Software Requirement Specification Document for Hotel Management Software

Introduction

Project Link:- https://github.com/mrtechtroid/Programming_2_Project/

Team Details

Submitted by: Operator Overloaded

Name	Roll Number	Email ID
Gourav Anirudh B J	IMT2023005	GouravAnirudh.BJ@iiitb.ac.in
Sahil Ajaykumar Kolte	IMT2023066	sahilajaykumar.kolte@iiitb.ac.in
Kunal Jindal	IMT2023049	kunal.jindal@iiitb.ac.in
Pramatha V Rao	IMT2023116	pramatha.rao@iiitb.ac.in
Navaneeth D	IMT2023095	Navaneeth.d@iiitb.ac.in
Yashub Goel	IMT2023117	Yashub.goel@iiitb.ac.in
Agam Roy	IMT2023131	Agam.Roy@iiitb.ac.in

Project Overview

The Hotel Management System (HMS) is a comprehensive software solution designed to streamline hotel operations through one primary interface: a hotel admin interface. It delivers an efficient, user-friendly experience for both guests and hotel staff.[*Optional feature: Customer side interface to book directly*]

The authenticated hotel admin interface helps staff manage essential operations such as room booking, billing, housekeeping, and rate management. By integrating Java and C++ using Java Native Interface (JNI), the system combines the performance advantages of C++ with the flexibility of Java for efficient backend processing.

Scope

- Each hotel will host its own independent instance of the system, admin interfaces, regardless of whether they belong to the same chain of hotels or not.
- The system is flexible and can be used by hotels of any size and configuration. However, each hotel is limited to managing at most one restaurant, regardless of the number of rooms and the system will manage table reservations, billing and menu items for that restaurant.
- We also assume that each stay begins at 12pm on a day and lasts till 11am on the next day.

Objectives

The primary objective of the Hotel Management System is to develop a comprehensive platform that caters to the specific needs of different roles within the hotel and restaurant environment. The system aims to achieve the following:

1. **User Based interfaces**
    a. Provide a seamless interface for hotel receptionists to manage room bookings, check-ins, check-outs, and room status updates.
    b. Enable restaurant managers to efficiently handle table management, placing orders, and billing processes.
2. **Implement Role-Based Access Control (RBAC) [Optional: Only when Customer Interface gets implemented]**
    a. Ensure that each role (customer, hotel receptionist, restaurant manager, hotel manager) can only access and manage the features relevant to their responsibilities, safeguarding the system from unauthorized access.
3. **Centralized Control for Hotel Manager**
    a. Provide the hotel manager with full access to oversee both hotel and restaurant operations, generate reports, and make informed decisions based on real-time data from all departments.
4. **Efficient Room Management:**
    a. Implement features to allow administrators to manage room availability, types, and statuses (occupied, vacant, under maintenance) in real-time.
5. **Guest Check-in/Check-out Management:**
    a. Streamline the process of guest check-ins and check-outs, ensuring accurate tracking of room occupancy and guest stay duration.
6. **Staff Management:**
    a. Provide a system for managing staff assignments, including housekeeping, maintenance, and front-desk duties, to ensure smooth hotel operations.
7. **Reports and Analytics:**
    a. Provide a reporting system for generating analytics on room occupancy, revenue, guest demographics, and other key metrics to help in business decision-making.
8. **Customer Relationship Management (CRM):**
    a. Maintain a record of guest information and history to personalize future stays and improve customer service.
9. **Restaurant Billing System:**

   a. Provide a system to keep track of all dishes offered by the restaurant, manage table reservations, and billing of dishes ordered by consumers.

10. **[Optional] Feedback System:**
    a. Provide a feedback and customer review system for room bookings, restaurant dine-ins, etc.

These objectives focus on enhancing productivity, security, and user satisfaction within the hotel and restaurant environment.

System Overview

Technical Specifications

Frontend:

[Implemented Partially] Command Line Interface (CLI):

- **Technology Used**: Java, C++ (via JNI)
- **Purpose**:
  - Provides a text-based interface where users or administrators can log in and perform role specific functions.
- **Functionality**:
  - Determines whether the login ID belongs to a customer or an admin by performing a password verification.
  - Admins can manage room bookings, check-ins/check-outs, billing, housekeeping, rate management, and customer feedback.
  - All operations are performed via simple commands, ensuring efficient task execution.

Website:

- **Technology Used:** React, TypeScript
- **Purpose**:
  - Hotel admin interface for managing room bookings, check-ins/check-outs, and housekeeping.
  - Interactive UI to display real-time information, booking history, and availability.
  - Responsive design for cross-device accessibility (desktop, mobile, tablet).

Middleware:

- **Technology Used:** Java Spring Boot
- **Purpose:**
  - Acts as a bridge between frontend (React/Command Line) and backend (Java & C++ components).
  - Handles business logic, role-based access control.
  - Orchestrates calls to C++ modules through the Java Native Interface (JNI).

Backend:

- **Technology Used:** C++ (via JNI), Java
- **Purpose:**
  - To manage primary backend operations (booking, check-ins, reports, etc.) and interaction with databases.
  - Information related to customers and the hotel will be stored in files and will be processed using File I/O using C++.
  - [**Optional**] Database management for storing booking, room, customer, and financial data.

Input/Output Requirements

Input Requirements:

1. **Customer Data:**
   a. Personal details such as name, email, contact number, and address during registration and booking.
2. **Room Reservation Data:**
   a. Booking requests including room type, check-in date, check-out date, and any special requests (e.g., extra services).
   b. Updates from hotel staff regarding room availability, check-in/check-out status.
3. **Restaurant Data:**
   a. Customer orders from the restaurant, including selected menu items, quantities, and delivery requests (if applicable).
4. **Hotel Admin Data:**
   a. Room management inputs such as room rates, room availability, and housekeeping schedules.

   b. Billing data including additional charges (e.g., restaurant orders or extra services).

5. **Authentication Data:**
   a. Username and password for login for secure access control.

Output Requirements:

1. **Room Availability Status:**
   a. Real-time display of room availability and booking status for both customers and admins.
2. **Restaurant Order Status:**
   a. Updates on the status of customer restaurant orders, including preparation and delivery status.
   b. Invoice generation for restaurant orders, linked to the customer's room or home delivery.
3. **Billing Information:**
   a. Final bill generation for customers at checkout, detailing room charges, restaurant orders, and any extra services used during their stay.
   b. Detailed billing reports for hotel admins, including room revenues and restaurant sales. [Optional]

Functional Requirements

Features

- **Authentication using Login/Logout:** Ensures that each person can only access and manage the features only if they authenticate safeguarding the system from unauthorized access.
- **Efficient Room Management:** Implement features that enable administrators to manage room availability, types, and statuses (occupied, vacant, under maintenance) . Administrators can also modify customer bookings, including adjusting the duration of the stay based on customer requests, ensuring flexibility and seamless room management.
- **Staff Management:** Provide a system for managing staff assignments, including housekeeping, maintenance, and front-desk duties, to ensure smooth hotel operations.
- **Customer Relationship Management (CRM):** Maintain a record of guest information and history to personalize future stays and improve customer service.

- **Room Billing System:** Features to calculate the total cost of the stay of customers. Customers will also be able to view the cost of their booking while booking on the hotel website.
- **Restaurant Billing System:** Provide a system to keep track of all dishes offered by the restaurant, manage table reservations, and billing of dishes ordered by consumers.
- **[Optional] Reports and Analytics:** Provide a reporting system for generating analytics on room occupancy, revenue, guest demographics, and other key metrics to help in business decision-making.
- **[Optional] Feedback System:** Provide a feedback and customer review system for room bookings, restaurant dine-ins, etc.

Use Cases

Admin:

- Manage room inventory.
- Generate bills
- Add and remove Dishes
- Add and remove tables
- Reserve Rooms

Usability

1. **Intuitive User Interface**: The system should provide a clean, user-friendly interface for hotel admins, allowing them to perform operations without specialized training.
2. **Mobile Responsiveness**: The customer-facing website (React app) should be fully responsive and usable on mobile devices.
3. **Error Handling**: Provide user-friendly error messages in case of invalid inputs, system errors, or failures.

Security

1. **Authentication**: Users must authenticate before accessing any admin functions (via JWT tokens in our case for user sessions).

Maintainability

- **Code Modularity:** Separate layers for frontend, middleware, and backend operations.
- **Error Handling:** User-friendly error messages and exception handling

Class Diagrams and OOPS principles

1) Abstraction

- The "Customer" class abstracts common functionalities and attributes shared by "RestaurantCustomer" and "HotelCustomer."
- Utility methods like isValidDate() and isValidTime() in the DateTime class abstract complex date/time validations.
- Constructors like Room() and RoomType() abstract the initialization process for Room and RoomType classes.

2) Encapsulation

- Private attributes in classes like Bill, DateTime, Staff, and Room ensure data security and follow the principle of data hiding.
- Public getter and setter methods in classes like Role, Room, and RoomType provide controlled access to attributes.
- The Staff class demonstrates encapsulation by wrapping attributes (e.g., name, phone) and methods within a single unit.

3) Inheritance

- The "Customer" class serves as an abstract base class for "RestaurantCustomer" and "HotelCustomer."
- The modular design of the Reservation, Hotel, and Bill classes suggests potential for hierarchical structures.
- Inferred inheritance in the "Restaurant," "Table," and "Dish" design could extend behavior or attributes.

4) Polymorphism

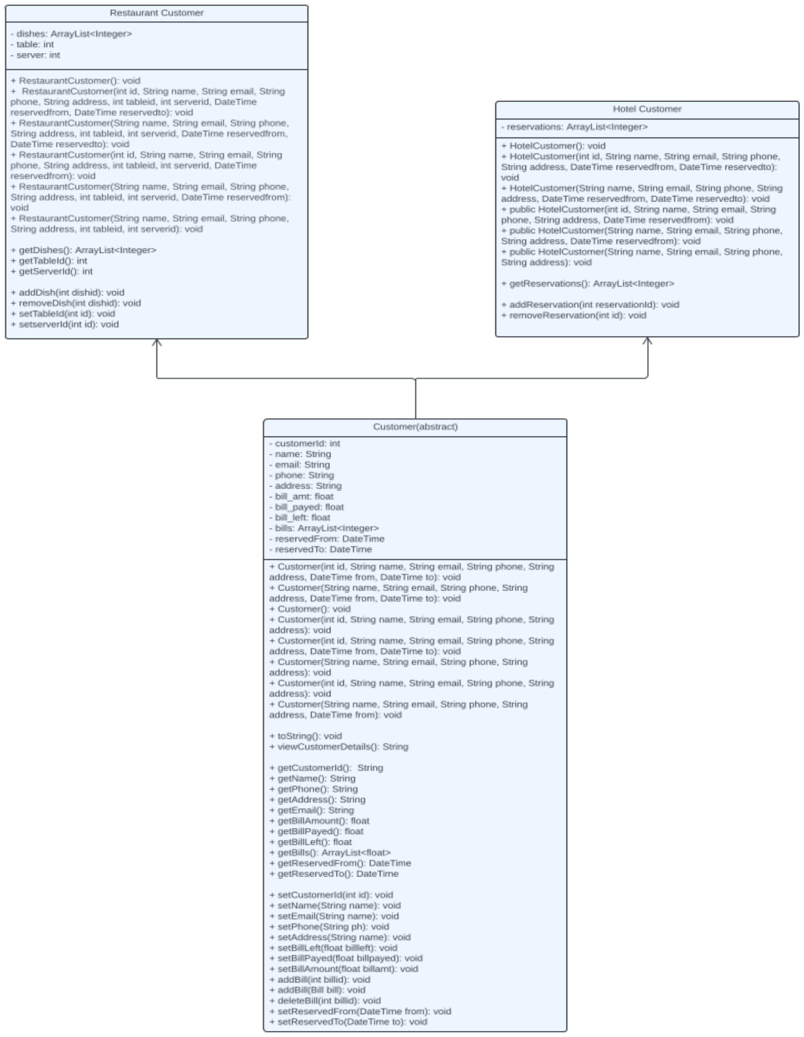
- The use of polymorphism is implied in "Restaurant," "Table," and "Dish" through potential method overloading or overriding.
- Polymorphism allows specialized behavior in subclasses like RestaurantCustomer and HotelCustomer through shared methods of the Customer base class.

5) Modularity

- Separation of responsibilities is evident in classes like Reservation, Hotel, and Bill, focusing on specific entities in the hotel management system.
- Room and RoomType classes divide room-specific and type-specific details, promoting single responsibility.
- The "Customer" design modularly separates restaurant and hotel contexts while maintaining a unified structure.
- The association between BillStore and Bill reflects a clear module for handling billing within a store.

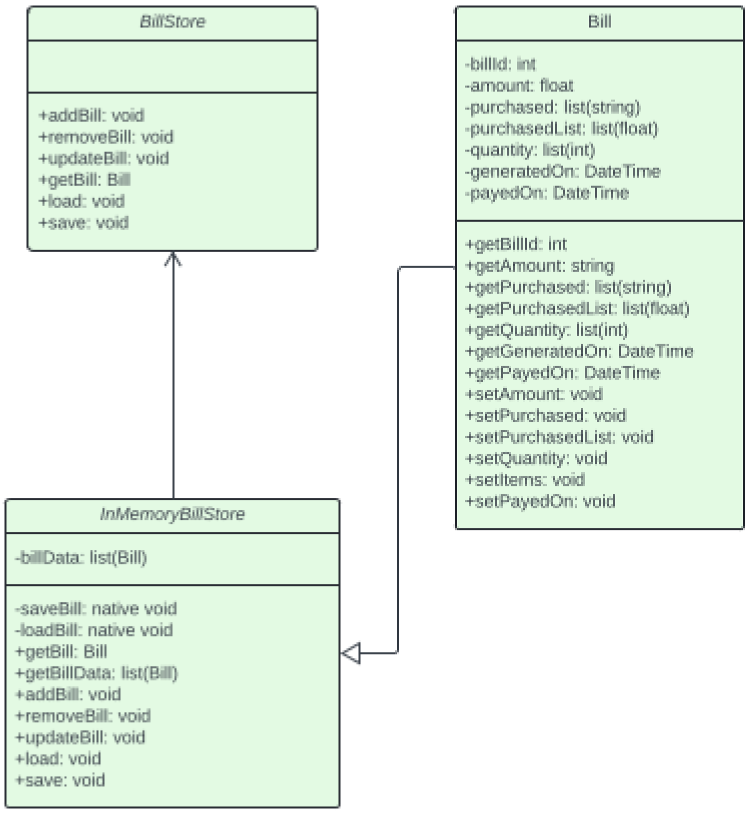
Class Diagrams

1.


Restaurant Customer
- dishes: ArrayList<Integer>
- table: int
- server: int
+ RestaurantCustomer(): void
+ RestaurantCustomer(int id, String name, String email, String phone, String address, int tableid, int serverid, DateTime reservedfrom, DateTime reservedto): void
+ RestaurantCustomer(String name, String email, String phone, String address, int tableid, int serverid, DateTime reservedfrom, DateTime reservedto): void
+ RestaurantCustomer(int id, String name, String email, String phone, String address, int tableid, int serverid, DateTime reservedfrom): void
+ RestaurantCustomer(String name, String email, String phone, String address, int tableid, int serverid, DateTime reservedfrom): void
+ RestaurantCustomer(String name, String email, String phone, String address, int tableid, int serverid): void
+ getDishes(): ArrayList<Integer>
+ getTableId(): int
+ getServerId(): int
+ addDish(int dishid): void
+ removeDish(int dishid): void
+ setTableId(int id): void
+ setserverId(int id): void
Hotel Customer
- reservations: ArrayList<Integer>
+ HotelCustomer(): void
+ HotelCustomer(int id, String name, String email, String phone, String address, DateTime reservedfrom, DateTime reservedto): void
+ HotelCustomer(String name, String email, String phone, String address, DateTime reservedfrom, DateTime reservedto): void
+ public HotelCustomer(int id, String name, String email, String phone, String address, DateTime reservedfrom): void
+ public HotelCustomer(String name, String email, String phone, String address, DateTime reservedfrom): void
+ public HotelCustomer(String name, String email, String phone, String address): void
+ getReservations(): ArrayList<Integer>
+ addReservation(int reservationId): void
+ removeReservation(int id): void
Customer(abstract)
- customerId: int
- name: String
- email: String
- phone: String
- address: String
- bill_amt: float
- bill_payed: float
- bill_left: float
- bills: ArrayList<Integer>
- reservedFrom: DateTime
- reservedTo: DateTime
+ Customer(int id, String name, String email, String phone, String address, DateTime from, DateTime to): void
+ Customer(String name, String email, String phone, String address, DateTime from, DateTime to): void
+ Customer(): void
+ Customer(int id, String name, String email, String phone, String address): void
+ Customer(int id, String name, String email, String phone, String address, DateTime from, DateTime to): void
+ Customer(String name, String email, String phone, String address): void
+ Customer(int id, String name, String email, String phone, String address): void
+ Customer(String name, String email, String phone, String address, DateTime from): void
+ toString(): void
+ viewCustomerDetails(): String
+ getCustomerId(): String
+ getName(): String
+ getPhone(): String
+ getAddress(): String
+ getEmail(): String
+ getBillAmount(): float
+ getBillPayed(): float
+ getBillLeft(): float
+ getBills(): ArrayList<float>
+ getReservedFrom(): DateTime
+ getReservedTo(): DateTime
+ setCustomerId(int id): void
+ setName(String name): void
+ setEmail(String name): void
+ setPhone(String ph): void
+ setAddress(String name): void
+ setBillLeft(float billleft): void
+ setBillPayed(float billpayed): void
+ setBillAmount(float billamt): void
+ addBill(int billid): void
+ addBill(Bill bill): void
+ deleteBill(int billid): void
+ setReservedFrom(DateTime from): void
+ setReservedTo(DateTime to): void

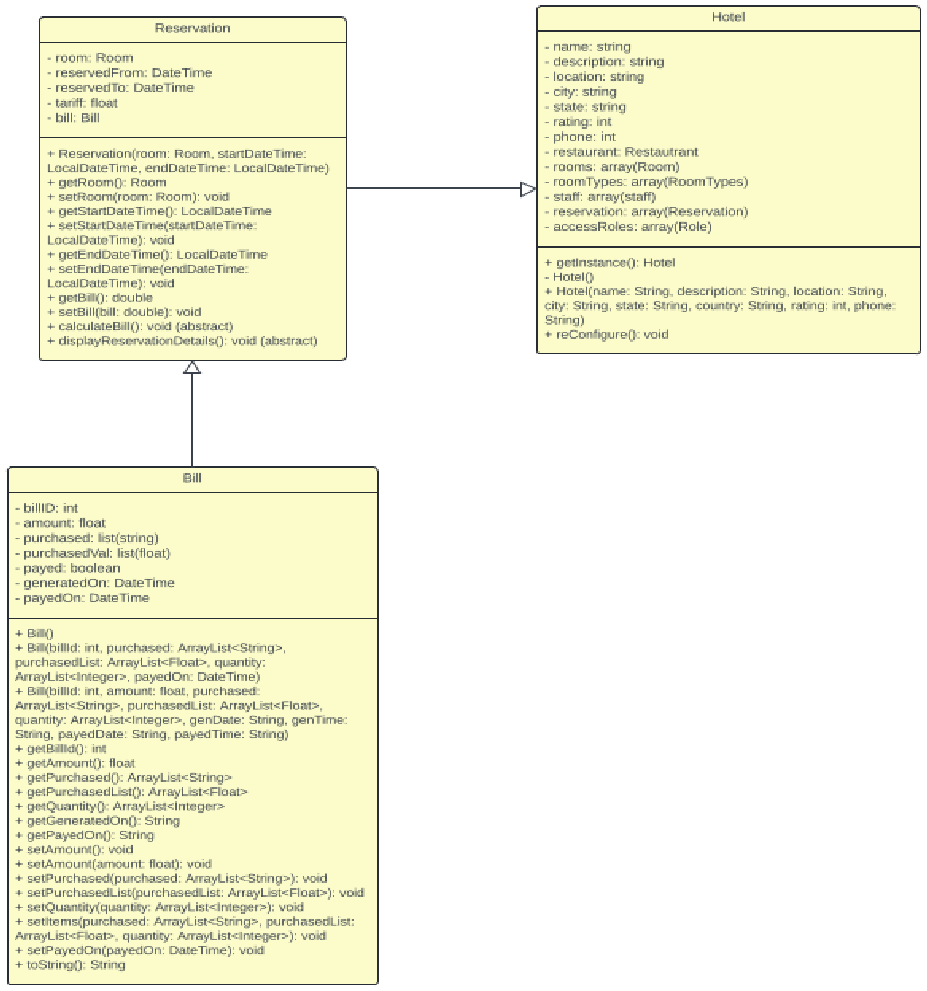
1. Main classes are "RestaurantCustomer" and "HotelCustomer" representing different customer types.
2. Both classes inherit from an abstract "Customer" class, indicating common attributes and methods.
3. "RestaurantCustomer" handles restaurant-specific customer data and functionality.
4. "HotelCustomer" manages hotel-related customer information, like reservations.
5. The class hierarchy demonstrates object-oriented principles like **abstraction** and **inheritance**.
6. The design suggests a structured system for managing customer data across restaurant and hotel contexts.

2.



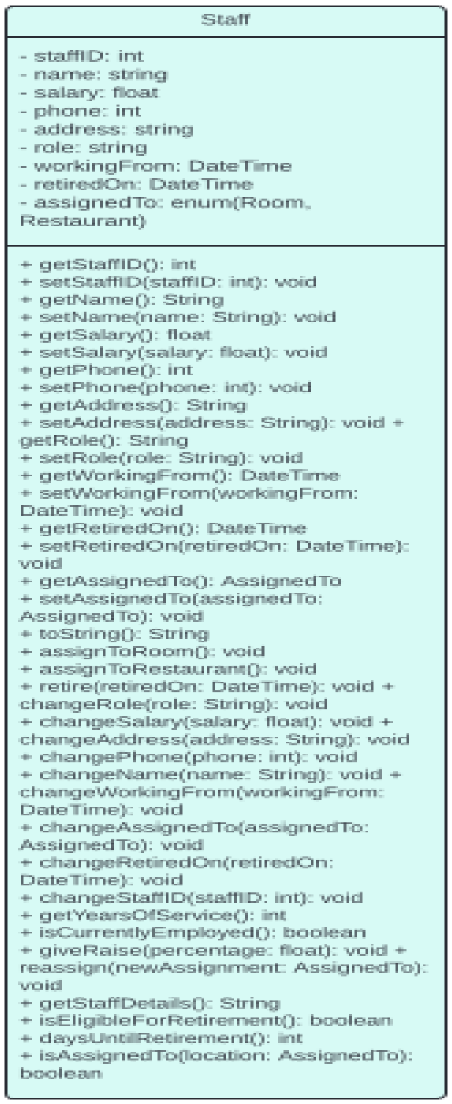
1. The main classes are "Bill" and "BillStore", representing the bill and the store/location where the bill is generated.
2. The "BillStore" class has attributes and methods related to managing the bill, such as addBill(), removeBill(), and updateBill().
3. The "Bill" class has attributes like amount, float, quantity, and methods like getBilled(), getPurchased(), and getQuantity() to access various bill-related information.
4. The association between the "BillStore" and "Bill" classes suggests that the bill is stored and managed within the BillStore.
5. The class diagram demonstrates **encapsulation**, as the classes have private attributes and provide public methods to access and modify the data.
6. Overall, the diagram represents a system for managing bills within a store or business context.

3.


Reservation
- room: Room
- reservedFrom: DateTime
- reservedTo: DateTime
- tariff: float
- bill: Bill
+ Reservation(room: Room, startDateTime: LocalDateTime, endDateTime: LocalDateTime)
+ getRoom(): Room
+ setRoom(room: Room): void
+ getStartDateTime(): LocalDateTime
+ setStartDateTime(startDateTime: LocalDateTime): void
+ getEndDateTime(): LocalDateTime
+ setEndDateTime(endDateTime: LocalDateTime): void
+ getBill(): double
+ setBill(bill: double): void
+ calculateBill(): void (abstract)
+ displayReservationDetails(): void (abstract)
Hotel
- name: string
- description: string
- location: string
- city: string
- state: string
- rating: int
- phone: int
- restaurant: Restautrant
- rooms: array(Room)
- roomTypes: array(RoomTypes)
- staff: array(staff)
- reservation: array(Reservation)
- accessRoles: array(Role)
+ getInstance(): Hotel
- Hotel()
+ Hotel(name: String, description: String, location: String, city: String, state: String, country: String, rating: int, phone: String)
+ reConfigure(): void
Bill
- billID: int
- amount: float
- purchased: list(string)
- purchasedVal: list(float)
- payed: boolean
- generatedOn: DateTime
- payedOn: DateTime
+ Bill()
+ Bill(billId: int, purchased: ArrayList<String>, purchasedList: ArrayList<Float>, quantity: ArrayList<Integer>, payedOn: DateTime)
+ Bill(billId: int, amount: float, purchased: ArrayList<String>, purchasedList: ArrayList<Float>, quantity: ArrayList<Integer>, genDate: String, genTime: String, payedDate: String, payedTime: String)
+ getBillId(): int
+ getAmount(): float
+ getPurchased(): ArrayList<String>
+ getPurchasedList(): ArrayList<Float>
+ getQuantity(): ArrayList<Integer>
+ getGeneratedOn(): String
+ getPayedOn(): String
+ setAmount(): void
+ setAmount(amount: float): void
+ setPurchased(purchased: ArrayList<String>): void
+ setPurchasedList(purchasedList: ArrayList<Float>): void
+ setQuantity(quantity: ArrayList<Integer>): void
+ setItems(purchased: ArrayList<String>, purchasedList: ArrayList<Float>, quantity: ArrayList<Integer>): void
+ setPayedOn(payedOn: DateTime): void
+ toString(): String

1. The main classes are Reservation, Hotel, and Bill, representing the core entities in a hotel management system.
2. The Reservation class handles details like the room, date/time, and associated bill information.
3. The Hotel class stores details about the hotel itself, such as the name, description, and facility types.
4. The Bill class contains bill-related data, including the amount, purchased items, and payment details.
5. The relationships between the classes suggest a structured design, where reservations are made at a hotel and are linked to a bill.
6. The overall diagram demonstrates principles like **encapsulation**, **composition**, and **abstraction** in the object-oriented modeling of the hotel management system.

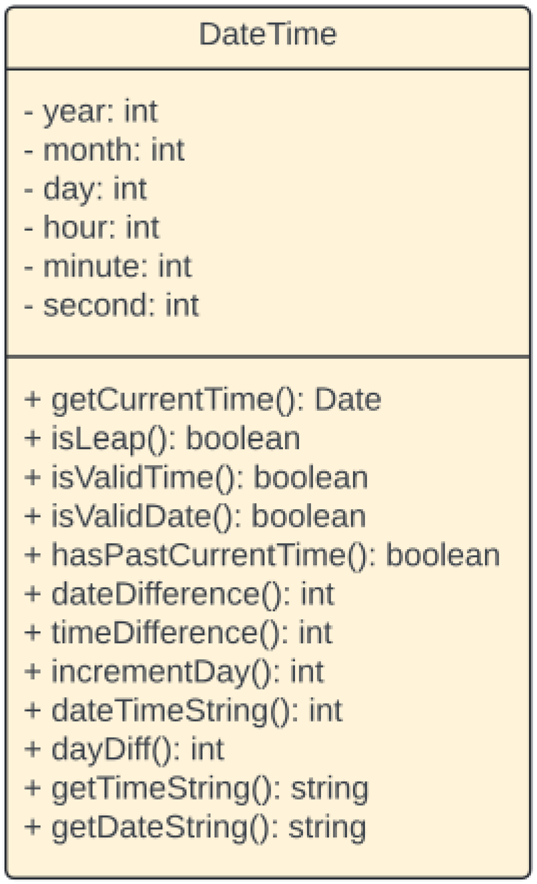
4.


Staff
- staffID: int
- name: string
- salary: float
- phone: int
- address: string
- role: string
- workingFrom: DateTime
- retiredOn: DateTime
- assignedTo: enum(Room, Restaurant)
+ getStaffID(): int
+ setStaffID(staffID: int): void
+ getName(): String
+ setName(name: String): void
+ getSalary(): float
+ setSalary(salary: float): void
+ getPhone(): int
+ setPhone(phone: int): void
+ getAddress(): String
+ setAddress(address: String): void + getRole(): String
+ setRole(role: String): void
+ getWorkingFrom(): DateTime
+ setWorkingFrom(workingFrom: DateTime): void
+ getRetiredOn(): DateTime
+ setRetiredOn(retiredOn: DateTime): void
+ getAssignedTo(): AssignedTo
+ setAssignedTo(assignedTo: AssignedTo): void
+ toString(): String
+ assignToRoom(): void
+ assignToRestaurant(): void
+ retire(retiredOn: DateTime): void + changeRole(role: String): void
+ changeSalary(salary: float): void + changeAddress(address: String): void
+ changePhone(phone: int): void
+ changeName(name: String): void + changeWorkingFrom(workingFrom: DateTime): void
+ changeAssignedTo(assignedTo: AssignedTo): void
+ changeRetiredOn(retiredOn: DateTime): void
+ changeStaffID(staffID: int): void
+ getYearsOfService(): int
+ isCurrentlyEmployed(): boolean
+ giveRaise(percentage: float): void + reassign(newAssignment: AssignedTo): void
+ getStaffDetails(): String
+ isEligibleForRetirement(): boolean
+ daysUntillRetirement(): int
+ isAssignedTo(location: AssignedTo): boolean

1. The main class represented is "Staff", which contains information about employees or personnel in the system.
2. The "Staff" class has attributes such as name, phone, address, and working hours. It also includes methods for managing staff-related data, like getting and setting salary, address, and assigned room.
3. The class has a rich set of attributes and methods, indicating that it is a central component in the system, responsible for storing and handling staff-related information.
4. The diagram demonstrates principles of **encapsulation**, as the attributes and methods of the "Staff" class are properly wrapped within the class structure.

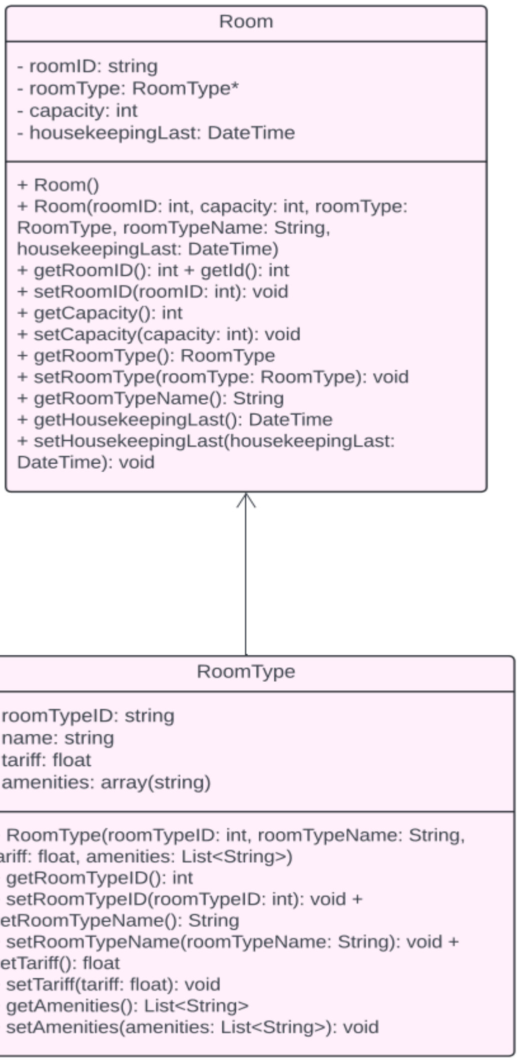
5. While the diagram only shows a single "Staff" class, in a more complex system, there may be additional classes or subclasses to handle different types of staff, roles, or organizational structures.

5.



1. The DateTime class encapsulates attributes and methods for handling date and time, demonstrating **encapsulation**.
2. Private attributes like year, month, and day ensure data security, following the principle of **data hiding**.
3. Methods like isValidDate() and isValidTime() validate input, ensuring proper use of the class.
4. The class provides utility functions like incrementDay() and dateDifference() for date manipulation.
5. Functions such as getTimeString() and getDateString() follow **abstraction** by simplifying complex date-time operations.
6. The inclusion of hasPastCurrentTime() and isLeap() allows for logical checks on date and time.

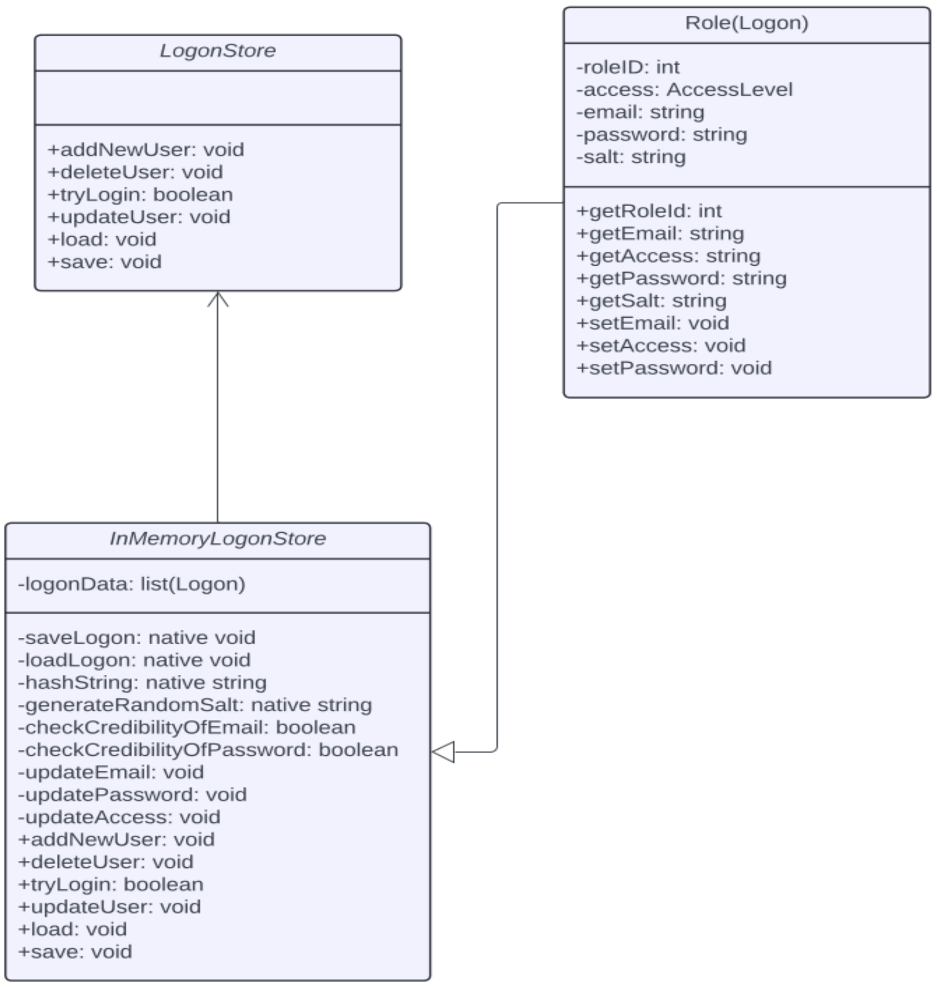
6.


Room
- roomID: string
- roomType: RoomType*
- capacity: int
- housekeepingLast: DateTime
+ Room()
+ Room(roomID: int, capacity: int, roomType: RoomType, roomTypeName: String, housekeepingLast: DateTime)
+ getRoomID(): int + getId(): int
+ setRoomID(roomID: int): void
+ getCapacity(): int
+ setCapacity(capacity: int): void
+ getRoomType(): RoomType
+ setRoomType(roomType: RoomType): void
+ getRoomTypeName(): String
+ getHousekeepingLast(): DateTime
+ setHousekeepingLast(housekeepingLast: DateTime): void
RoomType
- roomTypeID: string
- name: string
- tariff: float
- amenities: array(string)
+ RoomType(roomTypeID: int, roomTypeName: String, tariff: float, amenities: List<String>)
+ getRoomTypeID(): int
+ setRoomTypeID(roomTypeID: int): void + getRoomTypeName(): String
+ setRoomTypeName(roomTypeName: String): void + getTariff(): float
+ setTariff(tariff: float): void
+ getAmenities(): List<String>
+ setAmenities(amenities: List<String>): void

1. The UML diagram represents a **Room** class and a related **RoomType** class, showing an association between them.
2. **Encapsulation** is used as both classes have private attributes (roomID, roomTypeID, etc.) and public getter and setter methods to access and modify them.
3. The Room class has an aggregation relationship with RoomType (indicated by the RoomType* pointer), allowing rooms to reference different types of rooms.
4. Constructors like Room() and RoomType() demonstrate **abstraction** by initializing objects with predefined parameters.
5. The modular design separates room-specific details (Room) from type-specific details (RoomType), promoting **single responsibility** and reusability.

6. The setter and getter methods in both classes provide controlled access, ensuring **data hiding** and secure manipulation of attributes.

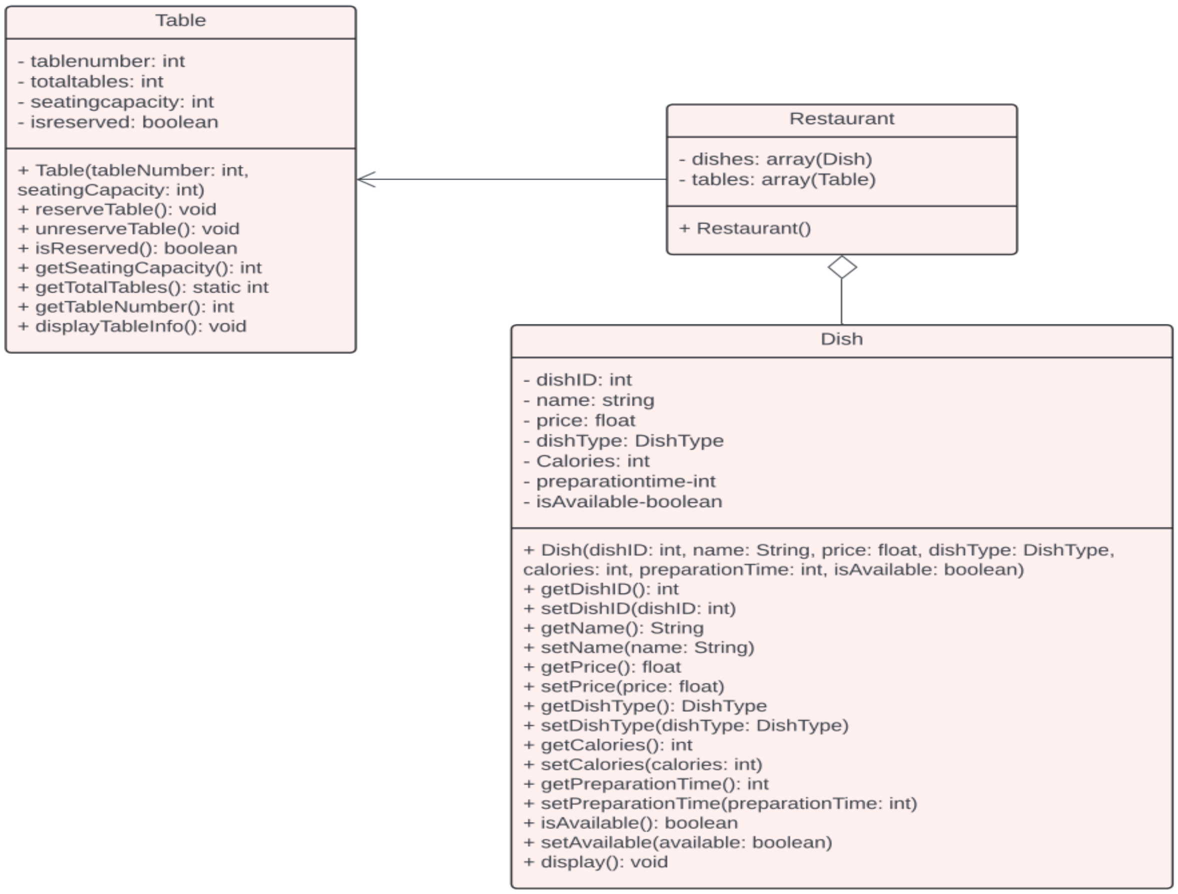
7.



1. The UML diagram represents a **LogonStore** class, a **Role** class, and a subclass **InMemoryLogonStore**.
2. **Inheritance** is applied as InMemoryLogonStore extends LogonStore, allowing it to implement or override the functionality.
3. The **Role** class encapsulates attributes like roleID, access, email, password, and salt and provides getter and setter methods to manage these attributes, adhering to **encapsulation** and **data hiding**.
4. The LogonStore class defines methods such as addNewUser(), deleteUser(), and tryLogin() for user management, demonstrating **abstraction** by focusing on essential functionality.

5. The InMemoryLogonStore class adds specific functionality like hashing strings, generating salts, and checking password or email credibility, following the **open/closed principle** by extending functionality without modifying the base class.
6. This structure provides a clear separation of responsibilities: Role handles user details, LogonStore provides a base for managing logon functionality, and InMemoryLogonStore implements storage-specific behavior.

8.



1. The diagram shows three classes: Table, Restaurant, and Dish.
2. Each class has attributes and methods to represent its characteristics and behavior.
3. A Restaurant has-a relationship with Dish and Table, meaning it contains multiple instances of these classes.
4. Inheritance is not explicitly shown, but could be inferred from the class names and attributes.
5. **Encapsulation** is used to hide implementation details within each class.
6. **Polymorphism** could be achieved through method overloading or overriding.

Development Setup

Workflow

The development workflow is structured into three primary layers: **Frontend**, **Middleware**, and **Backend**, ensuring a modular architecture that is easy to maintain, extend, and debug. Each layer communicates seamlessly with the others via REST APIs and JNI for optimal performance and flexibility.

Frontend(React)

- Handles user interactions and displays real-time data.
- Consumes APIs provided by the middleware for room bookings, check-ins, billing, and other operations.
- Designed to be fully responsive for usage on various devices, including desktops, tablets, and smartphones.

Middleware (Spring Boot)

- Acts as the bridge between the React frontend and the backend systems.
- Implements business logic, validates inputs, manages authentication/authorization, and orchestrates operations between Java and C++ modules using JNI.
- Provides endpoints for frontend interaction and communicates with backend services for file and data handling.

Backend(Java, C++ for File handling, JNI)

- **Java:** Implements controllers, service layers. Coordinates with C++ modules for file-based data processing. All the core logic of all operations is written in Java
- **C++:** Manages core operations such as file I/O for bookings, billing, and housekeeping, ensuring high performance for critical tasks.

Steps to run code:

Clone the repository and run the commands to build the project. The .so files will get automatically generated

```
# Clone the repository
git clone https://github.com/mrtechtroid/Programming_2_Project.git
## Navigate to the project directory
cd Programming_2_Project
#Run the commands
#To build the project
mvn clean install
#To build the project without the frontend
mvn clean install -Dskip.npm

mvn spring-boot:run -Dspring-boot.run arguments="in-memory gui" -Dskip.npm
```

Important Files and Folders

- Frontend- The code for the frontend is present in this directory
- Pom.xml- It contains the code for the build using Maven
- Java\com\operatoroverloaded\hotel - Contains all the codes written in java including the logic for all the operations
- Cpp- Contains the code for the file operations using the JNI

Testing and Logging

- Testing will cover room booking, unbooking, food ordering, and bill generation using mock data
- Database integrity will be verified through test cases that simulate typical hotel operations
- Logging mechanisms will track database query performance and handle unexpected errors, such as failed connections or invalid user input.

Conclusion

This Hotel Management software, backed by a database, is designed to streamline the hotel operations of booking rooms, managing customers, ordering food, and generating bills. With database persistence, the system can reliably store and retrieve customer data, ensuring the hotel’s state is maintained across program sessions. The use of a database enhances data retrieval speed, scalability, and overall efficiency in managing hotel activities.