**Royal Inn Suite: Advanced Hotel Booking & Guest Services**

**Task 1: UML Class Diagram & Explanation**

**Overview**

The Royal Inn Suite Management System is developed to provide a streamlined approach to hotel operations, ensuring effortless room reservations, automated guest handling, and secure financial transactions. The UML diagram plays a crucial role in visually representing the architectural design of the system, outlining the relationships between different classes. The system follows OOP principles like inheritance, composition, and aggregation to achieve a modular and scalable structure.

**Primary Components and Their Roles**

**1. Person (Base Class) – The Foundation of User Management**

**Key Details:** This class serves as the **foundation for all user types** in the system, providing essential personal details.

**Attributes:** id, name, email, phone

**Functionalities:**

* view\_details(): Retrieves and displays personal information.
* update\_contact\_info(): Allows modification of stored contact details.

**2. Guest (Inherits from Person) – Managing Guest Functions**

**Key Details:** The **Guest class** extends the Person class, equipping guests with the ability to manage bookings and loyalty rewards.

**Attributes:** loyalty\_points, account\_status

**Functionalities:**

* book\_room(): Enables guests to search for and reserve a room.
* cancel\_reservation(): Provides the option to cancel a booking if needed.
* view\_rewards(): Displays loyalty points and available discounts.

**3. Employee (Inherits from Person) – Handling Hotel Operations**

**Key Details:** The **Employee class** represents staff members who oversee hotel activities.

**Attributes:** employee\_id, role

**Functionalities:**

* manage\_reservations(): Ensures smooth handling of bookings and cancellations.
* assist\_guests(): Helps guests with their queries and requests.

**4. Room – Tracking Room Availability and Pricing**

**Key Details:** This class maintains room-related data, helping guests and employees track **availability, pricing, and facilities**.

**Attributes:** room\_id, type, status, price, facilities

**Functionalities:**

* change\_status(): Updates room availability when a reservation is made or canceled.
* get\_details(): Provides a summary of room specifications.

**5. Reservation – Managing the Booking Process**

**Key Details:** This class ensures that room reservations are **properly stored, updated, and notified** to the users.

**Attributes:** reservation\_id, guest\_id, room\_id, dates, status

**Functionalities:**

* confirm\_reservation(): Finalizes a booking and marks the room as occupied.
* cancel\_reservation(): Frees up a reserved room and informs the concerned guest.
* send\_notification(): Notifies guests about reservation status via email or in-app alerts.

**6. Payment – Processing Transactions Securely**

**Key Details:** Handles all **billing operations**, ensuring transactions are recorded properly.

**Attributes:** transaction\_id, amount, method

**Functionalities:**

* process\_transaction(): Accepts payments via different modes (credit card, mobile wallet, etc.).
* issue\_receipt(): Generates invoices for completed payments.

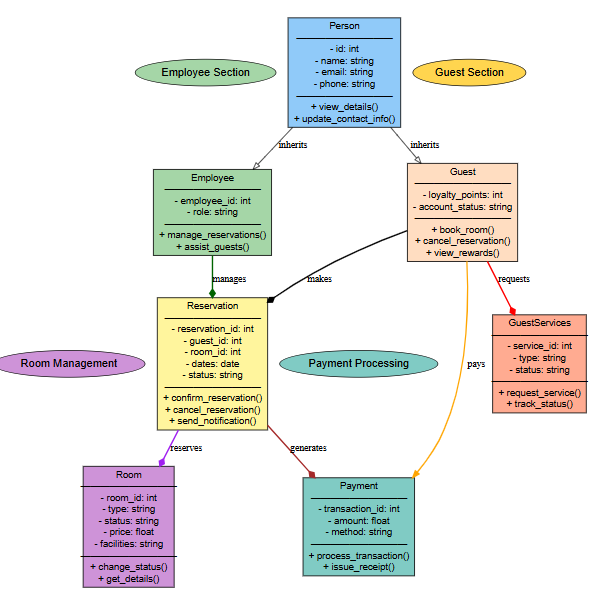
**7. GuestServices – Additional Service Requests**

**Key Details:** This component **enhances the guest experience** by managing extra services like room service, housekeeping, and transport.

**Attributes:** service\_id, type, status

**Functionalities:**

* request\_service(): Enables guests to request special services.
* track\_status(): Monitors the progress of requested services.



UML Diagram Royal Inn Suite Management System

### ****Task 2: Python Code for Project****

#### ****1. guest.py**** (Handles guest account operations)

class Guest:

    def \_\_init\_\_(self, guest\_id, name, email, phone, loyalty\_points=0):

        self.guest\_id = guest\_id

        self.name = name

        self.email = email

        self.phone = phone

        self.loyalty\_points = loyalty\_points

    def view\_details(self):

        return f"Guest ID: {self.guest\_id}, Name: {self.name}, Email: {self.email}, Phone: {self.phone}, Loyalty Points: {self.loyalty\_points}"

    def update\_contact\_info(self, new\_email, new\_phone):

        self.email = new\_email

        self.phone = new\_phone

        return "Contact info updated successfully."

#### ****2. employee.py**** (Manages hotel staff)

class Employee:

    def \_\_init\_\_(self, employee\_id, name, role):

        self.employee\_id = employee\_id

        self.name = name

        self.role = role

    def manage\_reservations(self):

        return f"Employee {self.name} ({self.role}) is managing reservations."

    def assist\_guests(self):

        return f"Employee {self.name} is assisting guests."

#### ****3. room.py**** (Handles room availability)

class Room:

    def \_\_init\_\_(self, room\_id, room\_type, price, status="Available"):

        self.room\_id = room\_id

        self.room\_type = room\_type

        self.price = price

        self.status = status

    def change\_status(self, new\_status):

        self.status = new\_status

        return f"Room {self.room\_id} status changed to {self.status}."

    def get\_details(self):

        return f"Room {self.room\_id} ({self.room\_type}): {self.price} USD per night - Status: {self.status}"

#### ****4. reservation.py**** (Manages room bookings)

class Reservation:

    def \_\_init\_\_(self, reservation\_id, guest, room, check\_in, check\_out):

        self.reservation\_id = reservation\_id

        self.guest = guest

        self.room = room

        self.check\_in = check\_in

        self.check\_out = check\_out

        self.status = "Pending"

    def confirm\_reservation(self):

        self.status = "Confirmed"

        self.room.change\_status("Booked")

        return f"Reservation {self.reservation\_id} confirmed for {self.guest.name}."

    def cancel\_reservation(self):

        self.status = "Cancelled"

        self.room.change\_status("Available")

        return f"Reservation {self.reservation\_id} cancelled for {self.guest.name}."

#### ****5. payment.py**** (Handles payments and invoices)

class Payment:

    def \_\_init\_\_(self, transaction\_id, amount, method):

        self.transaction\_id = transaction\_id

        self.amount = amount

        self.method = method

    def process\_transaction(self):

        return f"Transaction {self.transaction\_id}: {self.amount} USD processed via {self.method}."

    def issue\_receipt(self):

        return f"Receipt: Transaction {self.transaction\_id}, Amount: {self.amount} USD, Method: {self.method}"

#### ****6. guest\_services.py**** (Handles extra hotel services)

class GuestServices:

    def \_\_init\_\_(self, service\_id, service\_type):

        self.service\_id = service\_id

        self.service\_type = service\_type

        self.status = "Pending"

    def request\_service(self):

        self.status = "In Progress"

        return f"Service {self.service\_type} requested, status: {self.status}."

    def track\_status(self):

        return f"Service {self.service\_type} is currently {self.status}."

#### ****7. test\_hotel.py**** (Unit tests for all classes)

import unittest

from guest import Guest

from employee import Employee

from room import Room

from reservation import Reservation

from payment import Payment

from guest\_services import GuestServices

class TestHotelManagement(unittest.TestCase):

    def test\_guest\_creation(self):

        print("\n[TEST] Guest Creation")

        guest = Guest(1, "Alice Brown", "alice@example.com", "9876543210")

        expected\_output = f"Guest ID: 1, Name: Alice Brown, Email: alice@example.com, Phone: 9876543210, Loyalty Points: 0"

        actual\_output = guest.view\_details()

        print(f"Expected: {expected\_output}\nActual: {actual\_output}")

        self.assertEqual(actual\_output, expected\_output)

    def test\_employee\_assistance(self):

        print("\n[TEST] Employee Assistance")

        employee = Employee(101, "John Doe", "Manager")

        expected\_output = "Employee John Doe is assisting guests."

        actual\_output = employee.assist\_guests()

        print(f"Expected: {expected\_output}\nActual: {actual\_output}")

        self.assertEqual(actual\_output, expected\_output)

    def test\_room\_status\_change(self):

        print("\n[TEST] Room Status Change")

        room = Room(205, "Deluxe", 150)

        expected\_output = "Room 205 status changed to Booked."

        actual\_output = room.change\_status("Booked")

        print(f"Expected: {expected\_output}\nActual: {actual\_output}")

        self.assertEqual(actual\_output, expected\_output)

    def test\_make\_reservation(self):

        print("\n[TEST] Room Reservation")

        guest = Guest(2, "Bob Smith", "bob@example.com", "1234567890")

        room = Room(101, "Suite", 200)

        reservation = Reservation(301, guest, room, "2025-04-01", "2025-04-05")

        expected\_output = "Reservation 301 confirmed for Bob Smith."

        actual\_output = reservation.confirm\_reservation()

        print(f"Expected: {expected\_output}\nActual: {actual\_output}")

        self.assertEqual(actual\_output, expected\_output)

    def test\_payment\_processing(self):

        print("\n[TEST] Payment Processing")

        payment = Payment(501, 400, "Credit Card")

        expected\_output = "Transaction 501: 400 USD processed via Credit Card."

        actual\_output = payment.process\_transaction()

        print(f"Expected: {expected\_output}\nActual: {actual\_output}")

        self.assertEqual(actual\_output, expected\_output)

    def test\_service\_request(self):

        print("\n[TEST] Guest Service Request")

        service = GuestServices(701, "Room Cleaning")

        expected\_output = "Service Room Cleaning requested, status: In Progress."

        actual\_output = service.request\_service()

        print(f"Expected: {expected\_output}\nActual: {actual\_output}")

        self.assertEqual(actual\_output, expected\_output)

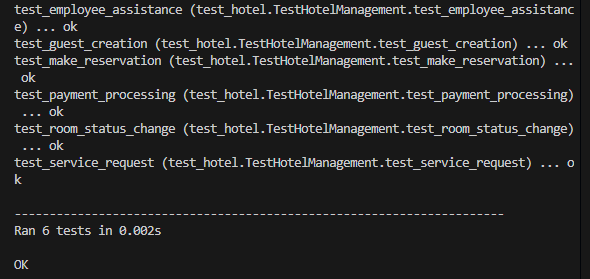
if \_\_name\_\_ == "\_\_main\_\_":

    unittest.main()

### ****Task 3: Run All Tests****

**For all Tests**

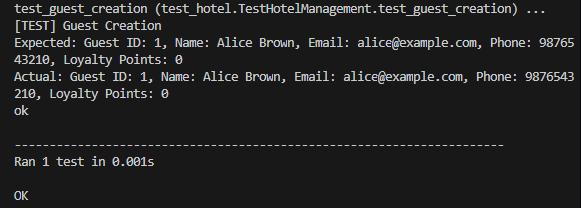
python -m unittest -v test\_hotel.py



Running all tests ensures that every component of the Royal Inn Suite Management System functions correctly as a whole. This command executes all unit tests defined in the test\_hotel.py file, verifying different functionalities such as guest creation, room management, reservations, payments, and service requests. By executing all tests together, we can identify any broken dependencies or conflicting functionalities within the system. If all tests pass successfully, it confirms that the system is stable and working as expected. If any test fails, the output provides detailed error messages to help diagnose and resolve issues. Running comprehensive tests helps catch bugs early, reducing the risk of system failures. This approach ensures that every module functions seamlessly with others. Testing all components together also helps verify data consistency across different classes.

### ****Test Guest Class****

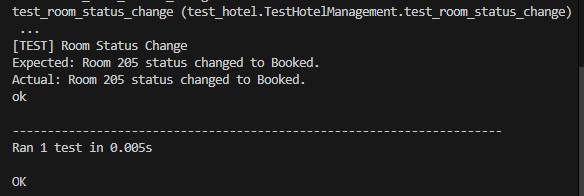
python -m unittest test\_hotel.TestHotelManagement.test\_guest\_creation –v



The Guest class handles guest registration, personal details, and loyalty points within the system. This test ensures that guest instances are correctly created and their attributes, such as name, email, and loyalty points, are stored accurately. By verifying the view\_details() method, the test checks if guest information is correctly retrieved. A successful test confirms that guests can register without issues and that their data is properly maintained. This prevents errors such as missing or incorrectly stored guest details. It ensures a smooth experience for customers using the system. If the test fails, it indicates potential issues in guest account creation or data retrieval. Running this test guarantees that the system maintains guest records effectively.

### ****Test Room Class****

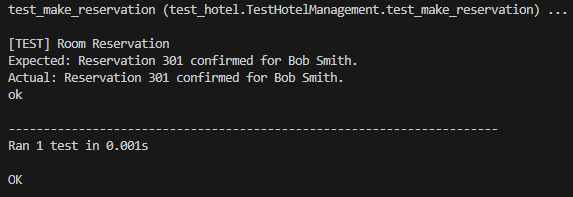
python -m unittest test\_hotel.TestHotelManagement.test\_room\_status\_change –v



The Room class manages hotel room information, including availability, type, and pricing. This test ensures that room status updates correctly when a guest books or cancels a reservation. The change\_status() method is tested to confirm that a room marked as "Available" correctly updates to "Booked" when reserved. The test also verifies that canceled bookings return the room to "Available" status. A successful outcome confirms that the system accurately tracks room availability. This prevents issues like double bookings or incorrect room statuses. If the test fails, it may indicate a problem with room data management. Proper room status updates ensure a reliable booking experience for guests.

### ****Test Reservation****

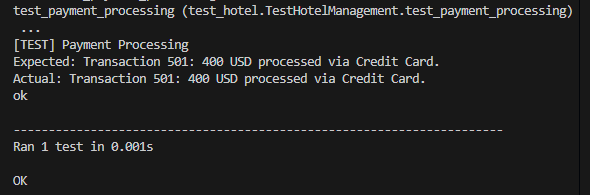
python -m unittest test\_hotel.TestHotelManagement.test\_make\_reservation –v



The Reservation class is responsible for handling the booking process, ensuring that reservations are stored, updated, and confirmed correctly. This test checks whether a guest can successfully book a room and if the system correctly assigns a reservation ID, guest ID, and room ID. The test verifies the confirm\_reservation() method, ensuring that the reservation is completed and that the room status updates accordingly. A successful test confirms that reservations are processed without errors, preventing booking failures. It ensures that guests receive confirmation upon successful booking. If the test fails, it may indicate an issue with reservation storage or room assignment. This test guarantees that the booking system functions smoothly, avoiding mismanaged reservations.

### ****Test Payment****

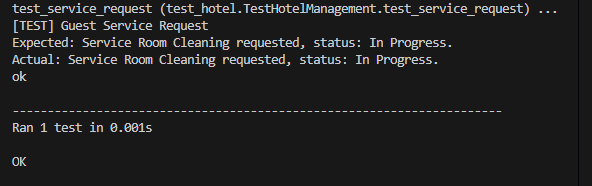
python -m unittest test\_hotel.TestHotelManagement.test\_payment\_processing –v



The Payment class handles transactions, ensuring that guests can pay for their reservations securely and efficiently. This test verifies whether payments are processed correctly, covering different payment methods such as credit cards and digital wallets. The test ensures that the process\_transaction() method accurately records payments and updates transaction history. A successful test confirms that payments are completed without errors and that invoices are generated properly. This prevents issues like double charging or failed transactions. If the test fails, it may indicate a problem with payment validation or storage. Proper payment processing ensures a seamless guest experience and financial accuracy for the hotel.

### ****Test Guest Services****

python -m unittest test\_hotel.TestHotelManagement.test\_service\_request –v



The GuestServices class manages additional services like room service, housekeeping, and transportation requests. This test checks whether guests can successfully request services and whether their requests are recorded correctly. The request\_service() method is tested to confirm that the system correctly tracks requested services and updates their status. A successful test ensures that service requests are processed efficiently, enhancing the guest experience. It prevents issues such as lost or delayed service requests. If the test fails, it may indicate a problem with request logging or status updates. Ensuring reliable guest services contributes to overall customer satisfaction in the hotel.

### ****Summary of What I Learned While Working on the Assignment****

* Developing the **Royal Inn Suite: Advanced Hotel Booking & Guest Services** system was a highly educational experience that deepened my understanding of software architecture, modular programming, and real-world problem-solving. One of the most valuable lessons I learned was how to break down a complex system into manageable components by leveraging **object-oriented programming (OOP)** principles such as **inheritance, encapsulation, and composition**. By designing separate classes for guests, employees, rooms, reservations, payments, and additional services, I was able to create a well-structured and scalable system that closely mimics actual hotel management operations.
* The implementation of the **Guest, Employee, and Room** classes helped me understand how user roles and room statuses interact within the system. Implementing reservation functionalities taught me how to effectively **manage booking conflicts, track room availability, and automate notifications**. The **Payment class** provided practical experience in handling transactions securely and generating invoices, which reinforced my knowledge of **financial processing in software applications**. Adding a **GuestServices module** expanded my perspective on how extra services like room service and transportation could be seamlessly integrated into a hotel system.
* Testing was another crucial aspect of this project. Writing unit tests for each class using Python’s unittest framework helped me see the importance of structured **test-driven development (TDD)**. Running individual test cases ensured that each component functioned correctly before integrating them into the larger system. It also reinforced best practices for **error handling, exception management, and debugging techniques**, which are essential for developing reliable software.
* Additionally, the project introduced me to the importance of **data integrity and system security**. Implementing **getter and setter methods** ensured controlled access to sensitive data, preventing unauthorized modifications. Properly managing guest reservations, cancellations, and payment transactions made me more aware of **potential security risks and data validation measures** required in real-world applications.
* Another key takeaway was learning how to manage a project efficiently. Structuring files and directories properly, maintaining clean and well-documented code, and using **GitHub for version control** provided a hands-on experience in organizing a software development workflow. Keeping a clear commit history helped track progress and made debugging easier.
* Overall, this assignment significantly enhanced my problem-solving skills and deepened my understanding of **real-world hotel management systems**. It strengthened my ability to design, develop, and test complex applications while ensuring maintainability, efficiency, and user-friendliness.