Skypay Technical Test Presentation

Hotel Reservation System

Technical Challenge Overview

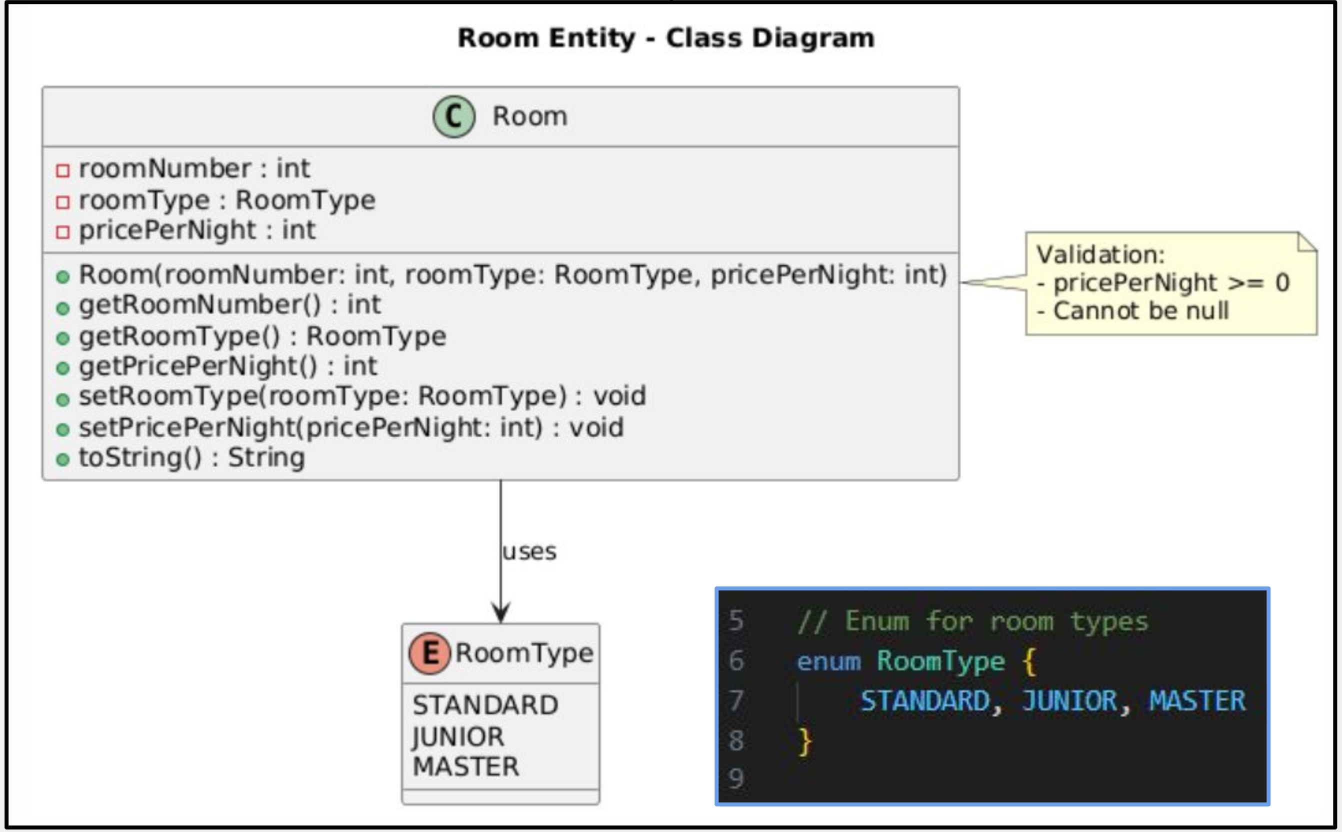
Content:

- ❖ Implement a Hotel Reservation System in Java
- ❖ Manage 3 main entities: Room, User, Booking

■ **Key Requirements:**

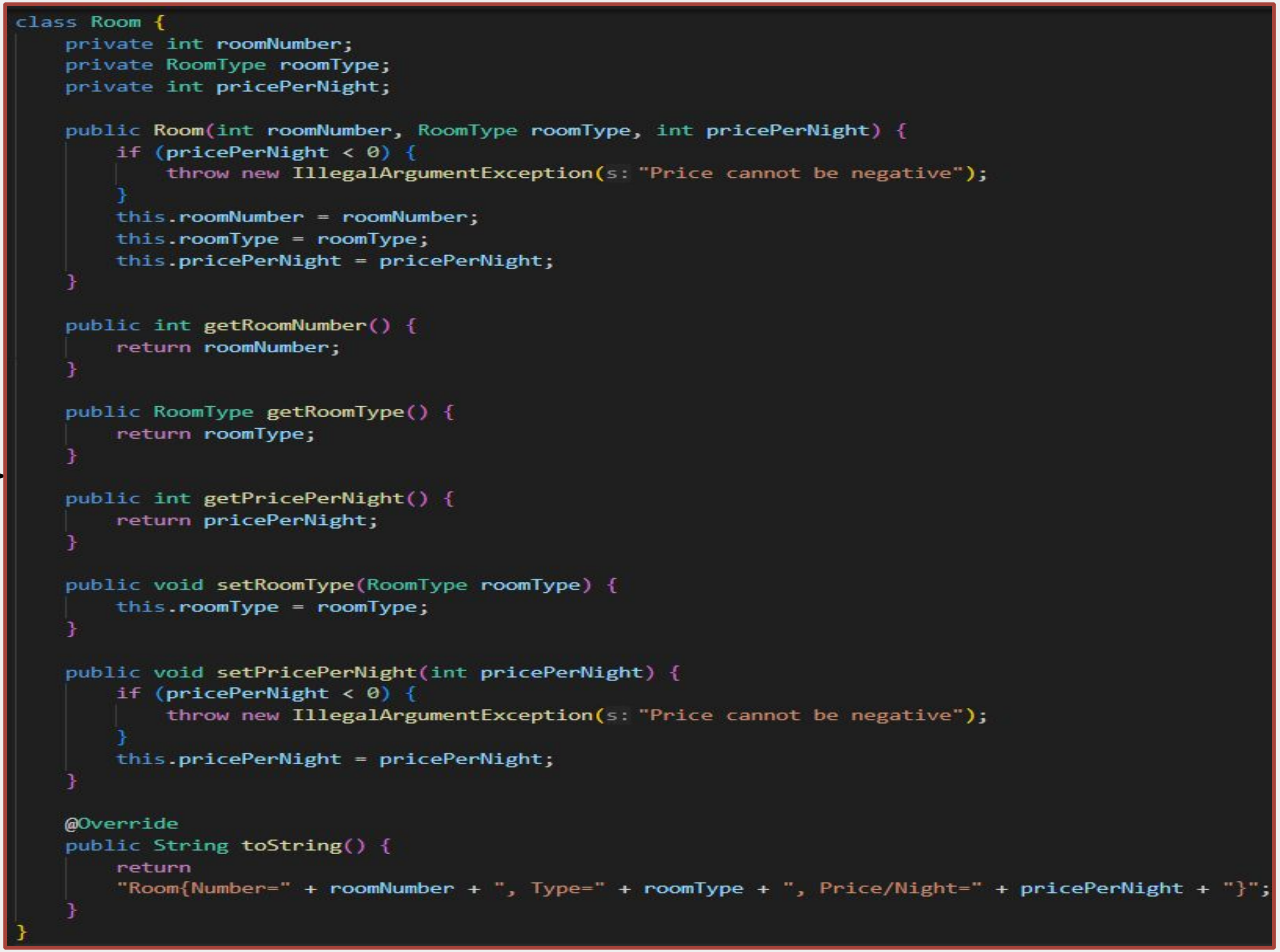
- ❖ Users can book rooms if they have sufficient balance
- ❖ Rooms have different types and prices
- ❖ Track booking history with snapshots
- ❖ Update rooms without affecting past bookings

Room Entity - Implementation


Room Entity - Class Diagram
Room
- roomNumber : int
- roomType : RoomType
- pricePerNight : int
+ Room(roomNumber: int, roomType: RoomType, pricePerNight: int)
+ getRoomNumber() : int
+ getRoomType() : RoomType
+ getPricePerNight() : int
+ setRoomType(roomType: RoomType) : void
+ setPricePerNight(pricePerNight: int) : void
+ toString() : String
Validation:
- pricePerNight >= 0
- Cannot be null
uses
RoomType
STANDARD
JUNIOR
MASTER

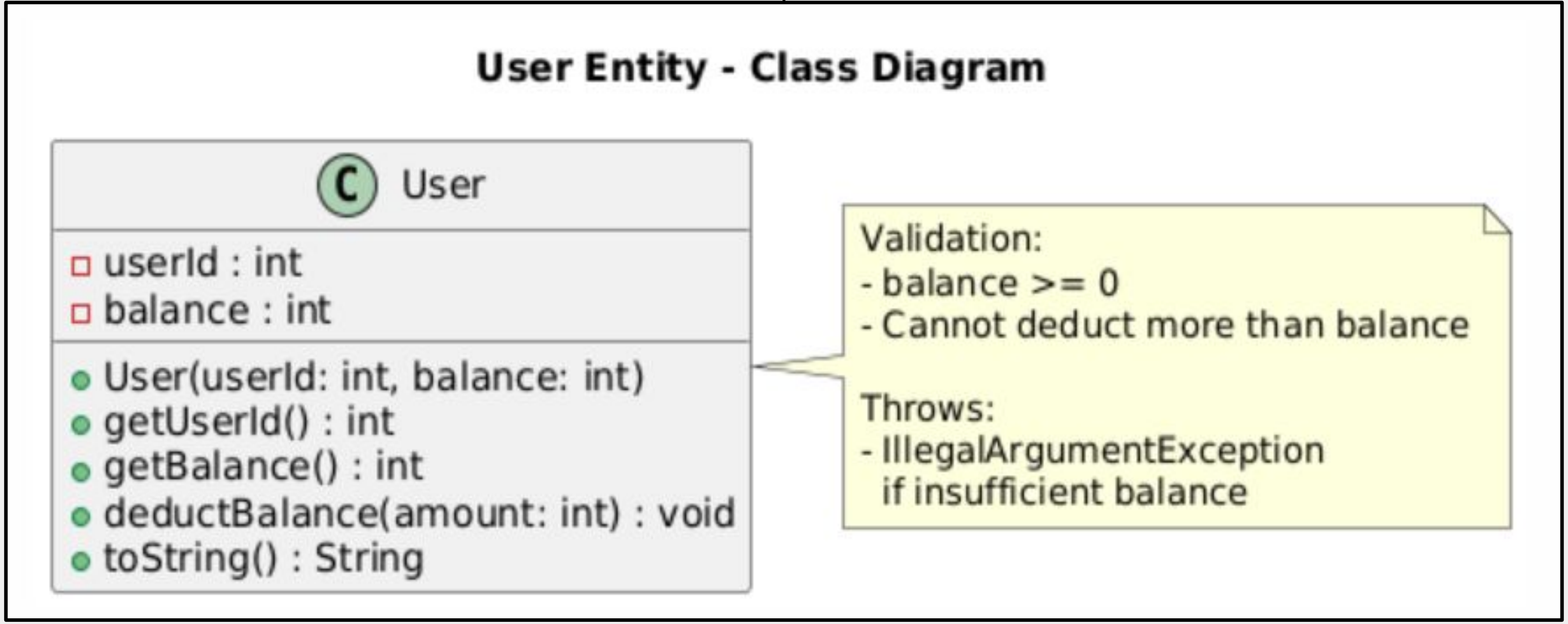
```
// Enum for room types
enum RoomType {
    STANDARD, JUNIOR, MASTER
}
```

Room Entity - Implementation

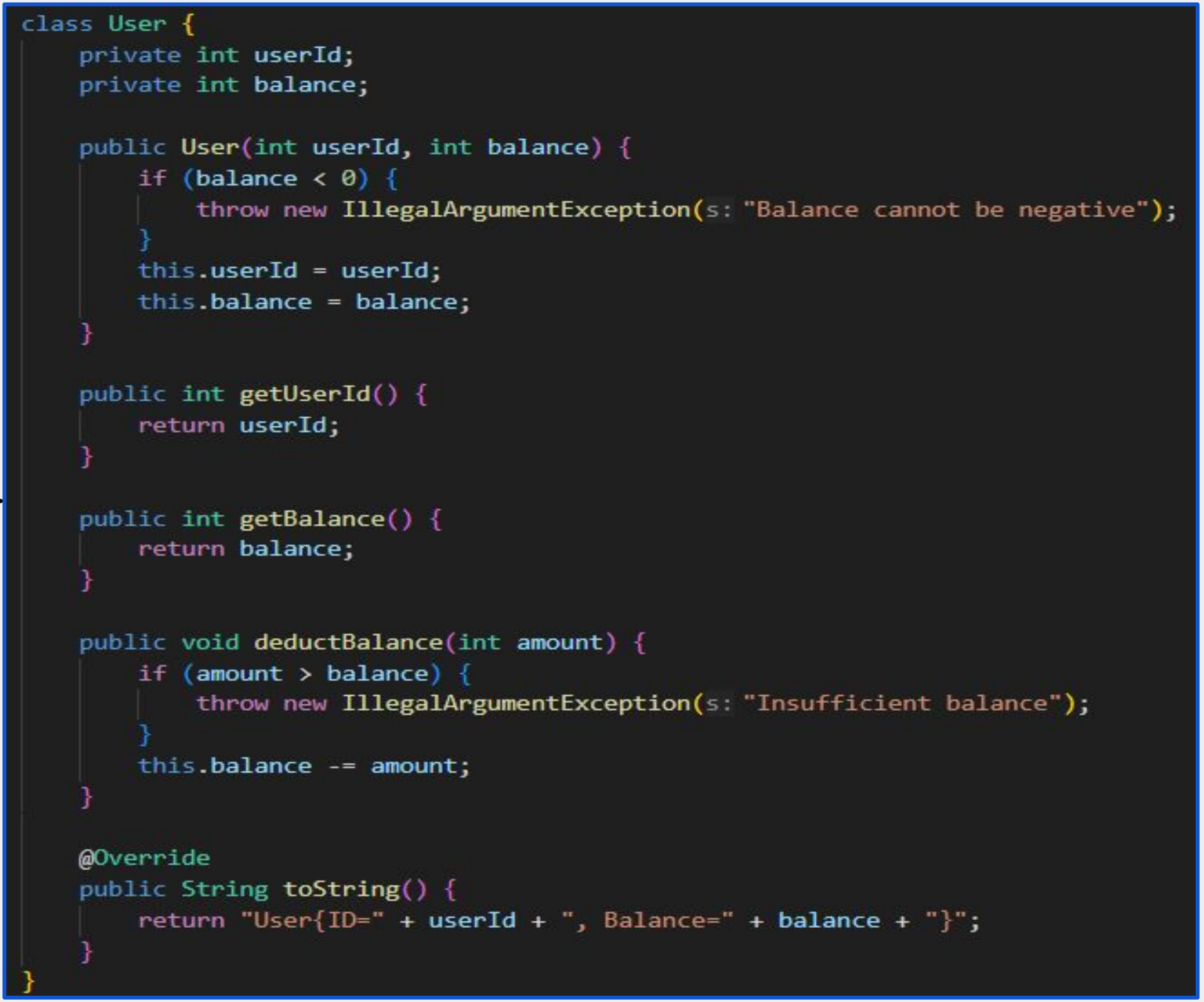


```java
class Room {
    private int roomNumber;
    private RoomType roomType;
    private int pricePerNight;

    public Room(int roomNumber, RoomType roomType, int pricePerNight) {
        if (pricePerNight < 0) {
            throw new IllegalArgumentException(s: "Price cannot be negative");
        }
        this.roomNumber = roomNumber;
        this.roomType = roomType;
        this.pricePerNight = pricePerNight;
    }

    public int getRoomNumber() {
        return roomNumber;
    }

    public RoomType getRoomType() {
        return roomType;
    }

    public int getPricePerNight() {
        return pricePerNight;
    }

    public void setRoomType(RoomType roomType) {
        this.roomType = roomType;
    }

    public void setPricePerNight(int pricePerNight) {
        if (pricePerNight < 0) {
            throw new IllegalArgumentException(s: "Price cannot be negative");
        }
        this.pricePerNight = pricePerNight;
    }

    @Override
    public String toString() {
        return
        "Room{Number=" + roomNumber + ", Type=" + roomType + ", Price/Night=" + pricePerNight + "}";
    }
}
```

User Entity - Implementation

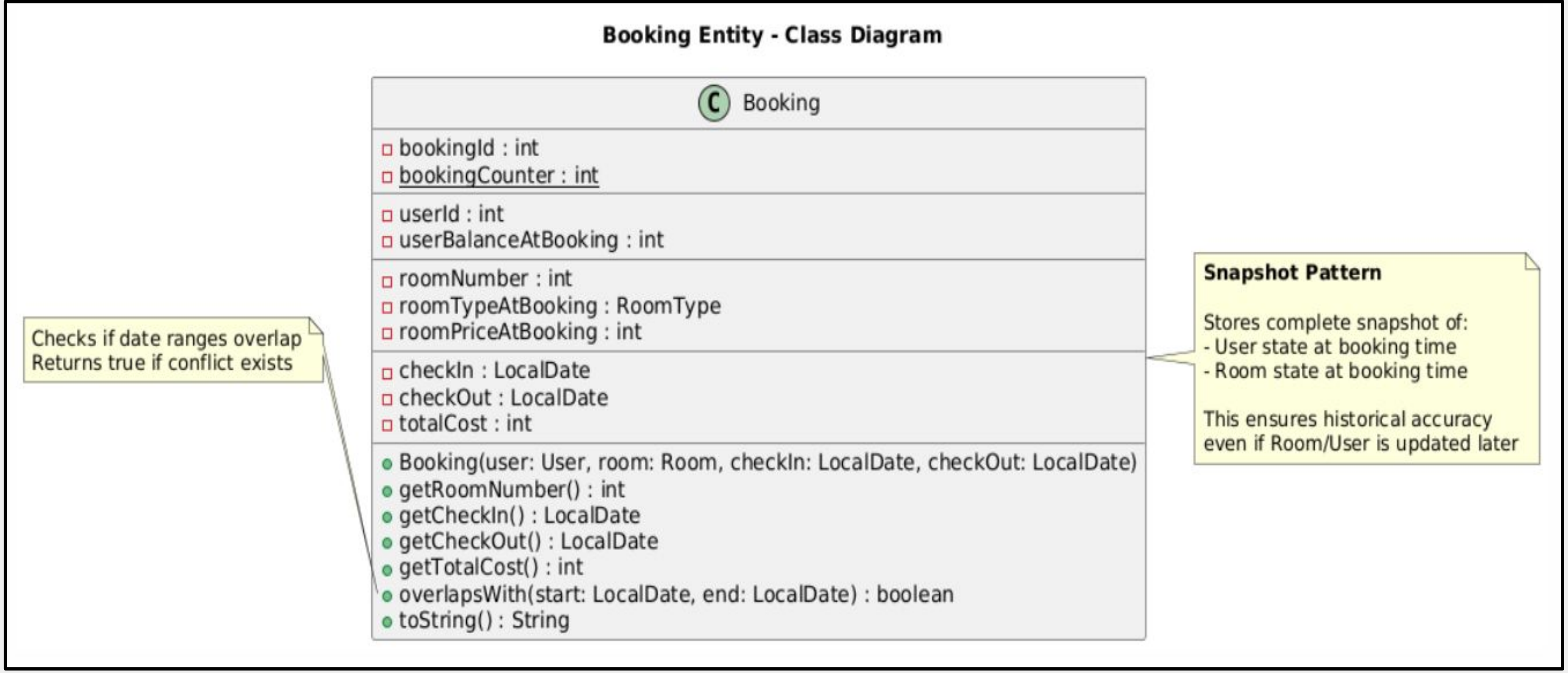

User Entity - Class Diagram
User
userId : int
balance : int
User(userId: int, balance: int)
getUserId() : int
getBalance() : int
deductBalance(amount: int) : void
toString() : String
Validation:
- balance >= 0
- Cannot deduct more than balance
Throws:
- IllegalArgumentException
if insufficient balance

User Entity - Implementation

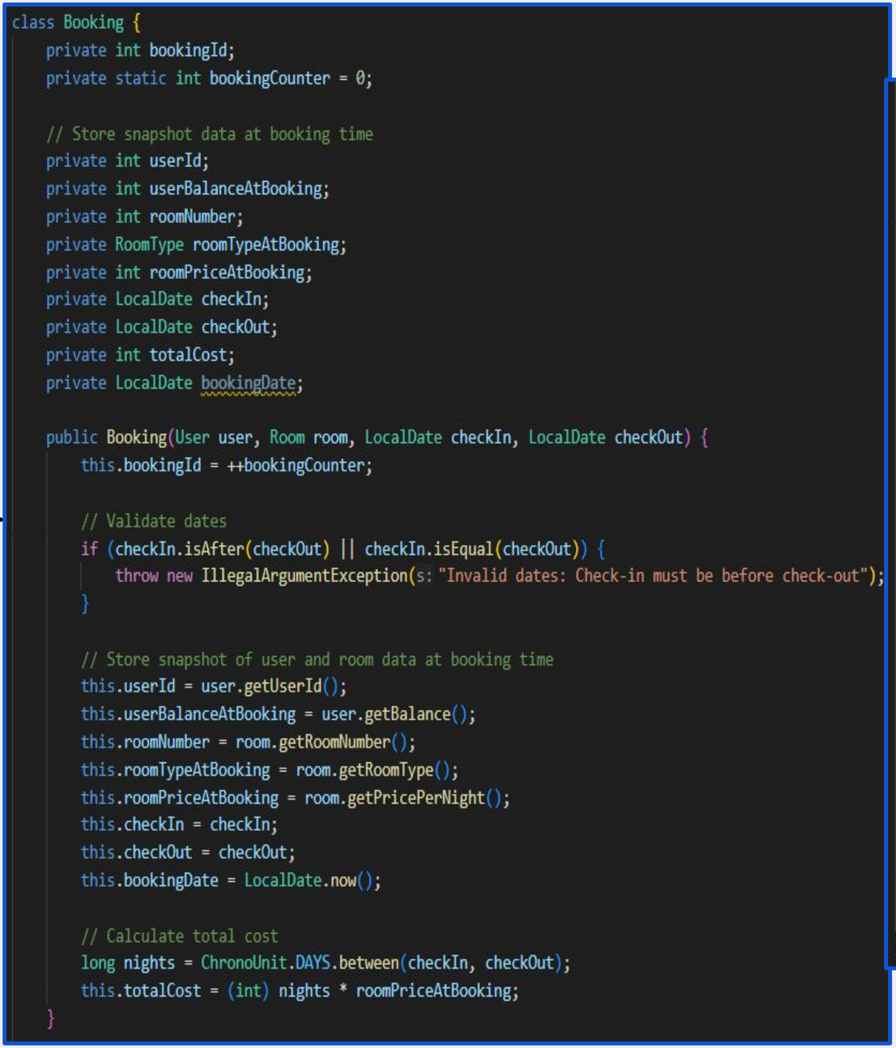


```java
class User {
    private int userId;
    private int balance;

    public User(int userId, int balance) {
        if (balance < 0) {
            throw new IllegalArgumentException(s: "Balance cannot be negative");
        }
        this.userId = userId;
        this.balance = balance;
    }

    public int getUserId() {
        return userId;
    }

    public int getBalance() {
        return balance;
    }

    public void deductBalance(int amount) {
        if (amount > balance) {
            throw new IllegalArgumentException(s: "Insufficient balance");
        }
        this.balance -= amount;
    }

    @Override
    public String toString() {
        return "User{ID=" + userId + ", Balance=" + balance + "}";
    }
}
```

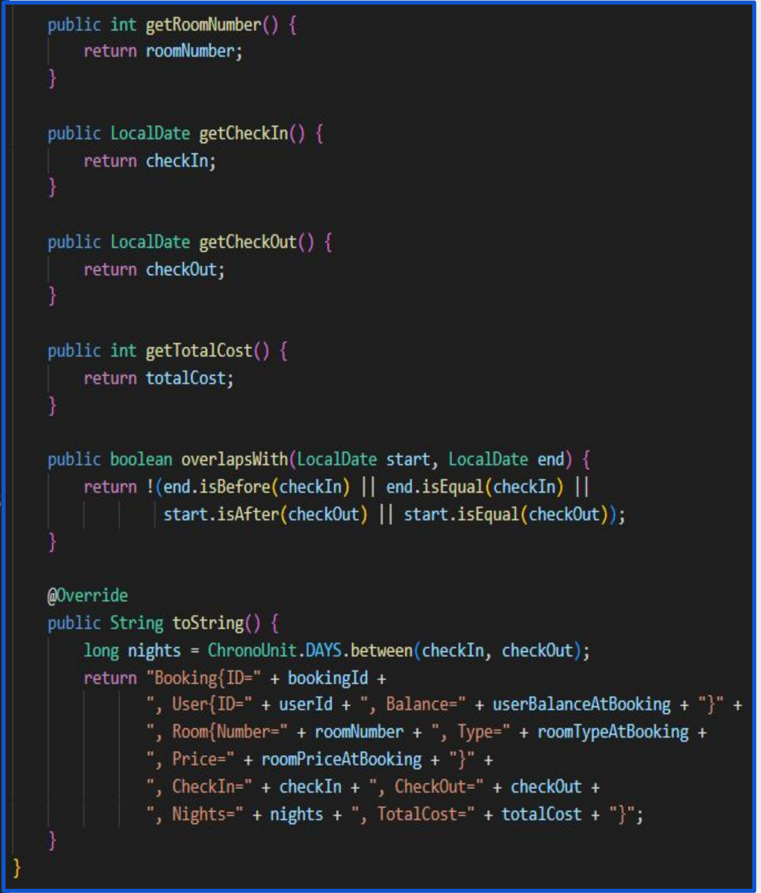
Booking Entity - Snapshot Pattern


Booking Entity - Class Diagram
Booking
bookingId : int
bookingCounter : int
userId : int
userBalanceAtBooking : int
roomNumber : int
roomTypeAtBooking : RoomType
roomPriceAtBooking : int
checkIn : LocalDate
checkOut : LocalDate
totalCost : int
Booking(user: User, room: Room, checkIn: LocalDate, checkOut: LocalDate)
getRoomNumber() : int
getCheckIn() : LocalDate
getCheckOut() : LocalDate
getTotalCost() : int
overlapsWith(start: LocalDate, end: LocalDate) : boolean
toString() : String
Checks if date ranges overlap
Returns true if conflict exists
Snapshot Pattern
Stores complete snapshot of:
- User state at booking time
- Room state at booking time
This ensures historical accuracy
even if Room/User is updated later

Booking Entity - Snapshot Pattern

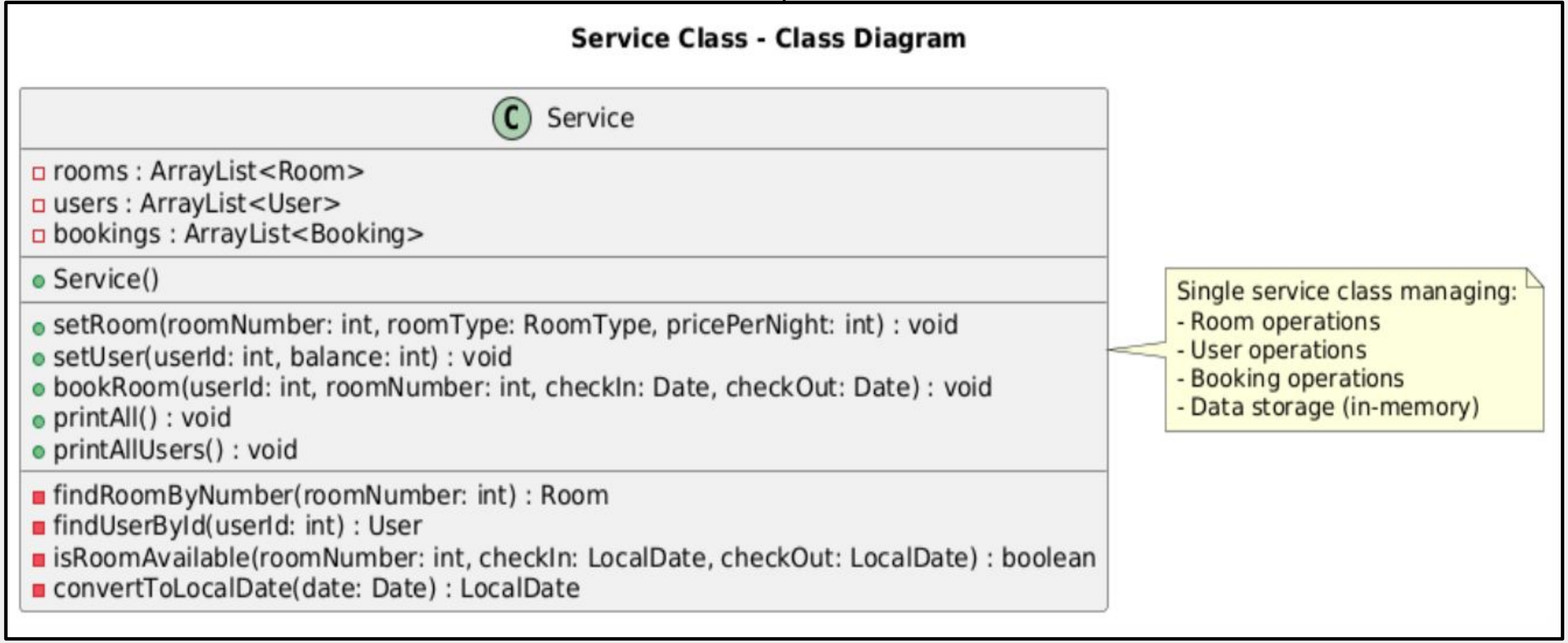


```java
class Booking {
    private int bookingId;
    private static int bookingCounter = 0;

    // Store snapshot data at booking time
    private int userId;
    private int userBalanceAtBooking;
    private int roomNumber;
    private RoomType roomTypeAtBooking;
    private int roomPriceAtBooking;
    private LocalDate checkIn;
    private LocalDate checkOut;
    private int totalCost;
    private LocalDate bookingDate;

    public Booking(User user, Room room, LocalDate checkIn, LocalDate checkOut) {
        this.bookingId = ++bookingCounter;

        // Validate dates
        if (checkIn.isAfter(checkOut) || checkIn.isEqual(checkOut)) {
            throw new IllegalArgumentException(s: "Invalid dates: Check-in must be before check-out");
        }

        // Store snapshot of user and room data at booking time
        this.userId = user.getUserId();
        this.userBalanceAtBooking = user.getBalance();
        this.roomNumber = room.getRoomNumber();
        this.roomTypeAtBooking = room.getRoomType();
        this.roomPriceAtBooking = room.getPricePerNight();
        this.checkIn = checkIn;
        this.checkOut = checkOut;
        this.bookingDate = LocalDate.now();

        // Calculate total cost
        long nights = ChronoUnit.DAYS.between(checkIn, checkOut);
        this.totalCost = (int) nights * roomPriceAtBooking;
    }
```



```java
    public int getRoomNumber() {
        return roomNumber;
    }

    public LocalDate getCheckIn() {
        return checkIn;
    }

    public LocalDate getCheckOut() {
        return checkOut;
    }

    public int getTotalCost() {
        return totalCost;
    }

    public boolean overlapsWith(LocalDate start, LocalDate end) {
        return !(end.isBefore(checkIn) || end.isEqual(checkIn) ||
                 start.isAfter(checkOut) || start.isEqual(checkOut));
    }

    @Override
    public String toString() {
        long nights = ChronoUnit.DAYS.between(checkIn, checkOut);
        return "Booking{ID=" + bookingId +
                ", User{ID=" + userId + ", Balance=" + userBalanceAtBooking + "}" +
                ", Room{Number=" + roomNumber + ", Type=" + roomTypeAtBooking +
                ", Price=" + roomPriceAtBooking + "}" +
                ", CheckIn=" + checkIn + ", CheckOut=" + checkOut +
                ", Nights=" + nights + ", TotalCost=" + totalCost + "}";
    }
}
```

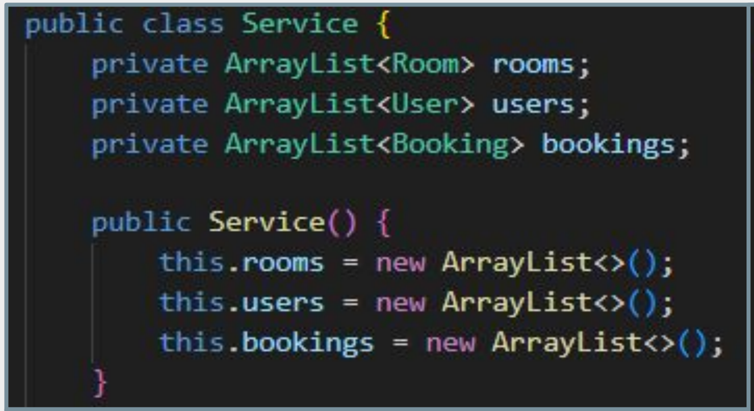
Service Layer Architecture



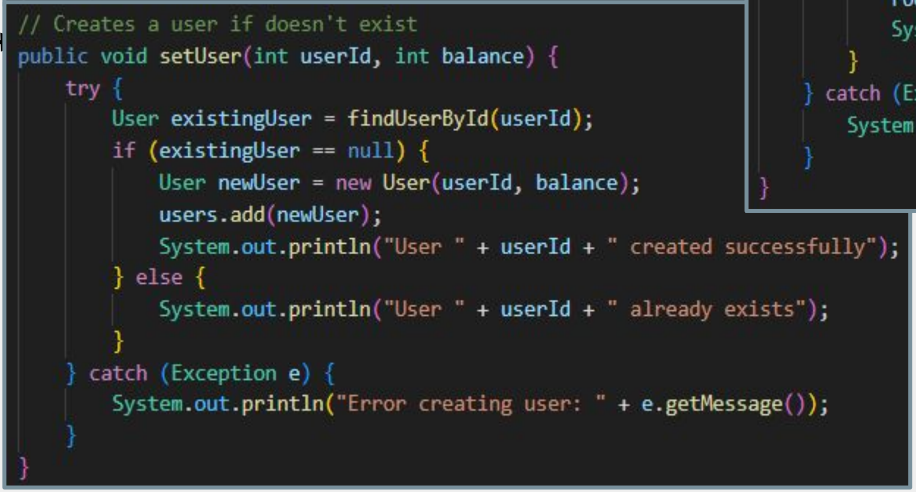
```
□ rooms : ArrayList<Room>
□ users : ArrayList<User>
□ bookings : ArrayList<Booking>

● Service()

● setRoom(roomNumber: int, roomType: RoomType, pricePerNight: int) : void
● setUser(userId: int, balance: int) : void
● bookRoom(userId: int, roomNumber: int, checkIn: Date, checkOut: Date) : void
● printAll() : void
● printAllUsers() : void

■ findRoomByNumber(roomNumber: int) : Room
■ findUserById(userId: int) : User
■ isRoomAvailable(roomNumber: int, checkIn: LocalDate, checkOut: LocalDate) : boolean
■ convertToLocalDate(date: Date) : LocalDate
```

Single service class managing:
- Room operations
- User operations
- Booking operations
- Data storage (in-memory)

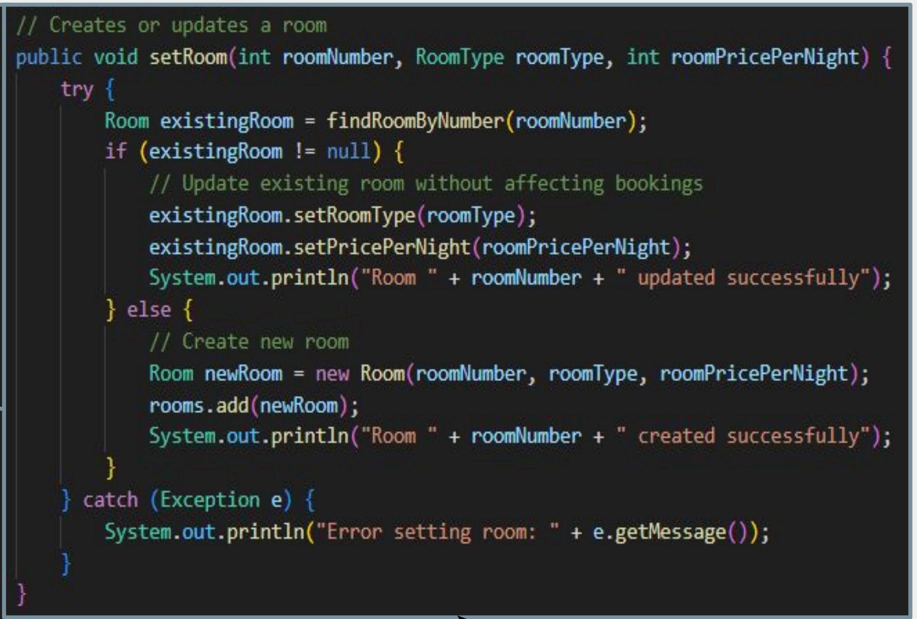
Service Layer Architecture



```java
public class Service {
    private ArrayList<Room> rooms;
    private ArrayList<User> users;
    private ArrayList<Booking> bookings;

    public Service() {
        this.rooms = new ArrayList<>();
        this.users = new ArrayList<>();
        this.bookings = new ArrayList<>();
    }
```



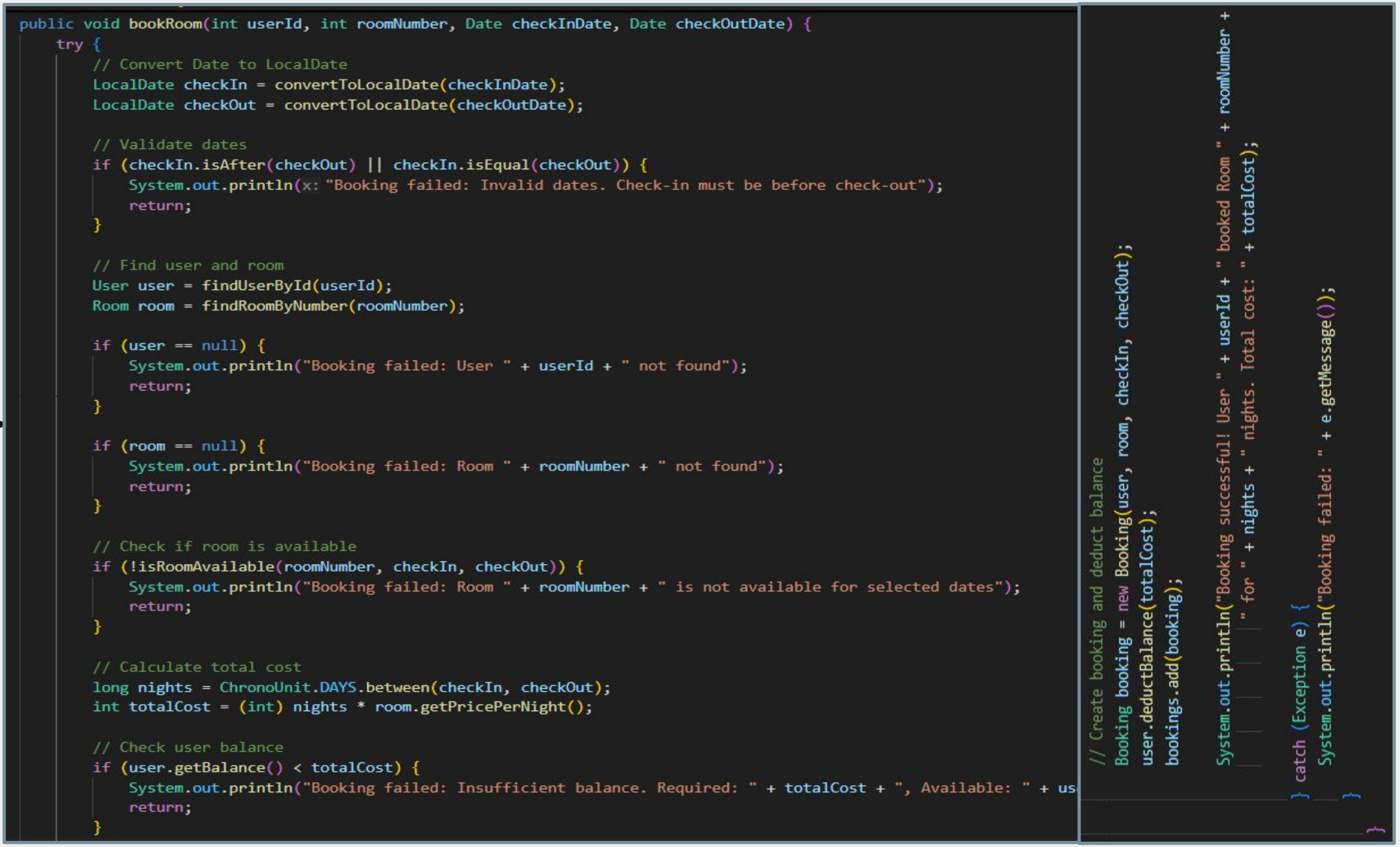
```java
// Creates a user if doesn't exist
public void setUser(int userId, int balance) {
    try {
        User existingUser = findUserById(userId);
        if (existingUser == null) {
            User newUser = new User(userId, balance);
            users.add(newUser);
            System.out.println("User " + userId + " created successfully");
        } else {
            System.out.println("User " + userId + " already exists");
        }
    } catch (Exception e) {
        System.out.println("Error creating user: " + e.getMessage());
    }
}
```



```java
// Creates or updates a room
public void setRoom(int roomNumber, RoomType roomType, int roomPricePerNight) {
    try {
        Room existingRoom = findRoomByNumber(roomNumber);
        if (existingRoom != null) {
            // Update existing room without affecting bookings
            existingRoom.setRoomType(roomType);
            existingRoom.setPricePerNight(roomPricePerNight);
            System.out.println("Room " + roomNumber + " updated successfully");
        } else {
            // Create new room
            Room newRoom = new Room(roomNumber, roomType, roomPricePerNight);
            rooms.add(newRoom);
            System.out.println("Room " + roomNumber + " created successfully");
        }
    } catch (Exception e) {
        System.out.println("Error setting room: " + e.getMessage());
    }
}
```

Updates room WITHOUT affecting past bookings (snapshot pattern)

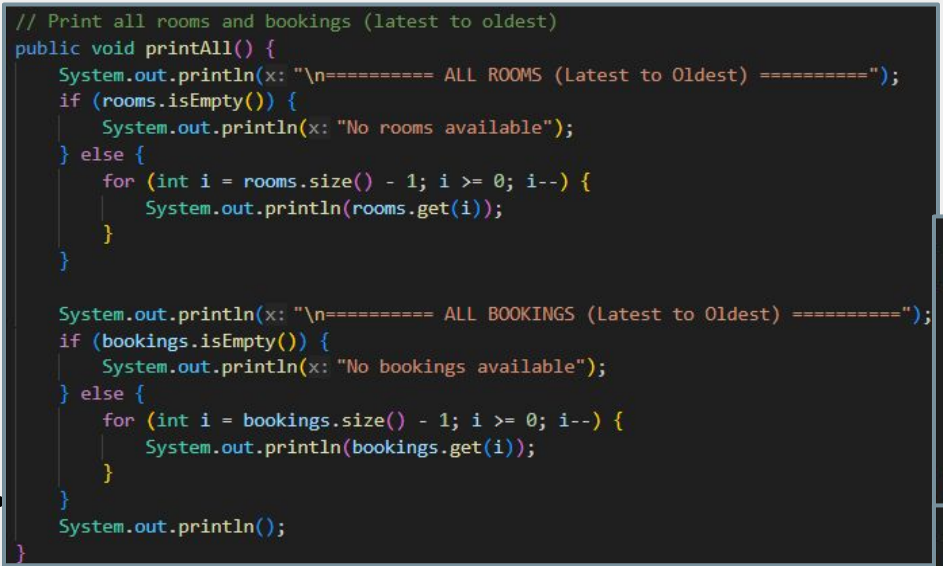
Booking Validation Logic



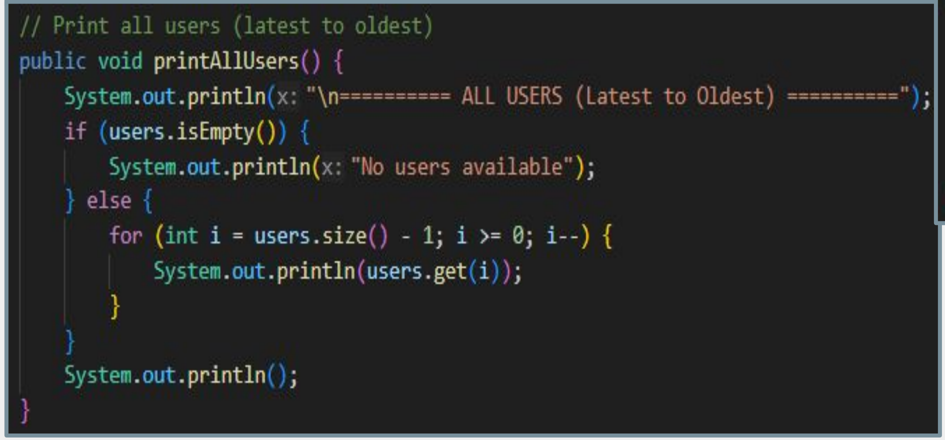
```java
public void bookRoom(int userId, int roomNumber, Date checkInDate, Date checkOutDate) {
    try {
        // Convert Date to LocalDate
        LocalDate checkIn = convertToLocalDate(checkInDate);
        LocalDate checkOut = convertToLocalDate(checkOutDate);

        // Validate dates
        if (checkIn.isAfter(checkOut) || checkIn.isEqual(checkOut)) {
            System.out.println(x: "Booking failed: Invalid dates. Check-in must be before check-out");
            return;
        }

        // Find user and room
        User user = findUserById(userId);
        Room room = findRoomByNumber(roomNumber);

        if (user == null) {
            System.out.println("Booking failed: User " + userId + " not found");
            return;
        }

        if (room == null) {
            System.out.println("Booking failed: Room " + roomNumber + " not found");
            return;
        }

        // Check if room is available
        if (!isRoomAvailable(roomNumber, checkIn, checkOut)) {
            System.out.println("Booking failed: Room " + roomNumber + " is not available for selected dates");
            return;
        }

        // Calculate total cost
        long nights = ChronoUnit.DAYS.between(checkIn, checkOut);
        int totalCost = (int) nights * room.getPricePerNight();

        // Check user balance
        if (user.getBalance() < totalCost) {
            System.out.println("Booking failed: Insufficient balance. Required: " + totalCost + ", Available: " + us
            return;
        }
```

```java
        // Create booking and deduct balance
        Booking booking = new Booking(user, room, checkIn, checkOut);
        user.deductBalance(totalCost);
        bookings.add(booking);

        System.out.println("Booking successful! User " + userId + " booked Room " + roomNumber +
            " for " + nights + " nights. Total cost: " + totalCost);
    } catch (Exception e) {
        System.out.println("Booking failed: " + e.getMessage());
    }
}
```

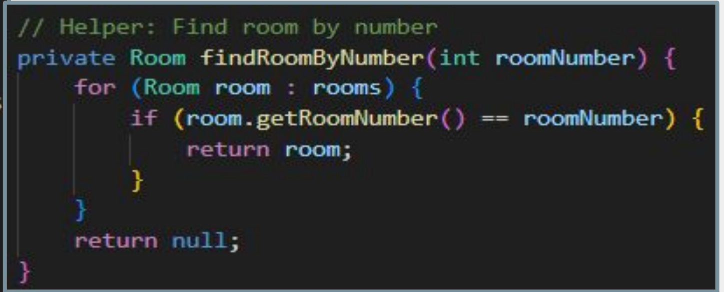
Booking Validation Logic



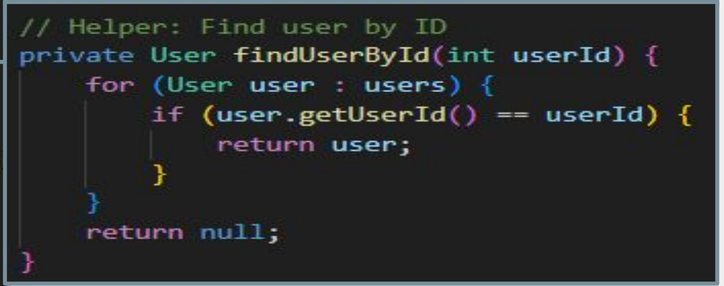
```java
// Print all rooms and bookings (latest to oldest)
public void printAll() {
    System.out.println(x: "\n========== ALL ROOMS (Latest to Oldest) ==========");
    if (rooms.isEmpty()) {
        System.out.println(x: "No rooms available");
    } else {
        for (int i = rooms.size() - 1; i >= 0; i--) {
            System.out.println(rooms.get(i));
        }
    }

    System.out.println(x: "\n========== ALL BOOKINGS (Latest to Oldest) ==========");
    if (bookings.isEmpty()) {
        System.out.println(x: "No bookings available");
    } else {
        for (int i = bookings.size() - 1; i >= 0; i--) {
            System.out.println(bookings.get(i));
        }
    }
    System.out.println();
}
```



```java
// Print all users (latest to oldest)
public void printAllUsers() {
    System.out.println(x: "\n========== ALL USERS (Latest to Oldest) ==========");
    if (users.isEmpty()) {
        System.out.println(x: "No users available");
    } else {
        for (int i = users.size() - 1; i >= 0; i--) {
            System.out.println(users.get(i));
        }
    }
    System.out.println();
}
```

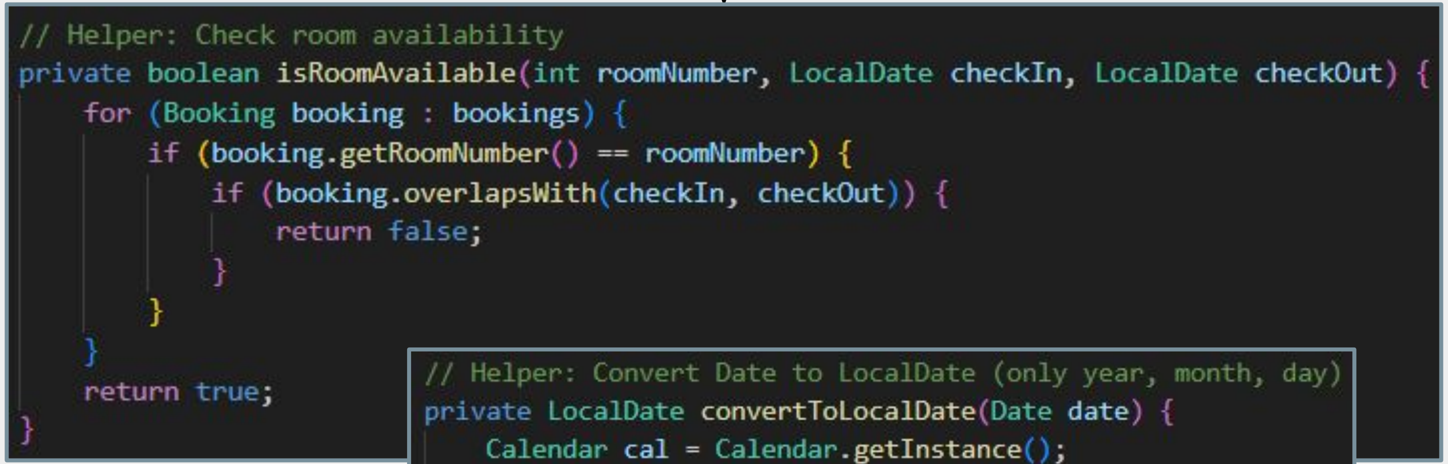


```java
// Helper: Find room by number
private Room findRoomByNumber(int roomNumber) {
    for (Room room : rooms) {
        if (room.getRoomNumber() == roomNumber) {
            return room;
        }
    }
    return null;
}
```

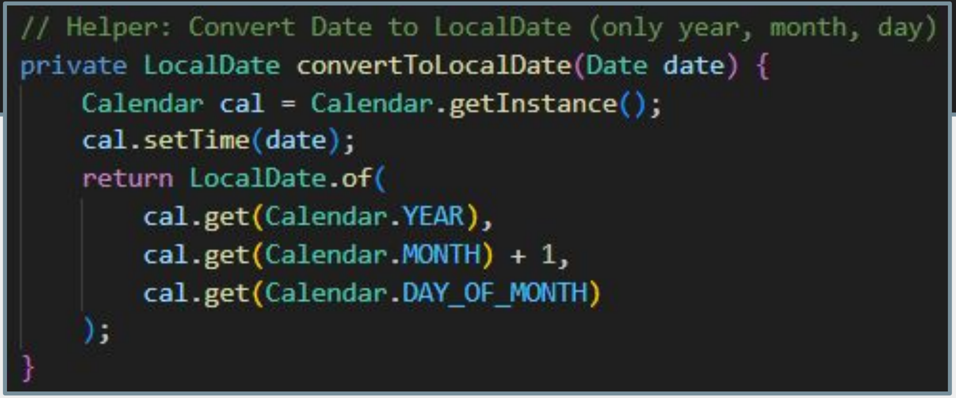


```java
// Helper: Find user by ID
private User findUserById(int userId) {
    for (User user : users) {
        if (user.getUserId() == userId) {
            return user;
        }
    }
    return null;
}
```

Room Availability Check

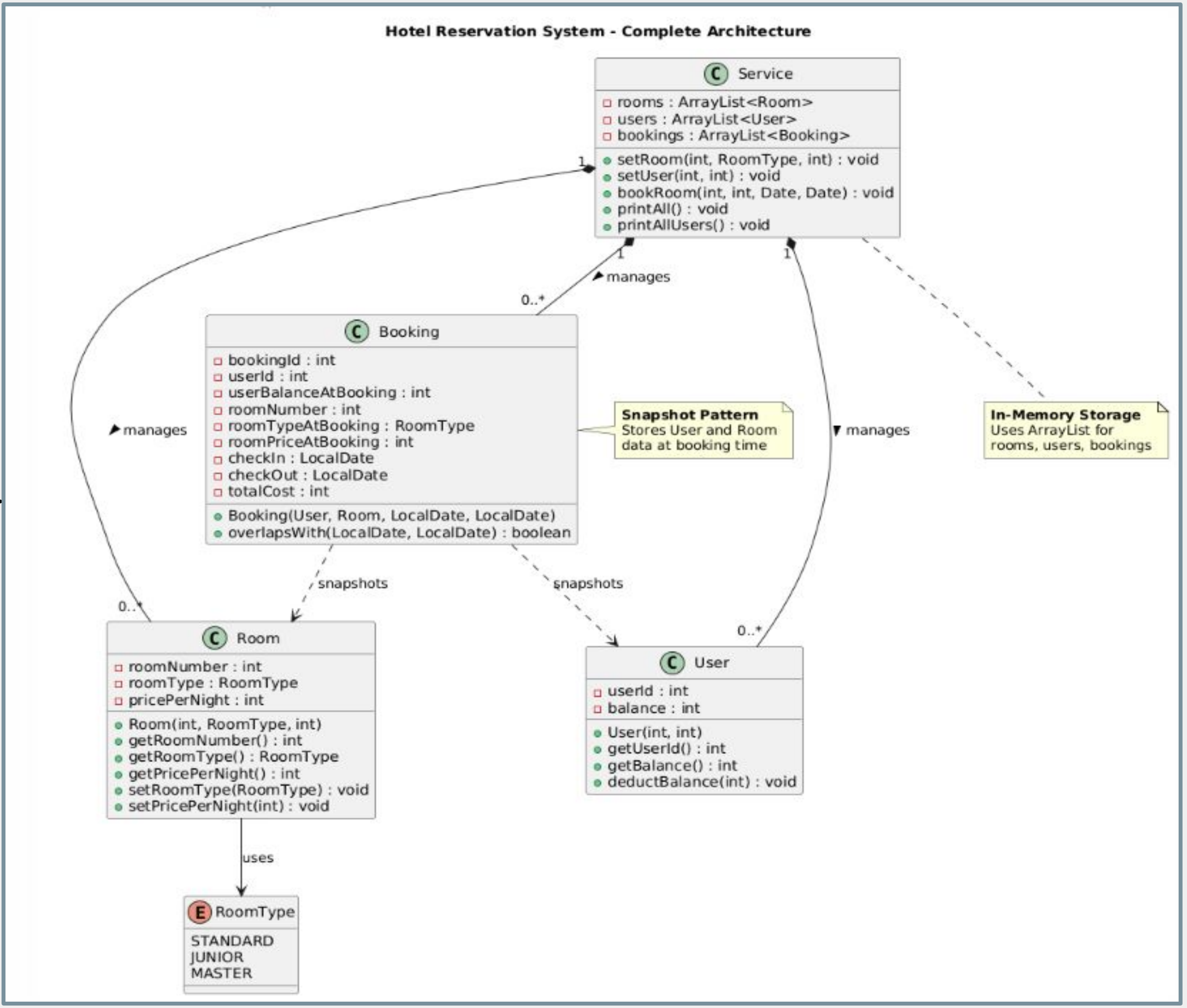


```java
// Helper: Check room availability
private boolean isRoomAvailable(int roomNumber, LocalDate checkIn, LocalDate checkOut) {
    for (Booking booking : bookings) {
        if (booking.getRoomNumber() == roomNumber) {
            if (booking.overlapsWith(checkIn, checkOut)) {
                return false;
            }
        }
    }
    return true;
}
```



```java
// Helper: Convert Date to LocalDate (only year, month, day)
private LocalDate convertToLocalDate(Date date) {
    Calendar cal = Calendar.getInstance();
    cal.setTime(date);
    return LocalDate.of(
        cal.get(Calendar.YEAR),
        cal.get(Calendar.MONTH) + 1,
        cal.get(Calendar.DAY_OF_MONTH)
    );
}
```

Complete System Architecture



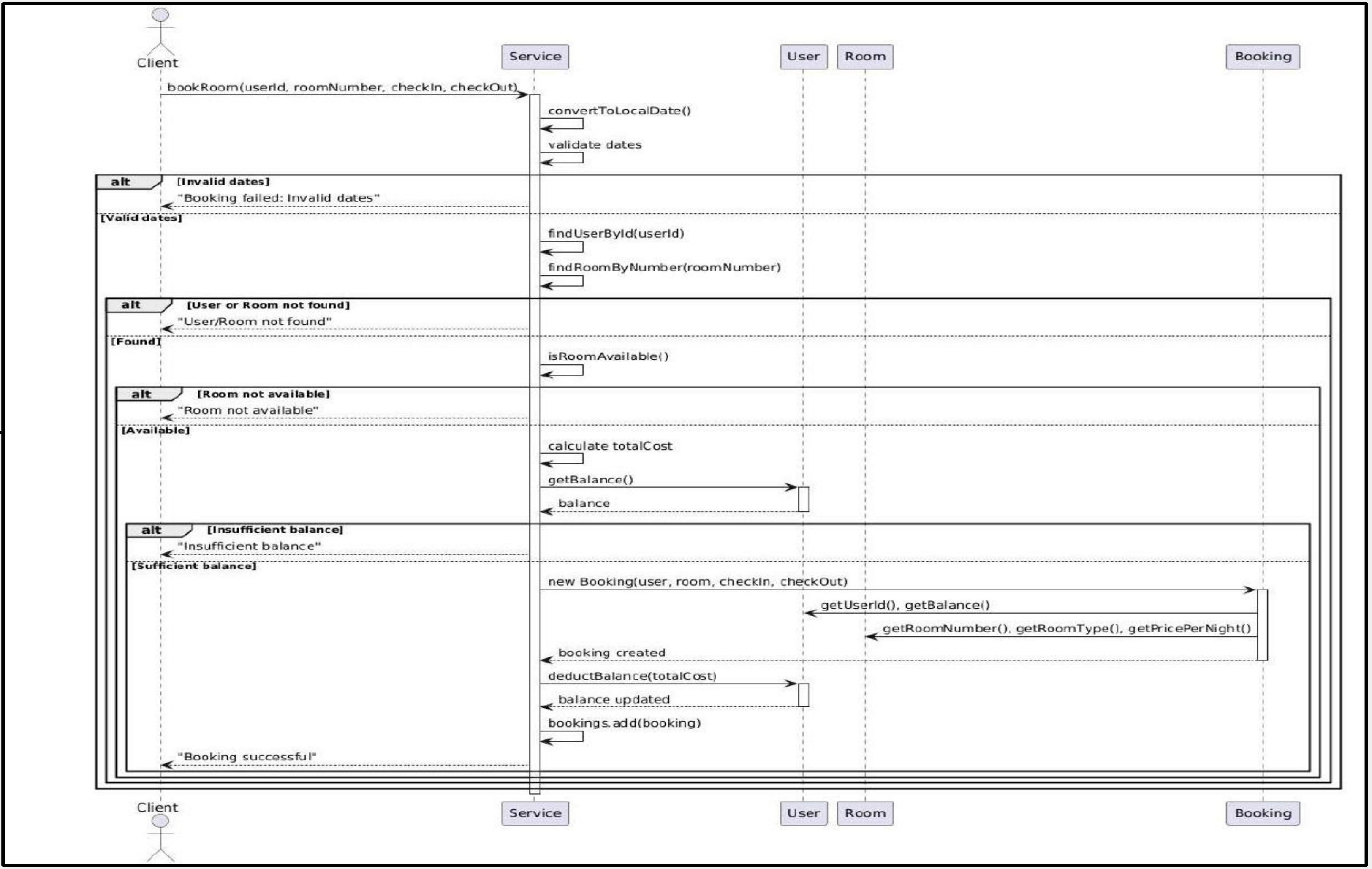
All 4 classes (Room, User, Booking, Service)

Relationships and cardinality

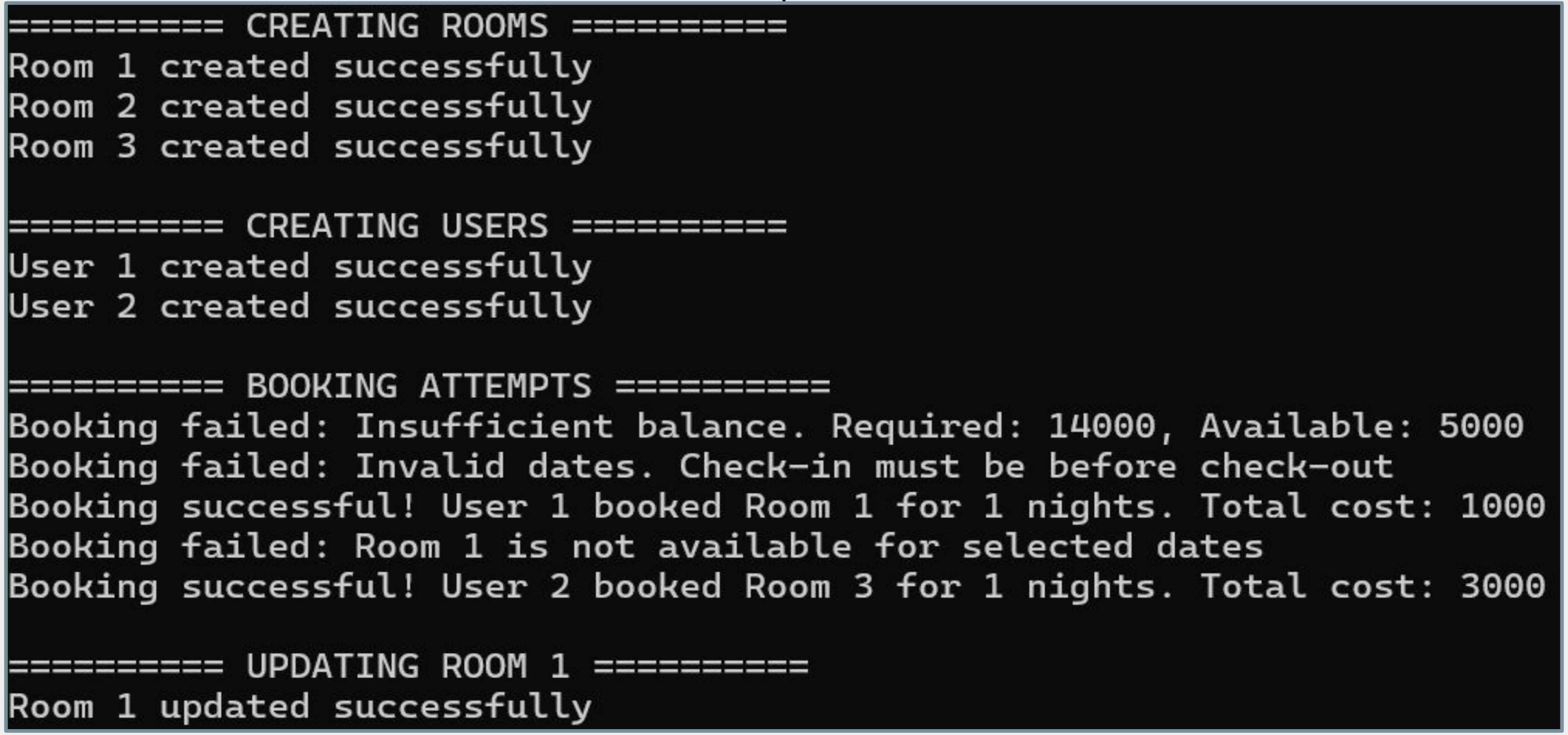
Key methods and attributes

Dependencies

Booking Sequence Diagram



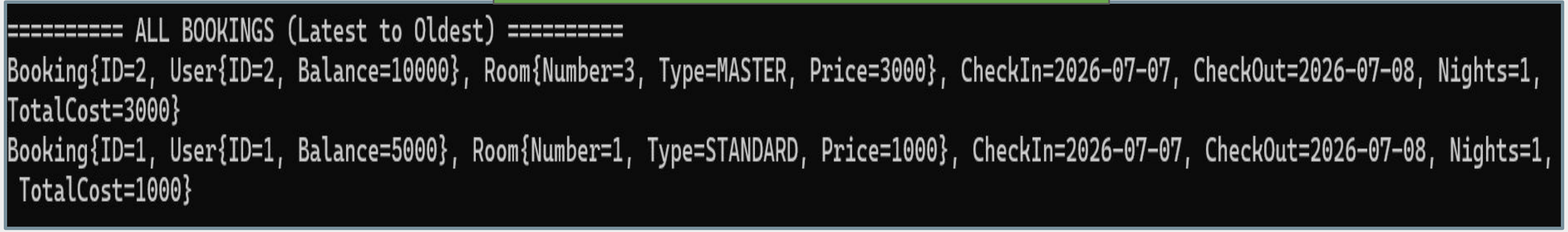
Test Results - Console Output



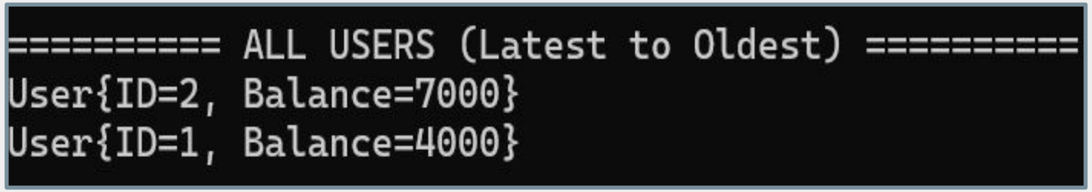
```
=========== CREATING ROOMS ===========
Room 1 created successfully
Room 2 created successfully
Room 3 created successfully

=========== CREATING USERS ===========
User 1 created successfully
User 2 created successfully

=========== BOOKING ATTEMPTS ===========
Booking failed: Insufficient balance. Required: 14000, Available: 5000
Booking failed: Invalid dates. Check-in must be before check-out
Booking successful! User 1 booked Room 1 for 1 nights. Total cost: 1000
Booking failed: Room 1 is not available for selected dates
Booking successful! User 2 booked Room 3 for 1 nights. Total cost: 3000

=========== UPDATING ROOM 1 ===========
Room 1 updated successfully
```

Final State

printAll() - Bookings Output



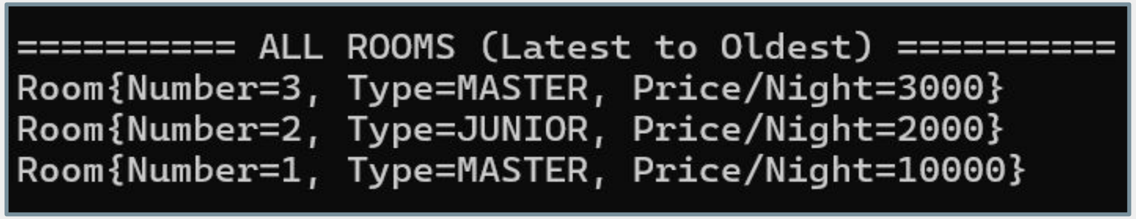
```
========== ALL BOOKINGS (Latest to Oldest) ==========
Booking{ID=2, User{ID=2, Balance=10000}, Room{Number=3, Type=MASTER, Price=3000}, CheckIn=2026-07-07, CheckOut=2026-07-08, Nights=1,
TotalCost=3000}
Booking{ID=1, User{ID=1, Balance=5000}, Room{Number=1, Type=STANDARD, Price=1000}, CheckIn=2026-07-07, CheckOut=2026-07-08, Nights=1,
 TotalCost=1000}
```



```
========== ALL USERS (Latest to Oldest) ==========
User{ID=2, Balance=7000}
User{ID=1, Balance=4000}
```

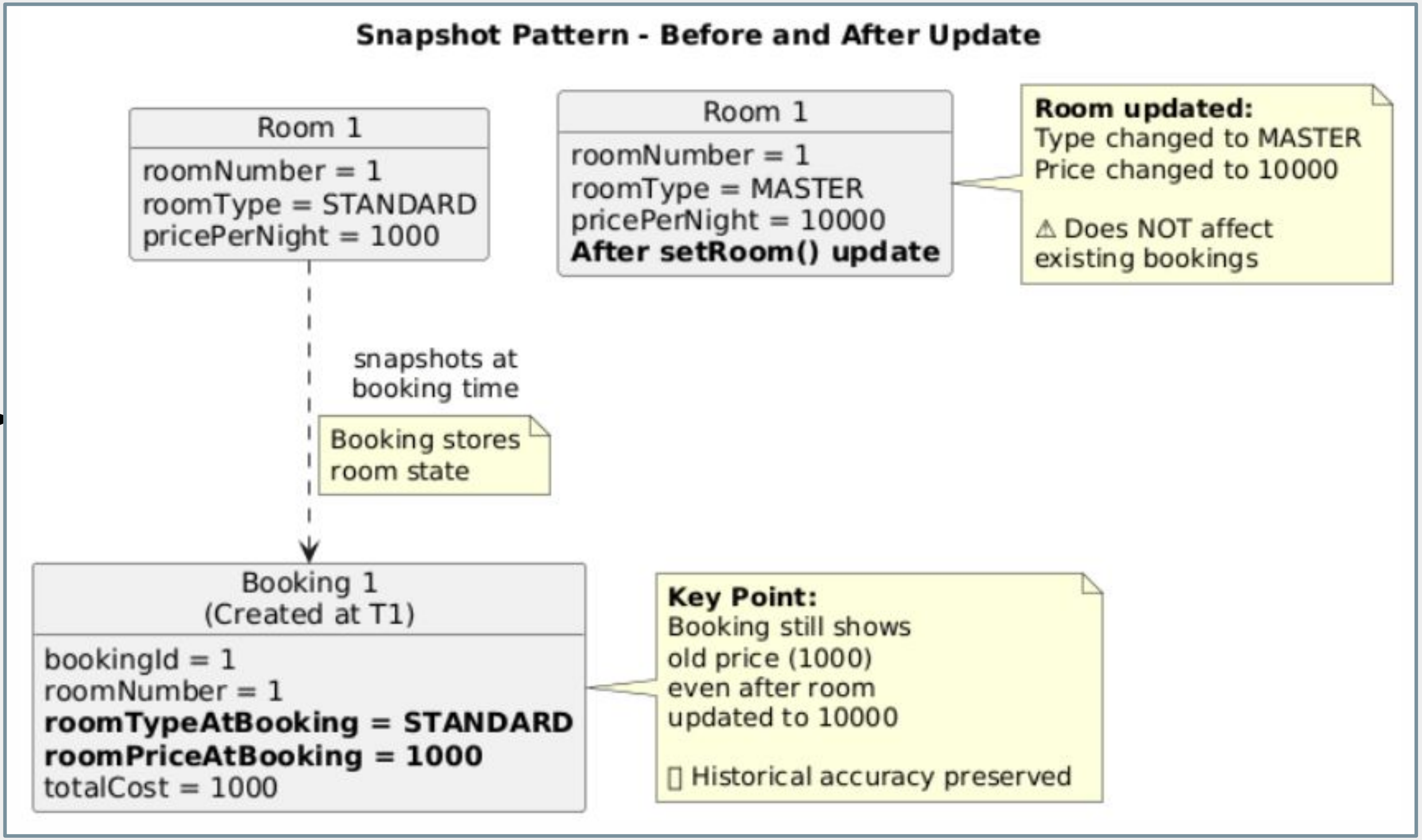
printAllUsers() Output

printAll() - Rooms Output

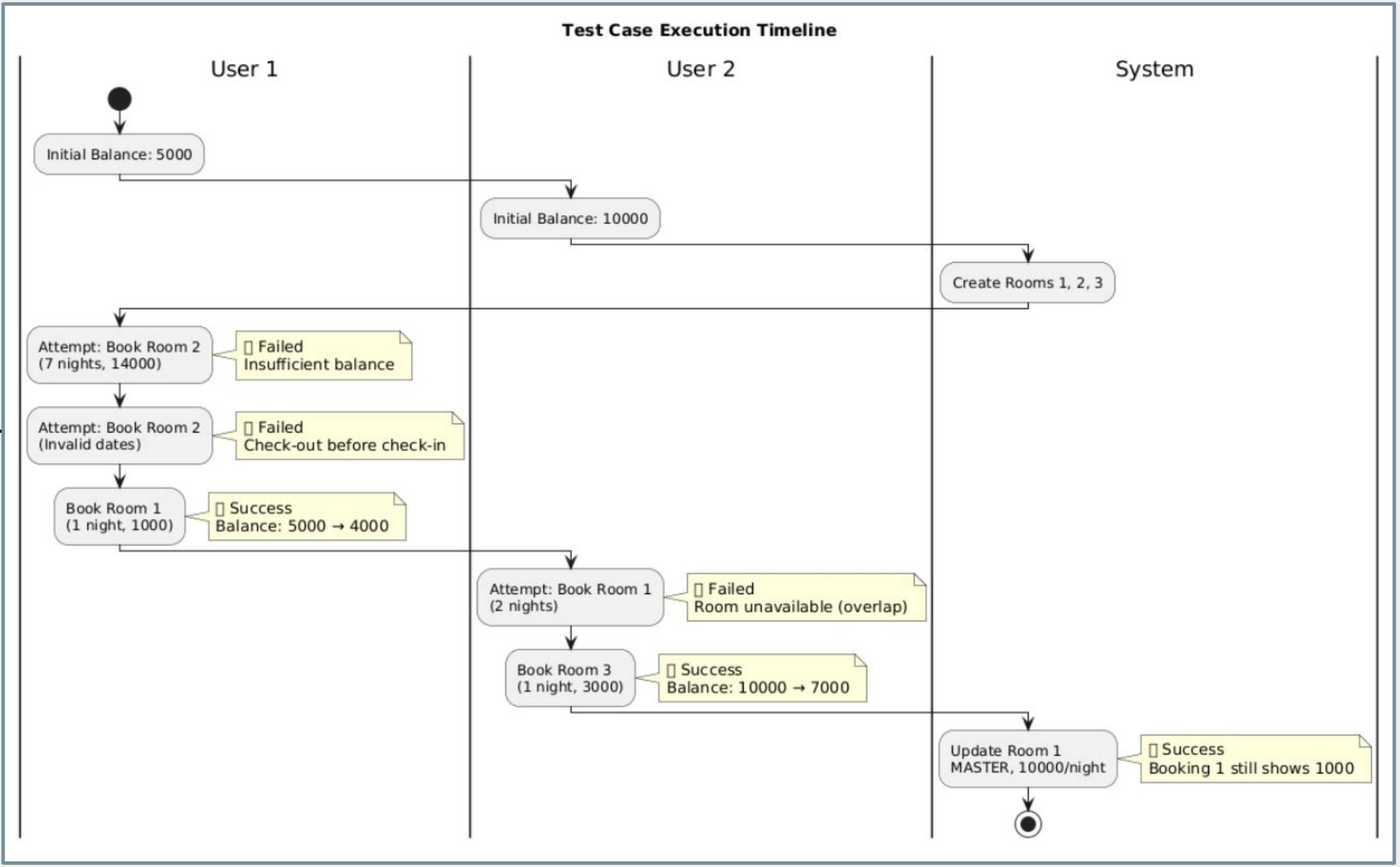


```
========== ALL ROOMS (Latest to Oldest) ==========
Room{Number=3, Type=MASTER, Price/Night=3000}
Room{Number=2, Type=JUNIOR, Price/Night=2000}
Room{Number=1, Type=MASTER, Price/Night=10000}
```

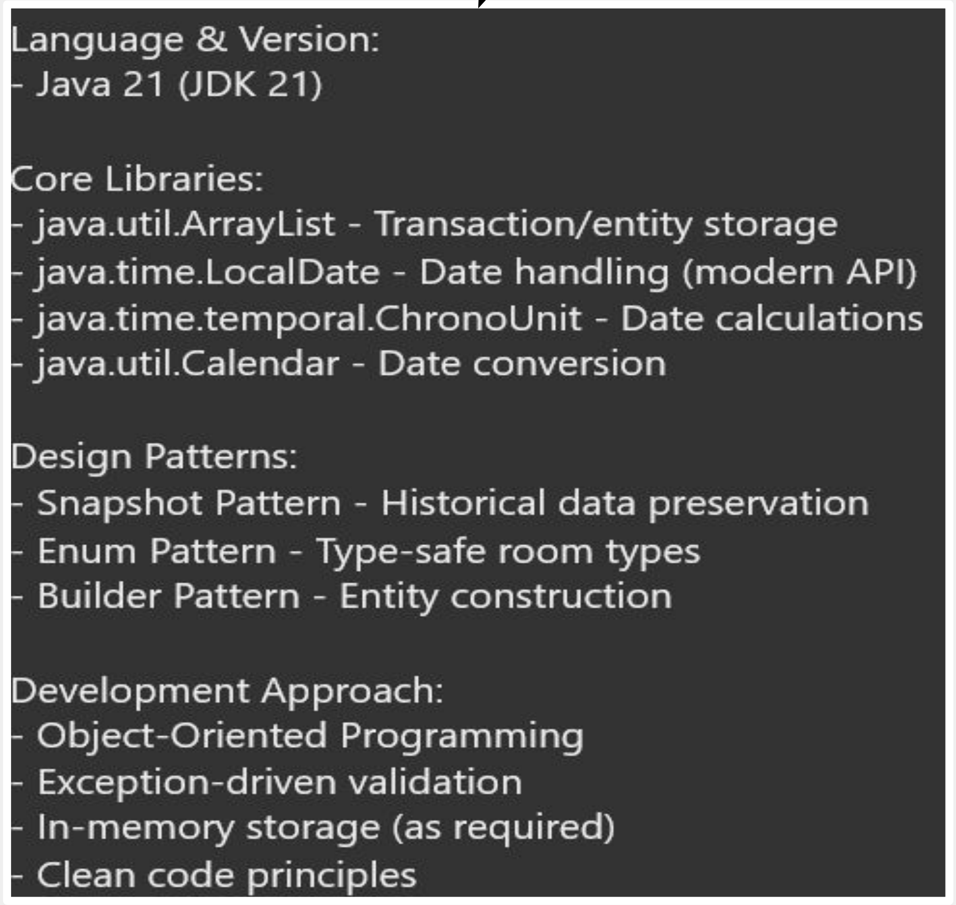
Snapshot Pattern Diagram


Snapshot Pattern - Before and After Update
Room 1
roomNumber = 1
roomType = STANDARD
pricePerNight = 1000
Room 1
roomNumber = 1
roomType = MASTER
pricePerNight = 10000
After setRoom() update
Room updated:
Type changed to MASTER
Price changed to 10000
⚠ Does NOT affect existing bookings
snapshots at booking time
Booking stores room state
Booking 1
(Created at T1)
bookingId = 1
roomNumber = 1
roomTypeAtBooking = STANDARD
roomPriceAtBooking = 1000
totalCost = 1000
Key Point:
Booking still shows old price (1000) even after room updated to 10000
▯ Historical accuracy preserved

Test Case Timeline


Test Case Execution Timeline
User 1
User 2
System
Initial Balance: 5000
Initial Balance: 10000
Create Rooms 1, 2, 3
Attempt: Book Room 2 (7 nights, 14000)
Failed Insufficient balance
Attempt: Book Room 2 (Invalid dates)
Failed Check-out before check-in
Book Room 1 (1 night, 1000)
Success Balance: 5000 → 4000
Attempt: Book Room 1 (2 nights)
Failed Room unavailable (overlap)
Book Room 3 (1 night, 3000)
Success Balance: 10000 → 7000
Update Room 1 MASTER, 10000/night
Success Booking 1 still shows 1000

Technologies Used



Language & Version:
- Java 21 (JDK 21)

Core Libraries:
- java.util.ArrayList - Transaction/entity storage
- java.time.LocalDate - Date handling (modern API)
- java.time.temporal.ChronoUnit - Date calculations
- java.util.Calendar - Date conversion

Design Patterns:
- Snapshot Pattern - Historical data preservation
- Enum Pattern - Type-safe room types
- Builder Pattern - Entity construction

Development Approach:
- Object-Oriented Programming
- Exception-driven validation
- In-memory storage (as required)
- Clean code principles

Bonus Question 1

Should we put all functions in the same service class?

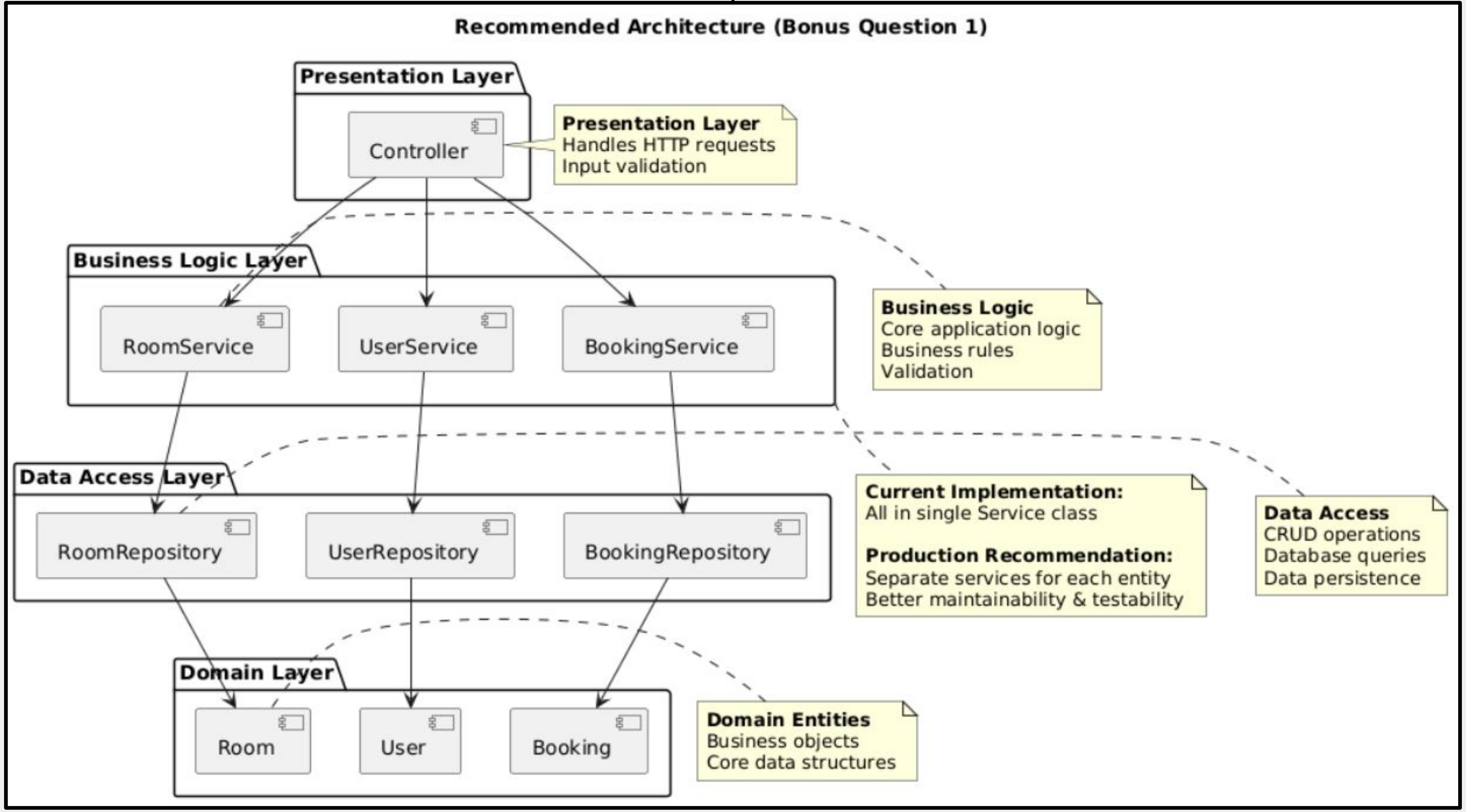
Answer	**NO - Not recommended for production**

Issues with Current Approach: ❌ Violates Single Responsibility Principle ❌ Poor separation of concerns ❌ Difficult to test individual components ❌ Hard to scale and maintain

For this test with limited scope, single service is acceptable, but production apps need proper layered architecture.

```
Recommended Structure:
├── Entities (Room, User, Booking)
├── Repositories (Data access)
├── Services (Business logic)
└── Controllers (Request handling)
```

Bonus Question 1



Bonus Question 2

Alternative approaches to handle room updates without affecting bookings?

Current Approach:	Snapshot Pattern

How it works:

- Booking stores complete snapshot of room/user data at booking time
- Updates to Room entity don't affect historical booking data

Advantages

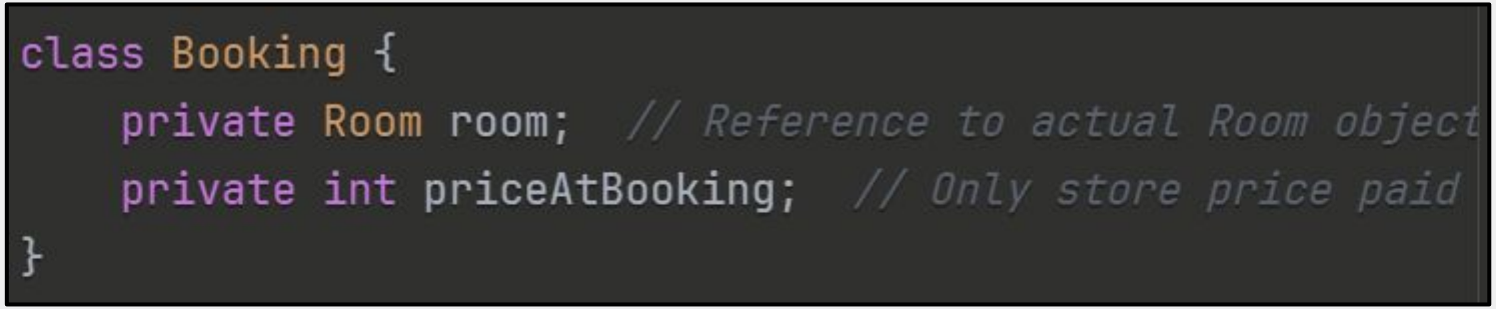
✅ Complete historical accuracy
✅ Financial integrity (audit trail)
✅ No cascading updates needed
✅ Simple to understand

Disadvantages

❌ Data duplication
❌ More storage required

Alternative Approach - Reference Pattern

How it works:



```
class Booking {
    private Room room;   // Reference to actual Room object
    private int priceAtBooking;   // Only store price paid
}
```

Advantages

✅ No data duplication
✅ Always shows current room info
✅ Easier consistency

Disadvantages

❌ Loses historical context
❌ Still need price snapshot for billing
❌ Complex room deletion handling

My Recommendation - Snapshot Pattern

Why Snapshot Pattern is Best:

1- Financial Integrity 🏦
✅ Accounting and auditing requirements
✅ Tax compliance
✅ Dispute resolution

2- Historical Accuracy 📊
✅ Know exact booking conditions
✅ Immutable transaction records

3- Business Logic 💼
✅ Past bookings shouldn't change
✅ Customer expectations preserved

Performance ⚡
✅ No joins needed for history
✅ Self-contained data

For hotel/payment systems (like Skypay), snapshot pattern prioritizes data integrity over storage efficiency.

Conclusion

What Was Achieved:

✅ Fully functional Hotel Reservation System
✅ All requirements met and tested
✅ Clean, maintainable code structure
✅ Proper exception handling
✅ Historical data integrity preserved
✅ Thoughtful design decisions explained

Key Takeaways:

- Snapshot pattern ensures financial
- accuracy Proper validation prevents data
- corruption OOP principles create maintainable code
- Balance between simplicity and best practices