null) {
            return RoomAccessResult.SYSTEM_ERROR;
        }
        return keyCard.authenticateAndUnlock(this, targetRoom);
    }

    public boolean matches(Guest other) {
        if (other == null) {
            return false;
        }
        boolean bothIdsKnown = id > 0 && other.id > 0;
        if (bothIdsKnown) {
            return id == other.id;
        }
        return name.equalsIgnoreCase(other.name);
    }
}
```

## KeyCard Use Case

```
package Main.Guest;

import Main.Room.room;

public class KeyCard implements KeyCardInterface {

    private int roomnumber;
    boolean status;
    Guest owner;
    public KeyCard(int roomnumber, boolean status, Guest owner){
        this.roomnumber = roomnumber;
        this.status = status;
        this.owner = owner;
    }

    public int getRoomnumber() {
        return roomnumber;
    }

    public boolean getStatus(){
        return status;
    }
```

```
    public Guest getOwner(){
        return owner;
    }

    public RoomAccessResult authenticateAndUnlock(Guest guest, room targetRoom)
{
        if (guest == null || targetRoom == null) {
            return RoomAccessResult.SYSTEM_ERROR;
        }
        if (!Boolean.TRUE.equals(status)) {
            return RoomAccessResult.CARD_INACTIVE;
        }
        if (!owner.matches(guest)) {
            return RoomAccessResult.INVALID_GUEST;
        }
        if (targetRoom.getRoomNumber() != roomnumber) {
            return RoomAccessResult.ROOM_MISMATCH;
        }
        boolean unlocked = targetRoom.unlock();
        return unlocked ? RoomAccessResult.ACCESS_GRANTED :
RoomAccessResult.ROOM_ALREADY_UNLOCKED;
    }

    @Override
    public String toString(){
        return roomnumber + " " + status + " " + owner.getName();
    }

    public String addToPastToString() { return roomnumber + " " +
owner.getName();}
}
```

```
package Main.Guest;

public enum RoomAccessResult {
    ACCESS_GRANTED("Access granted. Door unlocked."),
    CARD_INACTIVE("Access denied: your key card is inactive."),
    INVALID_GUEST("Access denied: the card is not assigned to you."),
    ROOM_MISMATCH("Access denied: the card does not unlock this room."),
    ROOM_ALREADY_UNLOCKED("Door already unlocked. Please enter."),
    SYSTEM_ERROR("System error: maintenance has been notified. Please contact
the front desk.");

    private final String message;

    RoomAccessResult(String message) {
        this.message = message;
    }
```

```
    public String getMessage() {
        return message;
    }

    public boolean isSuccess() {
        return this == ACCESS_GRANTED;
    }

    public boolean isSystemIssue() {
        return this == SYSTEM_ERROR;
    }
}
```

### Other things

```
package Main.Room;

import Main.Guest.Guest;

import java.time.LocalDate;

public class room implements RoomInterface {
    private int roomNumber;
    private String type;
    private String availability; // "Available" or "Reserved"
    private LocalDate  startDate; // YYYY-MM-DD or "null"
    private LocalDate endDate;   // YYYY-MM-DD or "null"
    private boolean locked;

    public room(int roomNumber, String type, String availability, LocalDate
startDate, LocalDate  endDate) {
        this.roomNumber = roomNumber;
        this.type = type;
        this.availability = availability;
        this.startDate = startDate;
        this.endDate = endDate;
        this.locked = true;
    }

    public static room fromString(String line) {
        String[] parts = line.split(",");
        int number = Integer.parseInt(parts[0]);
        LocalDate start = parts[3].equals("null") ? null :
LocalDate.parse(parts[3]);
        LocalDate end = parts[4].equals("null") ? null :
LocalDate.parse(parts[4]);
        return new room(number, parts[1], parts[2], start, end);
    }
```

```
    public int getRoomNumber() { return roomNumber; }
    public String getType() { return type; }
    public String getAvailability() { return availability; }
    public LocalDate getStartDate() { return startDate; }
    public LocalDate getEndDate() { return endDate; }
    public boolean isLocked() { return locked; }

    public void setAvailability(String availability) { this.availability =
availability; }
    public void setStartDate(LocalDate startDate) { this.startDate = startDate;
}
    public void setEndDate(LocalDate endDate) { this.endDate = endDate; }
    public void lock() {
        locked = true;
        availability = "Reserved";
    }
    public boolean unlock() {
        if (!locked) {
            return false;
        }
        locked = false;
        availability = "Occupied";
        return true;
    }

    @Override
    public boolean setStatus(String status) {
        this.availability = status;
        return true;
    }

    @Override
    public boolean assignToGuest(Guest guest) {
        this.availability = "Reserved";
        return true;
    }

    @Override
    public String toString() {
        return roomNumber + "," + type + "," + availability + "," + startDate +
"," + endDate;
    }
}
```

## The Main System Panel

```
package Main;

import java.io.FileNotFoundException;
import java.util.*;
import java.util.Scanner;
import java.io.File;

import Main.Booking.Booking;
import Main.Employee.*;
import Main.Guest.CheckSystemController;
import Main.Room.room;
import Main.Guest.Guest;
import Main.Guest.KeyCard;
import Main.Guest.RoomAccessResult;

public class Main {
    public static void main(String[] args) throws FileNotFoundException,
InterruptedException {
        Scanner sc = new Scanner(System.in);
        boolean running = true;

        File file = new File("Employee.txt");
        Scanner scnr = new Scanner(file);

        ArrayList<Employee> employees = new ArrayList<>();
        ArrayList<Housekeeping> housekeepings = new ArrayList<>();
        frontdeskteam FDManager = null;
        Housekeeping HKManager = null;

        // Get List of Employees Working
        while (scnr.hasNextLine()) {
            String line = scnr.nextLine();
            String[] employee = line.split(" ");

            if (employee.length < 4) continue; // prevent index error

            int id = Integer.parseInt(employee[0]);
            String name = employee[1] + employee[2];
            String position = employee[3];

            if (position.equals("FrontDesk")) {
                FDManager = new frontdeskteam(id, name);
                employees.add(FDManager);
            } else if (position.equals("Cleaner")) {
                HKManager = new Housekeeping(id, name);
                employees.add(HKManager);
            }
        }
```

```
if (FDManager != null && HKManager != null) {
    FDManager.addHouseKeepingManager(HKManager);
}

while (running) {
    printMenu();

    System.out.print("\nSelect an option (1-6) or 0 to exit: ");
    String choice = sc.nextLine().trim();

    if (choice.equals("1")) {
        Booking.handleBooking();
    } else if (choice.equals("2")) {
        System.out.println("Check In option selected.");
        Employee frontDesk = null;

        // Find available FrontDesk Employee
        for (Employee e : employees) {
            if (e.role().equals("FrontDesk")) {
                frontDesk = e;
            }
        }

        CheckSystemController.RunCheckin(frontDesk);

    } else if (choice.equals("6")) {
        handleRoomAccess(sc);
    } else if (choice.equals("3")) {
        System.out.println("Check Out option selected.");
        Employee frontDesk = null;

        // Find available FrontDesk Employee
        for (Employee e : employees) {
            if (e.role().equals("FrontDesk")) {
                frontDesk = e;
            }
        }

        // Call the checkout system
        CheckSystemController.RunCheckOut(frontDesk);
        //Thread.sleep(90000);

    } else if (choice.equals("0")) {
        break;

    } else if (choice.equals("9")) {
        int RoomNumber = 10;
        int i = 0;
        while (i < 3) {
```

```
                    housekeepings.add(new Housekeeping(i, "alpha"));
                    System.out.println("added new housekeeping " + i);
                    i++;
                }
                while (true) {
                    housekeepings.get(0).addToCleanQueue(new room(RoomNumber,
null, null, null, null));
                    RoomNumber += 1;
                }

            }else if (choice.equals("4")) {
                System.out.print("Enter your name to log in: ");
                String name = sc.nextLine().trim();
                frontdeskteam.processQueue(name);
            }
            else if (choice.equals("5")) {
                System.out.print("Enter your name to log in as cleaner: ");
                String cleanerName = sc.nextLine().trim();
                Housekeeping.processCleaning(cleanerName);
            }
            else {
                System.out.println("That's not an option try again");
                continue;
            }
        }
    }

    private static void printMenu() {
        System.out.println("╔═══════════════════════════════════════════╗");
        System.out.println("║             HOTEL ERP SYSTEM              ║");
        System.out.println("╠═══════════════════════════════════════════╣");
        System.out.println("║ 1. Book                                   ║");
        System.out.println("║ 2. Check In                               ║");
        System.out.println("║ 3. Check Out                              ║");
        System.out.println("║ 4. Front Desk                             ║");
        System.out.println("║ 5. Cleaning Staff                         ║");
        System.out.println("║ 6. Access Room (Key Card)                 ║");
        System.out.println("║ 0. Exit                                   ║");
        System.out.println("╚═══════════════════════════════════════════╝");
    }

    private static void handleRoomAccess(Scanner sc) {
        new RoomAccessFlow(sc).execute();
    }

    private static final class RoomAccessFlow {
        private static final String DATA_ROOT =
"Backend/Hotel/untitled/src/Main/";
        private static final String FALLBACK_ROOT = "src/Main/";
```

```
        private final Scanner input;

        RoomAccessFlow(Scanner input) {
            this.input = input;
        }

        void execute() {
            System.out.println("===== ROOM ACCESS AUTHENTICATION =====");
            List<KeyCard> issuedCards = loadIssuedKeyCards();
            if (issuedCards.isEmpty()) {
                System.out.println("No key cards are currently issued. Please
visit the front desk.");
                return;
            }

            System.out.println("Key cards on file:");
            for (KeyCard card : issuedCards) {
                System.out.println("  Room " + card.getRoomnumber() + " -> " +
card.getOwner().getName());
            }

            System.out.print("Room number: ");
            int roomNumber = parseNumber(input.nextLine().trim());

            System.out.print("Name on key card: ");
            String guestName = input.nextLine().trim();

            if (roomNumber <= 0 || guestName.isEmpty()) {
                System.out.println("Access denied: incomplete information
provided.");
                System.out.println("Indicator flashes red. Please contact the
front desk.");
                return;
            }

            Optional<KeyCard> keyCard = findMatchingCard(issuedCards,
roomNumber, guestName);
            if (keyCard.isEmpty()) {

System.out.println(RoomAccessResult.INVALID_GUEST.getMessage());
                System.out.println("Indicator flashes red. Please contact the
front desk.");
                return;
            }

            Optional<room> targetRoom = loadRoom(roomNumber);
            if (targetRoom.isEmpty()) {
                System.out.println(RoomAccessResult.SYSTEM_ERROR.getMessage());
```

```
                System.out.println("Maintenance notified. Please visit the
front desk.");
                return;
            }

            Guest guest = new Guest(guestName, -1);
            RoomAccessResult result = guest.scanKeyCard(keyCard.get(),
targetRoom.get());
            System.out.println(result.getMessage());

            switch (result) {
                case ACCESS_GRANTED:
                    System.out.println("Door for room " + roomNumber + " is now
unlocked.");
                    break;
                case ROOM_ALREADY_UNLOCKED:
                    System.out.println("Door was already unlocked. Please
enter.");
                    break;
                case SYSTEM_ERROR:
                    System.out.println("Maintenance notified. Please visit the
front desk.");
                    break;
                default:
                    System.out.println("Indicator flashes red. Please contact
the front desk.");
                    break;
            }
        }

        private List<KeyCard> loadIssuedKeyCards() {
            File source = resolveDataFile("Keycard.txt");
            List<KeyCard> cards = new ArrayList<>();
            if (!source.exists()) {
                return cards;
            }

            try (Scanner scanner = new Scanner(source)) {
                while (scanner.hasNextLine()) {
                    String line = scanner.nextLine().trim();
                    if (line.isEmpty()) {
                        continue;
                    }
                    String[] tokens = line.split("\\s+");
                    if (tokens.length < 3) {
                        continue;
                    }

                    int recordedRoom = parseNumber(tokens[0]);
```

```
                    if (recordedRoom <= 0) {
                        continue;
                    }

                    boolean isActive = Boolean.parseBoolean(tokens[1]);
                    String ownerName = rebuildName(tokens);
                    if (ownerName.isEmpty()) {
                        continue;
                    }

                    cards.add(new KeyCard(recordedRoom, isActive, new
Guest(ownerName, -1)));
                }
            } catch (FileNotFoundException e) {
                cards.clear();
            }

            return cards;
        }

        private Optional<KeyCard> findMatchingCard(List<KeyCard> cards, int
roomNumber, String guestName) {
            KeyCard match = null;
            for (KeyCard card : cards) {
                if (card.getRoomnumber() == roomNumber &&
                        card.getOwner().getName().equalsIgnoreCase(guestName))
{
                    match = card;
                }
            }
            return Optional.ofNullable(match);
        }

        private Optional<room> loadRoom(int roomNumber) {
            if (roomNumber <= 0) {
                return Optional.empty();
            }

            File source = resolveDataFile("Room.txt");
            if (!source.exists()) {
                return Optional.empty();
            }

            try (Scanner scanner = new Scanner(source)) {
                if (scanner.hasNextLine()) {
                    scanner.nextLine();
                }
                while (scanner.hasNextLine()) {
                    String line = scanner.nextLine().trim();
```

```
                    if (line.isEmpty()) {
                        continue;
                    }
                    String[] parts = line.split(",");
                    if (parts.length < 2) {
                        continue;
                    }

                    int recordedRoom = parseNumber(parts[0].trim());
                    if (recordedRoom != roomNumber) {
                        continue;
                    }

                    String type = parts[1].trim();
                    room entry = new room(recordedRoom, type, "Reserved", null,
null);
                    entry.lock();
                    return Optional.of(entry);
                }
            } catch (FileNotFoundException e) {
                return Optional.empty();
            }

            return Optional.empty();
        }

        private File resolveDataFile(String fileName) {
            String[] prefixes = {
                    "",
                    DATA_ROOT,
                    FALLBACK_ROOT,
                    "Backend/Hotel/untitled/",
                    "Backend/Hotel/"
            };

            for (String prefix : prefixes) {
                File candidate = prefix.isEmpty() ? new File(fileName) : new
File(prefix + fileName);
                if (candidate.exists()) {
                    return candidate;
                }
            }

            return new File(DATA_ROOT + fileName);
        }

        private int parseNumber(String value) {
            if (value == null || value.isEmpty()) {
                return -1;
```

```
            }
            try {
                return Integer.parseInt(value);
            } catch (NumberFormatException ex) {
                return -1;
            }
        }

        private String rebuildName(String[] tokens) {
            if (tokens.length <= 2) {
                return "";
            }
            StringBuilder builder = new StringBuilder();
            for (int i = 2; i < tokens.length; i++) {
                if (i > 2) {
                    builder.append(' ');
                }
                builder.append(tokens[i]);
            }
            return builder.toString();
        }
    }
}
```