Hotel Management System: Adherence to SOLID Principles

Introduction

The Hotel Management System is designed to manage hotel operations, including room booking, payment processing, and record maintenance. The code structure follows the SOLID design principles to ensure a flexible, maintainable, and scalable architecture.

1. Single Responsibility Principle (SRP)

Definition: A class should have only one reason to change, meaning it should only have one job or responsibility.

Implementation:

- Room Class: Focuses on room details and status, handling methods like book() and checkout().
- Customer Class: Manages customer-related information, such as name, contact details, and assigned room.
- Hotel Class: Acts as the main system coordinator, responsible for booking rooms, displaying records, and handling payments.
- Payment Classes: Each payment method class (BkashPayment, RocketPayment, etc.) handles its specific payment logic.

By adhering to SRP, each class has a clear, focused responsibility, simplifying maintenance and reducing code complexity.

2. Open/Closed Principle (OCP)

Definition: Classes should be open for extension but closed for modification.

Implementation:

- The PaymentMethod Interface allows the addition of new payment methods without modifying existing code. For example, adding a new payment class (NewPaymentService) only requires implementing the PaymentMethod interface.
- The Hotel Class can handle various room types and services through easily extensible methods, such as bookRoom() and showMenu(), without altering the existing logic.

This structure makes it easy to extend the system's functionality without risking existing features.

3. Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of a subclass without affecting the program's correctness.

Implementation:

- All PaymentMethod implementations (BkashPayment, RocketPayment, etc.) can be used interchangeably. The processPayment() method is called in the same way, regardless of the specific payment class used.
- This ensures that the Hotel Class can work seamlessly with any payment method, adhering to LSP.

The uniform behavior of payment classes guarantees that substitutions won't break the program.

4. Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on methods they do not use.

Implementation:

- The system uses specific interfaces: BookingService, RoomService, RecordService, and PaymentMethod. Each interface defines a distinct set of methods that are relevant to a particular service.
- For example, RoomService only declares showMenu(), ensuring that the implementing classes do not have to include unnecessary methods.

By breaking down interfaces, the system becomes more flexible, and classes only implement methods they require.

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions.

Implementation:

- The Hotel Class depends on the PaymentMethod interface, not on concrete payment classes. This abstraction allows the payment mechanism to be easily changed or extended.
- Dependency injection could be applied to further decouple classes and make testing more straightforward.

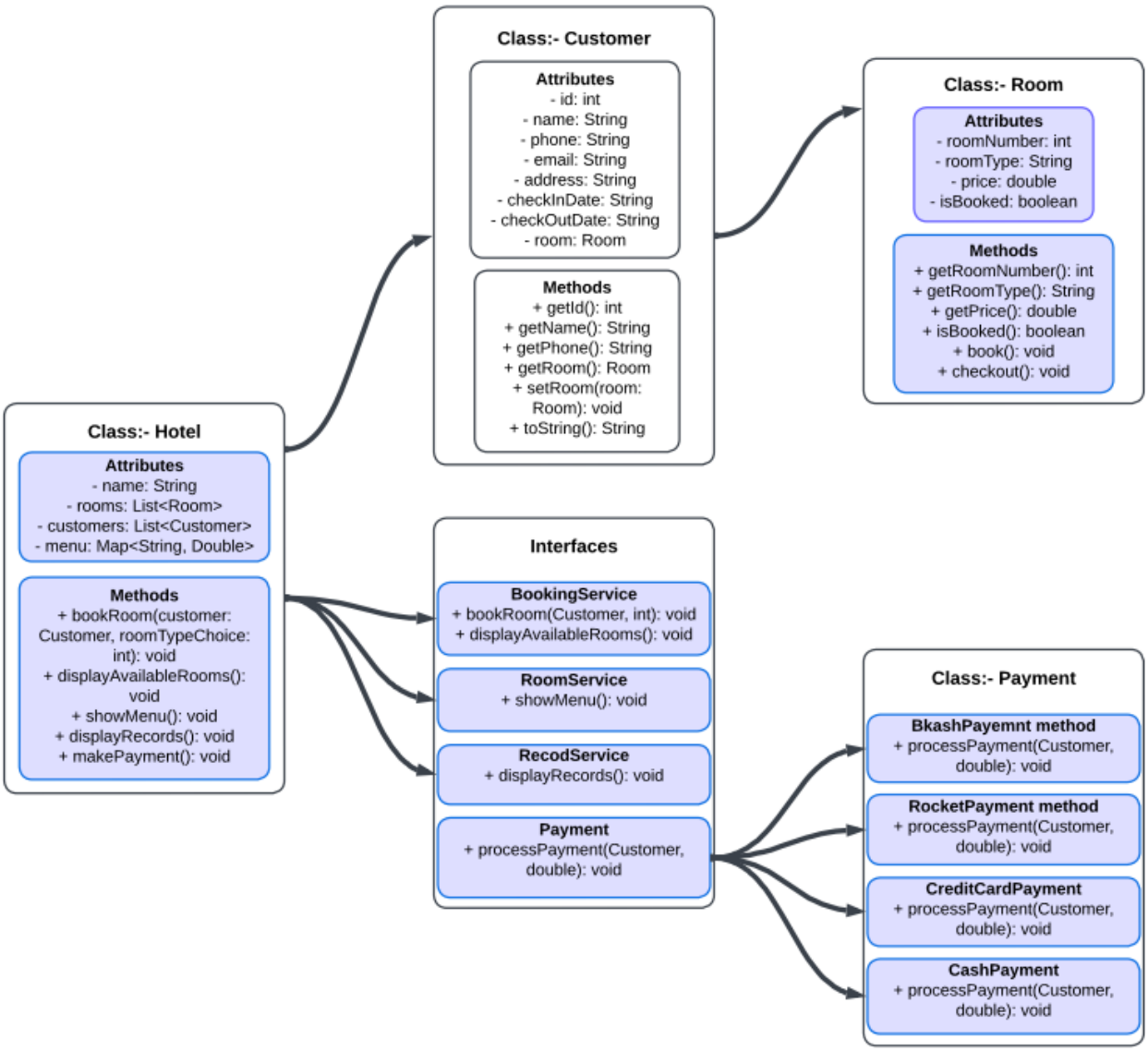
Using abstractions for payment processing promotes a loosely coupled design, enhancing testability and maintainability.

Conclusion

The Hotel Management System is designed with a solid architecture that adheres to SOLID principles. This design choice enhances code clarity, flexibility, and extensibility, making it suitable for future expansion and easier maintenance.

Thank You!

UML Diagram


Class:- Customer
Attributes
- id: int
- name: String
- phone: String
- email: String
- address: String
- checkInDate: String
- checkOutDate: String
- room: Room
Methods
+ getId(): int
+ getName(): String
+ getPhone(): String
+ getRoom(): Room
+ setRoom(room: Room): void
+ toString(): String
Class:- Room
Attributes
- roomNumber: int
- roomType: String
- price: double
- isBooked: boolean
Methods
+ getRoomNumber(): int
+ getRoomType(): String
+ getPrice(): double
+ isBooked(): boolean
+ book(): void
+ checkout(): void
Class:- Hotel
Attributes
- name: String
- rooms: List<Room>
- customers: List<Customer>
- menu: Map<String, Double>
Methods
+ bookRoom(customer: Customer, roomTypeChoice: int): void
+ displayAvailableRooms(): void
+ showMenu(): void
+ displayRecords(): void
+ makePayment(): void
Interfaces
BookingService
+ bookRoom(Customer, int): void
+ displayAvailableRooms(): void
RoomService
+ showMenu(): void
RecodService
+ displayRecords(): void
Payment
+ processPayment(Customer, double): void
Class:- Payment
BkashPayemnt method
+ processPayment(Customer, double): void
RocketPayment method
+ processPayment(Customer, double): void
CreditCardPayment
+ processPayment(Customer, double): void
CashPayment
+ processPayment(Customer, double): void