Common

Cover all the LLD here

|  |
| --- |
| Hotel booking LLD  <https://www.lldcoding.com/design-lld-a-hotel-management-system-machine-coding-interview> |
| Answer: |

**Features**

We’ll focus on the following set of requirements while designing the Hotel Management System:

1. The system should support the booking of different room types like standard, deluxe, family suite, etc.
2. Guests should be able to search the room inventory and book any available room.
3. The system should be able to retrieve information, such as who booked a particular room, or what rooms were booked by a specific customer.
4. The system should allow customers to cancel their booking - and provide them with a full refund if the cancelation occurs before 24 hours of the check-in date.
5. The system should be able to send notifications whenever the booking is nearing the check-in or check-out date.
6. The system should maintain a room housekeeping log to keep track of all housekeeping tasks.
7. Any customer should be able to add room services and food items.
8. Customers can ask for different amenities.
9. The customers should be able to pay their bills through credit card, check or cash.

Questions in my mind:

1. What will be cost of the room ,and how we will calculate the room cost where should we store and mapping of room cost

Entity:

|  |
| --- |
| Enum RoomType: STANDARD, DELUX, FAMILY SUIT |
| HouseKeeping  -name  -actions  -desc  -date :time |
| Room:  -roomNumber  -roomType  -desc  -houseKeepingList:HouseKeeping  -addhousekeepingTask()  -gethouseKeepingTask() |
| Booking:  bookingId  HotelId:  HotelName:  Address:  totalRoom  checkingDate, checkingTime, checkoutDate,checkoutTime |
| Hotel  List<Room> rooms  addRoom  removeRoom |
| RoomSearchService  //how I will search or store rooms based on booked or available for the following date  availableRoom(hotelId,roomType,fromDate,toDate)  roomStatus(hotelId,roomId,date) |
| SystemRoomService   * roomStatus(hotelId,roomNumber,date) * searchRoomBookedByUser(hotelId,customerId) :List<Booking> pageable 10 |
| BookingService  -dobooking()  --addRoomService(bookingId,roomService)  --addFoodItem(bookingId,foodItems)  addAmenities(bookingId,ListOfAmenities)  -cancelBooking |
| Notification Service  sendNotification(booking) |
| PaymentService  payBill()  ? Have a payment class |

**Step 1: Clarify and Break Down Requirements**

**🎯 Core Features**

1. **Room Types & Inventory**
   * Standard, Deluxe, Family Suite
   * Room availability and occupancy tracking
2. **Search and Booking**
   * Search rooms by date, type, availability
   * Book available rooms
3. **Customer Details & Bookings**
   * Retrieve who booked what room
   * Get all rooms booked by a specific customer
4. **Cancellation & Refunds**
   * Cancel before 24 hours → full refund
   * After that → no refund / partial refund (optional)
5. **Notifications**
   * Nearing check-in/check-out reminders
   * Payment confirmation, service updates, etc.
6. **Housekeeping Log**
   * Track housekeeping tasks per room and date
7. **Room Services & Food Orders**
   * Add food/room service requests to a booking
8. **Amenities Requests**
   * Request for extra beds, towels, toiletries, etc.
9. **Billing & Payment**
   * Multiple payment modes: card, check, cash
   * Invoice generation

**✅ Step 2: Entities to Think About**

| **Entity** | **Responsibilities** |
| --- | --- |
| Customer | Profile, bookings, contact info |
| Room | Room number, type, status (Available, Booked, Under maintenance) |
| Booking | Customer, room, check-in/out date, status, payment |
| RoomType | ENUM or class to define Standard, Deluxe, Suite |
| Payment | Amount, method, payment status |
| HousekeepingLog | Tasks done on a room with timestamps |
| ServiceRequest | Food, laundry, maintenance requests |
| AmenityRequest | Extra bed, toiletries, etc. |
| Notification | Email/SMS triggers with time-based logic |

Step 4: Design Considerations

🧱 System Design Prep

Think in terms of microservices or modular monolith

Database schema: normalized with proper indexing for availability and booking

Concurrency control: avoid double-booking (locking or optimistic concurrency)

Scheduler: for notification and housekeeping job runners

Audit & Logging: payment, housekeeping, service logs

|  |
| --- |
|  |
| class Customer  {  String id;  String name;  String contacInfo  String email;  List<Booking>  } |
| class Room{  String roonNumber;  RoomType roomType;  RoomStatus roomStatus;  }  enum RoomType {STANDAR, DELUX , SUITE}  enum RoomStatus: {BOOKED, AVAILABLE, MAINTENANCE}  enum PaymentStatus: PENDING,COMPLETE,PARTIAL  enum PaymentMethod: CREID,DEBIT,CASH |
| Class Booking{  bookingId, bookingDate; Customer, Room, checkIn, checkout,BookingStatus, payment, List<ServiceRequest> serviceRequestList, List<AmenityRequest>,totalAmount  } |
| Class Payment{  paymentId,  bookingId,  amount:double  payment date  paymentStatus  paymentMethod:  }  PaymentSummary{  bookingId,  totalAmount:double  amountPaidTillNow:Double  PaymentStatus: status  List<PaymentTransaction>  }  PaymentTransaction{  transactionId  paidAt  paymentMethod  TransactionId  }  So booking will not have payment information |
| Class HouseKeepingLog  {  Room room;  LocalDateTime;  String task;  }  class ServiceRequest {  String description;  LocalDateTime requestedAt;  }  class AmenityRequest {  AmenityType type;  LocalDateTime requestedAt;  }  Class BillingService  {  generateInvoice()  processPayment()  }  Class Scheduler  {  fetchAllBookingNearCheckingDate()  fetchAllBookingNearCheckoutDate()  } |

Done with the class

Now Interface and Services

Design pattern

Observer Pattern

Send notifications when booking is near check-in or canceled.

booking.addObserver(notificationService);

booking.cancel(); // triggers observers to notify

Add notification service in booking serice

We used SOLID principle very good

## SOLID Principles in Hotel Booking System

### **S - Single Responsibility Principle**

A class should have only one reason to change.

| **Class** | **Responsibility** |
| --- | --- |
| BookingService | Handle booking logic (search, cancel, confirm) |
| PaymentService | Handle payments/refunds |
| NotificationService | Handle message/email/sms sending |
| RoomService | Handle room availability, status updates |
| HousekeepingService | Track housekeeping logs |

### **L - Liskov Substitution Principle**

Subtypes must be substitutable for their base types.

* Use PaymentMethod subclasses for UPI, CreditCard, Cash — they all follow same interface.

**I - Interface Segregation Principle**

Clients should not be forced to depend on methods they don't use.

Split large interfaces like this:

interface BookingCancellation {

RefundResponse cancelBooking(String bookingId);

}

interface BookingSearch {

List<Room> searchRooms(RoomType, LocalDate from, LocalDate to);

}

D - Dependency Inversion Principle

High-level modules should not depend on low-level modules.

Use interfaces and dependency injection (DI) (e.g., via Spring Boot @Autowired or constructor injection).

class BookingService {

private final PaymentService paymentService;

private final RefundPolicy refundPolicy;

public BookingService(PaymentService ps, RefundPolicy rp) {

this.paymentService = ps;

this.refundPolicy = rp;

}

}

2. Design Patterns You Can Use

🧱 1. Factory Pattern

Use to create different types of Room, PaymentMethod, ServiceRequest.

RoomFactory.createRoom(RoomType.DELUXE);

🧾 5. Decorator Pattern

Add extra charges (e.g., for amenities or food) to a Booking dynamically.

interface Billable {

double calculateTotal();

}

class BaseRoomBooking implements Billable { ... }

class ExtraBedDecorator extends BookingDecorator { ... }

class FoodServiceDecorator extends BookingDecorator { ... }

Dynamically add optional charges to a booking without modifying the core booking logic.

Step 1: Define a common interface

public interface Billable {

double calculateTotal();

String getDescription();

}

Step 2: Base class (core booking)

public class BaseRoomBooking implements Billable {

private double basePrice;

public BaseRoomBooking(double basePrice) {

this.basePrice = basePrice;

}

@Override

public double calculateTotal() {

return basePrice;

}

@Override

public String getDescription() {

return "Room Booking";

}

}

🔹 Step 3: Define the BookingDecorator abstract class

public abstract class BookingDecorator implements Billable {

protected Billable booking;

public BookingDecorator(Billable booking) {

this.booking = booking;

}

@Override

public double calculateTotal() {

return booking.calculateTotal();

}

@Override

public String getDescription() {

return booking.getDescription();

}

}

This class delegates to the wrapped Billable object and is meant to be extended by specific decorators.

Step 4: Add concrete decorators

🛏 Extra Bed

public class ExtraBedDecorator extends BookingDecorator {

public ExtraBedDecorator(Billable booking) {

super(booking);

}

@Override

public double calculateTotal() {

return super.calculateTotal() + 500; // Extra bed charge

}

@Override

public String getDescription() {

return super.getDescription() + ", Extra Bed";

}

}

Food Service

public class FoodServiceDecorator extends BookingDecorator {

public FoodServiceDecorator(Billable booking) {

super(booking);

}

@Override

public double calculateTotal() {

return super.calculateTotal() + 800; // Food service charge

}

@Override

public String getDescription() {

return super.getDescription() + ", Food Service";

}

}