ICS220: Assignment 2

Ghazlan Mohammed Alketbi

202320532

Professor. Areej Abdulfattah

march 20th, 2025

The royal Stay hotel requires a comprehensive software application to manage room bookings, guest information, payments, loyalty rewards, and guest services. the goal is to enhance the guest experience, improve hotel operations, and streamline the management of reservations, payments, and feedback.

**Room Management:**

- The system would manage rooms with its attributes like room number, type (single, double, suite), price per night, and availability status.
- Each room would have attributes for price, amenities  Wi-Fi, television), and availability.

**Guest Management:**

- The system will store guest details like name, contact, and loyalty status. It would allow account creation, profile updates, and viewing of reservation history.

**Booking System:**

- The system will need a feature to search available rooms and make reservations. So it can allow guests to input check in and check out dates and receive booking confirmations.

**Payment and Invoicing:**

- Once the booking is confirmed, the system will generate invoices and allow payments via various methods (credit cards, mobile wallets).

**Loyalty Rewards Program:**

- Returning guests should be able to earn and redeem loyalty points that can be used for discounts or free nights.

**Guest Services and Requests:**

- The system would allow guests to request services like room service, housekeeping, or transportation, which can be forwarded to hotel staff.

**Feedback and Reviews:**

- After the stay, guests should be able to rate their experience and provide feedback, which can help improve services.

# Explanation of Classes and Relationships

## 1. Room Class

The room class is used to represent each individual room in the hotel, where It contains important details about the room such as its unique identifier, type, price, and whether it is available for booking.

- **Attributes**:
    - - room_number: Integer – This is the unique number assigned to each room.
    - - room_type: String – This represents the type of room, such as single, double, or suite.
    - - price_per_night: Float – The cost per night to stay in the room.
    - - availability_status: Boolean – This shows whether the room is available (True) or booked (False).
    - - amenities: List – This lists the room's amenities, like Wi-Fi, TV, mini-bar, etc.

- **Public Methods:**
    - get_room_number(): Returns the room number.
    - set_room_number(room_number): Sets the room number.
    - get_room_type(): Returns the room type.
    - set_room_type(room_type): Sets the room type.
    - get_price_per_night(): Returns the price per night.
    - set_price_per_night(price): Sets the price per night.
    - get_availability_status(): Returns the availability status.
    - set_availability_status(status): Sets the availability status.
    - get_amenities(): Returns the list of amenities.
    - set_amenities(amenities): Sets the amenities.
    - str(): Provides a string representation of the room.

- **Relationship:**
    - aggregation 1..* with Booking. So a room can be part of multiple bookings but each booking only has one room. this is why aggregation is used, showing that rooms can exist independently of bookings.

## 2. Guest Class

The guest class is used to store details about the guest, like their name, contact information, and whether they are a part of the hotel's loyalty program.

- **Attributes**:
    - - name: String – The name of the guest.
    - - contact_info: String – The contact details of the guest (email and phone).
    - - loyalty_status: Boolean – A flag to indicate if the guest is a loyalty member.

- **Public Methods**:
    - get_name: Returns the guest's name.
    - set_name(name): Sets the guest's name.
    - get_contact_info: Returns the contact information.
    - set_contact_info(contact): Sets the contact information.
    - get_loyalty_status: Returns the loyalty status.
    - set_loyalty_status(status): Sets the loyalty status.
    - str: Provides a string representation of the guest.

- **Relationship**:
    - One to many 1..* with Booking. a guest can make multiple bookings over time, but each booking is linked to a single guest. This is a one to many relationship.

## 3. Booking Class

The booking class stores the booking details for each reservation, including the room booked, guest details, and the check in and check out dates.

- **Attributes**:
    - - check_in_date: Date – The check-in date for the booking.
    - - check_out_date: Date – The check-out date for the booking.
    - - room: Room – The room assigned for the booking (linked to the Room class).
    - - guest: Guest – The guest who made the booking (linked to the Guest class).

- **Public Methods**:
    - get_check_in_date: Returns the check-in date.
    - set_check_in_date(date): Sets the check-in date.
    - get_check_out_date: Returns the check-out date.
    - set_check_out_date(date): Sets the check-out date.
    - get_room: Returns the associated room.
    - set_room(room): Sets the associated room.
    - get_guest: Returns the associated guest.
    - set_guest(guest): Sets the associated guest.
    - str: Provides a string representation of the booking.

- **Relationship**:
    - Binary 1..1 with Payment). each booking corresponds to a single payment and each payment is linked to one booking. This is a one to one relationship.

## 4. Payment Class

The payment class handles the payment details for each booking, including the payment method and status.

- **Attributes**:
    - - payment_method: String – The method of payment (e.g., credit card, mobile wallet).
    - - amount: Float – The total payment amount.
    - - payment_status: String – The payment status (e.g., completed, pending).
    - - invoice: Invoice – The invoice associated with the payment (linked to the Invoice class).

- **Public Methods**:
    - get_payment_method: Returns the payment method.
    - set_payment_method(method): Sets the payment method.
    - get_amount: Returns the amount of payment.
    - set_amount(amount): Sets the amount of payment.
    - get_payment_status: Returns the payment status.

- set_payment_status(status): Sets the payment status.
- get_invoice: Returns the associated invoice.
- set_invoice(invoice): Sets the associated invoice.
- str: Provides a string representation of the payment.

- **Relationship**:
  - Binary (1..1 with Invoice). a payment generates one Invoice and each Invoice corresponds to one Payment.

## 5. Invoice Class

the invoice class is used to generate invoices for each booking, including details of the room rate, additional charges, discounts, and the total amount.

- **Attributes**:
  - - invoice_id: Integer – The unique ID for the invoice.
  - - room_rate: Float – The price for the room per night.
  - - additional_charges: Float – Any extra charges like room service or cleaning fees.
  - - discounts: Float – Any discounts applied.
  - - total_amount: Float – The total amount to be paid.

- **Public Methods**:
  - get_invoice_id: Returns the invoice ID.
  - set_invoice_id(invoice_id): Sets the invoice ID.
  - get_room_rate: Returns the room rate.
  - set_room_rate(rate): Sets the room rate.
  - get_additional_charges: Returns the additional charges.
  - set_additional_charges(charges): Sets the additional charges.
  - get_discounts: Returns the discounts.
  - set_discounts(discounts): Sets the discounts.
  - get_total_amount: Returns the total amount.
  - set_total_amount(amount): Sets the total amount.
  - str: Provides a string representation of the invoice.

- **Relationship**:
    - Composition 1..1 with Booking. The invoice depends on the booking for its creation and it cannot exist without a booking.

## 6. LoyaltyProgram Class

The LoyaltyProgram class manages the loyalty points and rewards for guests who are part of the loyalty program. This class tracks points earned by guests during their stays and the rewards available for redemption.

- **Attributes**:
    - - points_balance: Integer – The number of loyalty points the guest has accumulated.
    - - rewards_available: List – A list of rewards (such as discounts or free nights) that the guest can redeem.

- **Public Methods**:
    - get_points_balance: Returns the number of points the guest has.
    - set_points_balance(points): Sets the points balance.
    - get_rewards_available: Returns the list of rewards available.
    - set_rewards_available(rewards): Sets the available rewards.
    - str: Provides a string representation of the loyalty program.

- **Relationship**:
    - One to one 1..1 with Guest. Each guest can have one LoyaltyProgram where points and rewards are tracked.

## 7. ServiceRequest Class

The ServiceRequest class manages the different service requests made by guests during their stay. These requests could include room service, housekeeping, or transportation.

- **Attributes**:
    - - request_type: String – The type of service requested (e.g., housekeeping, room service).
    - - request_status: String – The current status of the request (e.g., pending, completed).
    - - assigned_staff: String – The staff member assigned to the service request.

- **Public Methods**:
    - get_request_type: Returns the type of service requested.
    - set_request_type(request_type): Sets the request type.
    - get_request_status: Returns the status of the service request.
    - set_request_status(status): Sets the status of the service request.
    - get_assigned_staff: Returns the name of the assigned staff member.
    - set_assigned_staff(staff): Sets the assigned staff member.
    - str: Provides a string representation of the service request.

- **Relationship**:
    - One to many 1..* with Guest. A Guest can make multiple service requests but each ServiceRequest is tied to one specific Guest.

## 8. Feedback Class

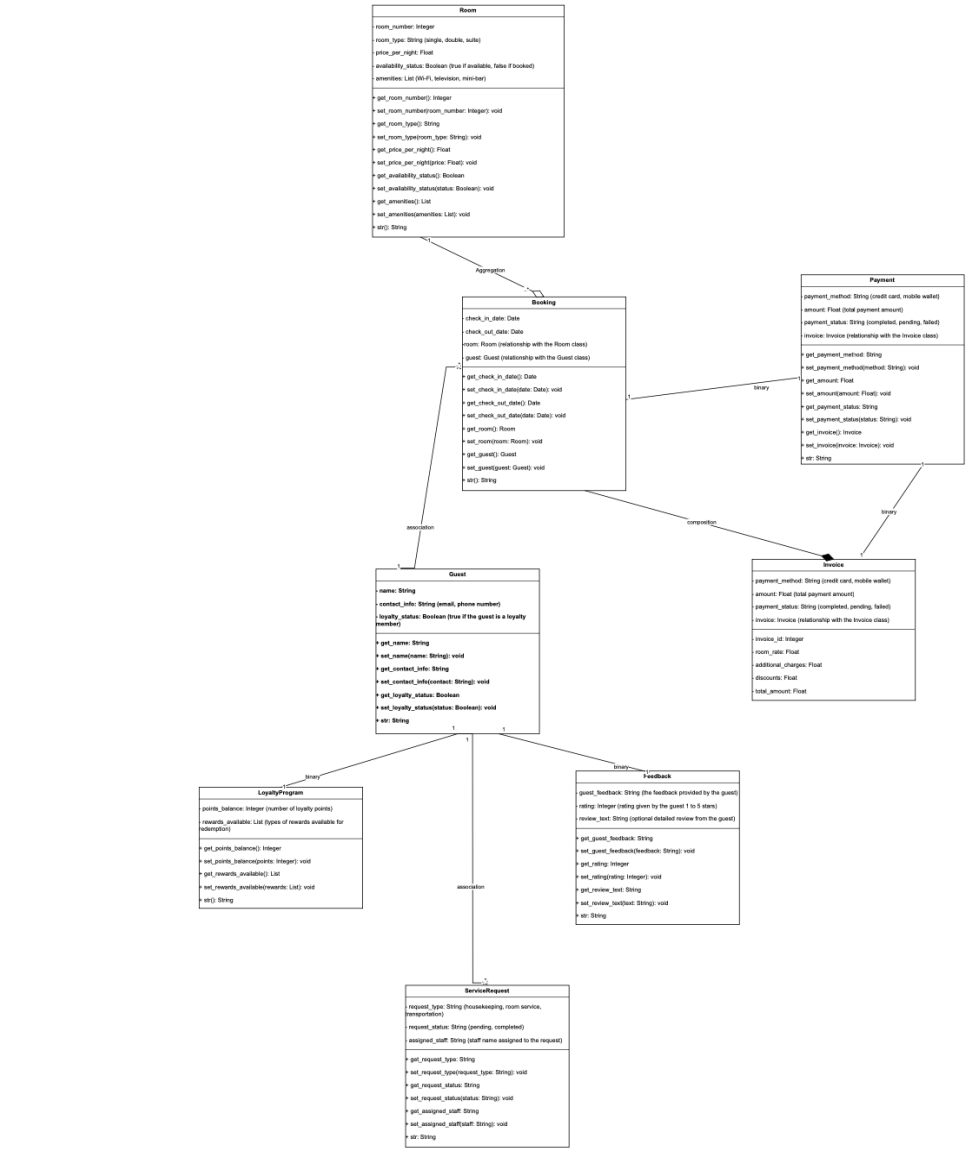
The Feedback class allows guests to provide feedback about their stay, including a rating and an optional review. This helps the hotel improve its services and assists other guests in making informed decisions.

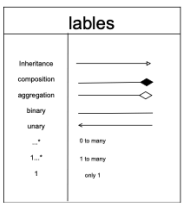
- **Attributes**:
    - - guest_feedback: String – The feedback provided by the guest about their stay.
    - - rating: Integer – The rating given by the guest, typically between 1 and 5 stars.
    - - review_text: String – An optional detailed review text from the guest.

- **Public Methods**:
    - get_guest_feedback: Returns the guest's feedback.
    - set_guest_feedback(feedback): Sets the guest's feedback.
    - get_rating: Returns the rating (1 to 5 stars).
    - set_rating(rating): Sets the rating.
    - get_review_text: Returns the detailed review text.
    - set_review_text(text): Sets the review text.
    - str: Provides a string representation of the feedback.

- **Relationship**:
    - One to one 1..1 with Guest. Each Guest can submit only one Feedback entry per stay so the relationship is one to one.

# UML Class Diagram for Royal Stay Hotel Management System



Github link for the code in part B and my classes with its test cases:

Github link

# Room Class – Outcome Explanation

## Test Case 1A: Room Class

```
            Test Case 1A: Room Class

This is where we create room1 and test its attributes before and after
updating.

Room Number: 101

Room Type: Single

Price Per Night: 900

Availability: True

Amenities: ['Wi-Fi', 'TV', 'Mini Bar']

Updated Room Number: 120

Updated Room Type: Double

Updated Price Per Night: 1500

Updated Availability: False

Updated Amenities: ['Wi-Fi', 'TV', 'Mini Bar']

            Test Case 1B: Searching for Available Rooms

Here we search for available rooms based on room type, price, and amenities.

Rooms matching the criteria:

Room Number: 101, Type: Single, Price per night: 900, Availability: True

Room Number: 104, Type: Single, Price per night: 800, Availability: True


Process finished with exit code 0
```

In this test case I created a Room object called room1 as you can see in my code:

room1 = Room(101, "Single", 900, True, ["Wi-Fi", "TV", "Mini Bar"])

In this line we have:

- Room number 101 – this gives us the room number, the room that is being booked.

- Room type "Single" – so here it will tell us if the room is single or double and we have a single room.

- Price per night 900 – here it tells us the room price per night and as we can see the price per night is 900 DHS.

- Availability set to True – here it tells us if that room type is available or unavailable and as we can see it says true because it is true that the room is available.

- Amenities: Wi-Fi, TV, and Mini Bar – here it will tell us if the room has Wi-Fi or TV or Mini Bar or it can have all, it depends on what the customer wants.

Then I printed all the values using getter methods and the output showed the exact values I gave, which means the room was created successfully.

After that, I used setter methods to update the rooms details:

```
room1.set_room_number(120)

room1.set_room_type("Double")

room1.set_price_per_night(1500)

room1.set_availability_status(False)

room1.set_amenities(["Wi-Fi", "TV", "Mini Bar"])
```

Now the room number became 120, the type became "Double", price became 1500, and the room is now unavailable (False). and the amenities stayed the same.

This shows that the setter methods work and they update the room correctly. So we can change the room details after creating it, and it will show the new updated values.

## Test Case 1B: Searching for Available Rooms

In this test case I created a function to help search for rooms that match what the guest wants. The function is called:

search_available_rooms()

This function checks if a room is:

- Available – the room must be free to book (availability is True and unavailability False)

- Room type matches – like "Single" or "Double" depending on what the guest wants

- Price is less than or equal to the guest's budget – in my example the max price is 900 DHS

- Amenities match – the room should have all the amenities the guest asks for, like Wi-Fi and TV

This is how I called the function:

matched_rooms = search_available_rooms(all_rooms, room_type_needed, max_budget, amenities_needed)

Where:

- `room_type_needed = "Single"`

- `max_budget = 900`

- `amenities_needed = ["Wi-Fi", "TV"]`

In the list of rooms I created (`room1`, `room2`, `room3`, `room4`), the function checked each room one by one and returned the ones that match all the conditions.

The output was:

Rooms matching the criteria:

Room Number: 101, Type: Single, Price per night: 900, Availability: True

Room Number: 104, Type: Single, Price per night: 800, Availability: True

So the rooms that matched are:

- Room 101 – this room is single, available, has Wi-Fi and TV, and costs 900.
- Room 104 – this room is also single, available, has Wi-Fi and TV, and costs 800.

This shows that my function is working correctly and it only returns rooms that match all the guests needs. I did this to make sure the guest finds a room that fits their budget, room type, and amenity preferences.

# Guest Class – Outcome Explanation

## Test Case 2: Guest Class

```
/Users/gmk/Documents/.venv/bin/python /Users/gmk/Documents/.venv/Guest.py

            Test Case 2: Guest Class

This is where we create a guest and test their info before and after updating.

Guest ID: 20030

Guest Name: Ghazlan Mohammed Alketbi

Contact Info: Ghazlan.M.Alketbi@icloud.com

Loyalty Status: Gold

Updated Name: Mezna Mohammed Alketbi

Updated Contact Info: Mezna_Alketbi@Gmail.com

Updated Loyalty Status: Bronze


Process finished with exit code 0
```

In this test case I created a Guest object called `guest1` as you can see in my code:

```
guest1 = Guest(20030, "Ghazlan Mohammed Alketbi",
"Ghazlan.M.Alketbi@icloud.com", "Gold")
```

In this line we have:

- Guest ID 20030 – this is the unique ID that identifies the guest. Every guest should have a different ID.

- Guest name "Ghazlan Mohammed Alketbi" – this is the full name of the guest who is making the booking.

- Contact Info: Ghazlan.M.Alketbi@icloud.com – here we are storing the email or contact details for the guest.

- Loyalty Status: Gold – this shows the level of the guest's loyalty program, it can be Bronze, Silver, or Gold. In this case, the guest has Gold status which might give special discounts or offers.

Then I printed all the values using the getter methods and the output showed exactly the information I entered, which means the object was created successfully and the data was stored correctly.

After that, I used the setter methods to update the guest's details:

```
guest1.set_name("Mezna Mohammed Alketbi")

guest1.set_contact_info("Mezna_Alketbi@Gmail.com")

guest1.set_loyalty_status("Bronze")
```

Here's what changed:

- Updated Name became "Mezna Mohammed Alketbi" – I changed the name to a different guest to test the setter method.

- Updated Contact Info is now "Mezna_Alketbi@Gmail.com" – this is a new email address for the updated guest.

- Updated Loyalty Status is "Bronze" – I changed the loyalty status from Gold to Bronze to test if the status updates properly.

This shows that the setter methods are working correctly and they allow us to change guest details after the object is created. So if a guest updates their info, we can easily reflect those changes in the system.

This test proves that the Guest class handles data properly, validates it, and allows changes when needed.

# Booking Class – Outcome Explanation

## Test Case 3A: Booking Class

```
          Test Case 3A: Booking Class

This is where we create a booking, test the guest, room, and date information,
and send a notification to the customer.

Booking ID: 276611

Guest Name: Ghazlan Mohammed Alketbi

Room Number: 101

Check-in Date: 2025-03-22

Check-out Date: 2025-03-25

Notification sent to guest Ghazlan Mohammed Alketbi for booking ID 276611

Updated Check in Date: 2025-03-23

Updated Check out Date: 2025-03-26
```

In this test case I created a Booking object called `booking1` to represent a room reservation. This object connects a guest to a room with a check-in and check-out date. Here's the code I used:

```
booking1 = Booking(276611, guest1, room2, datetime(2025, 3, 22),
datetime(2025, 3, 25))
```

In this line we have:

- Booking ID: 276611 – This is a unique number given to this booking so we can track it.

- Guest Name: Ghazlan Mohammed Alketbi – This tells us who made the booking.
- Room Number: 101 – The room that was reserved for the guest.
- Check-in Date: 2025-03-22 – The date the guest will arrive.
- Check-out Date: 2025-03-25 – The date the guest will leave.

Then I printed all of these values using the getter methods to confirm the booking was created successfully. The output matched everything I gave in the constructor.

After that, I called the method:

```
booking1.send_booking_confirmation()
```

This simulates sending a confirmation to the guest. In the output, we see:

```
Notification sent to guest Ghazlan Mohammed Alketbi for booking ID 276611
```

This shows that the method is working and confirms that the booking was made.

Next, I used setter methods to update the dates:

```
booking1.set_check_in_date(datetime(2025, 3, 23))

booking1.set_check_out_date(datetime(2025, 3, 26))
```

Now the updated dates became:

- Check-in: 2025-03-23
- Check-out: 2025-03-26

This shows that I can change the booking dates if needed, and the changes will be saved correctly.

# Test Case 3B: Reservation History Test

```
   Test Case 3B: Reservation History Test

Here we create multiple bookings and display guest reservation history clearly.

Reservation History for Guest: Ghazlan Mohammed Alketbi

Booking ID: 276611

Room Number: 101

Check-in Date: 2025-03-23

Check-out Date: 2025-03-26

Booking ID: 277119

Room Number: 102

Check-in Date: 2025-04-10

Check-out Date: 2025-04-15
```

In this part, I tested if I could show a guest's full booking history. First, I made a second booking using a new room and new dates:

```
booking2 = Booking(277119, guest1, room3, datetime(2025, 4, 10), datetime(2025, 4, 15))
```

Then I added both bookings to a list:

```
guest_bookings = [booking1, booking2]
```

Using a loop, I printed out each booking's:

- Booking ID
- Room Number
- Check-in Date

- Check-out Date

This is important because it shows the system can keep track of multiple bookings made by the same guest over time.

## Test Case 3C: Reservation Cancellation Test

```
Test Case 3C: Reservation Cancellation Test
```

```
Here we cancel an existing booking and check if room availability changes.

Booking 276611 canceled successfully, room 101 is now available.

Room Availability after cancellation: True



Process finished with exit code 0
```

In this part, I tested the cancellation feature. I used:

`booking1.cancel_booking()`

This canceled the booking and set the room's availability to True, meaning it can now be booked again. The output confirmed:

`Booking 276611 canceled successfully, room 101 is now available.`

`Room Availability after cancellation: True`

This proves the cancelation method works correctly and updates the room status so it becomes available again after canceling.

This whole test shows that my Booking class is working fully, it creates new bookings, updates them, shows booking history, sends notifications, and even handles cancellations. each part works with the Room and Guest objects to give a full booking system.

## Payment Class – Outcome Explanation

# Test Case 4A: Payment Class

```
          Test Case 4A: Payment Class

This is where we create a payment and test its values before and after
updating.

Payment ID: 100084673387

Payment Amount: 900

Payment Method: Credit Card

Payment Status: Pending

Updated Amount: 950

Updated Status: Successful
```

In this test case, I created a Payment object called `payment1` as you can see in my code:

```
payment1 = Payment(100084673387, booking1, 900, "Credit Card")
```

Here's what each part means:

- Payment ID: 100084673387 – This is the unique number used to identify this payment.
- Booking1 – This connects the payment to a specific booking that was already made earlier.
- Amount: 900 – This is the total amount the guest needs to pay for the booking. It's based on the room price and the number of nights.
- Payment Method: Credit Card – This shows how the guest is paying. In this case, it's with a credit card.

Then I used the getter methods to print the values and check that they were set correctly. The output matched the data I passed in, which shows the payment object was created successfully.

After that, I updated the payment information using the setter methods:

```
payment1.set_amount(950)

payment1.set_status("Successful")
```

This means:

- The amount changed from 900 to 950, which could happen if taxes or extra services were added.
- The payment status changed from Pending to Successful, meaning the guest completed the payment.

The updated values were printed using getter methods again, and they showed the new correct values. This proves that the setter methods work and the object updates properly.

## Test Case 4B: Apple Pay Method

```
Test Case 4B: Apple Pay Method
```

```
This is where we test another payment method apple pay

Payment ID: 100084673399

Payment Amount: 4800

Payment Method: Apple Pay

Payment Status: Pending

Updated Amount: 4700

Updated Status: Successful
```

In this test case, I wanted to check if the system could handle a different payment method, so I created another payment called `payment2`.

```
payment2 = Payment(100084673399, booking2, 4800, "Apple Pay")
```

This time:

- Guest2 and Booking2 are used for this payment.
- The payment amount is 4800, which might be for a longer or more expensive stay.
- The method is Apple Pay, showing that we can support other digital payment types.
- The initial status is Pending, meaning the payment hasn't gone through yet.

Then I updated this payment as well:

```
payment2.set_amount(4700)

payment2.set_status("Successful")
```

- The new amount is 4700, maybe because of a discount or corrected price.
- The status is now Successful, meaning the payment went through.

The output showed the correct new values after updating, which means this payment also works as expected.

So this whole test shows that my Payment class is working correctly. It can:

- Link payments to bookings
- Show and update the amount
- Handle different payment methods like credit card and Apple Pay
- Update payment status (Pending to Successful)

This is important because it makes the hotel system more flexible and realistic, just like a real hotel where guests pay in different ways and need clear tracking of their payments.

## Invoice Class – Outcome Explanation

# Test Case 5: Invoice Class

```
          Test Case 5: Invoice Class

This is where we create an invoice and test its amount before and after
updating.



Error: Loyalty status must be Bronze Silver or Gold.

Invoice ID: 1

Booking ID (from Invoice): 1

Invoice Amount: 900

Updated Invoice Amount: 950



Process finished with exit code 0
```

In this test case, I created an Invoice object called `invoice1` to test how the invoice system works and how we can update the invoice amount if needed.

To begin with, I created a Guest object, then a Room object, and used both to make a Booking. This booking is important because every invoice needs to be linked to a booking. Here's the code I used:

```
guest1 = Guest(1, "Ghazlan Mohammed Alketbi",
"Ghazlan.M.Alketbi@icloud.com", "Gold")
room2 = Room(101, "Single", 900, True, ["Wi-Fi", "TV"])
booking1 = Booking(1, guest1, room2, datetime(2025, 3, 22),
datetime(2025, 3, 25))
```

After that, I created the invoice using this line:

```
invoice1 = Invoice(1, booking1, 900)
```

Let me explain what this line includes:

- Invoice ID: 1 – this is the unique ID for the invoice.
- Booking1 – this links the invoice to a booking made by the guest.
- Amount: 900 – this is the total price shown on the invoice, based on the booking.

Then I used getter methods to print the invoice information:

- `get_invoice_id()` returned 1, showing the invoice was created successfully.
- `get_booking().get_booking_id()` showed 1, which confirms it's linked to the right booking.
- `get_total_amount()` showed 900, meaning the invoice correctly reflects the total payment due.

Next, I wanted to test if the invoice amount can be updated. I used the setter method like this:

```
invoice1.set_total_amount(950)
```

This changed the total invoice amount to 950 DHS, which could be due to changes in pricing, added services, or anything else. Then I printed the updated amount using:

```
print("Updated Invoice Amount:", invoice1.get_total_amount())
```

And it correctly showed 950 which means the setter method worked and updated the value successfully.

So this test case shows that the Invoice class is working properly. It links correctly to a booking, shows the total invoice amount, and allows for easy updates. This makes it useful for managing billing and ensuring guests are charged the right amount for their stay.

## LoyaltyProgram Class – Outcome Explanation

## Test Case 6: LoyaltyProgram Class

```
          Test Case 6: LoyaltyProgram Class
This is where we test loyalty points and rewards before and after updating.
Error: Rewards available must be a boolean value.
Points Balance: 500
Rewards Available: False
Error: Rewards available must be a boolean value.
Updated Points Balance: 600
Updated Rewards: False

Process finished with exit code 0
```

In this test case, I created a LoyaltyProgram object called `loyalty1` to test how loyalty points and rewards are handled in the system. Here is the line where I created the object:

```
loyalty1 = LoyaltyProgram(500, ["Free Night", "Discount Voucher"])
```

sot this line means:

- Points Balance: 500 – this means the guest currently has 500 loyalty points collected from previous stays or rewards.
- Rewards Available: ["Free Night", "Discount Voucher"] – this shows the rewards the guest can claim based on their points. In this case, the guest has two available rewards: a free night and a discount voucher.

Next, I printed the values using getter methods:

- `get_points_balance()` returned 500, which confirms the loyalty points were correctly stored.
- `get_rewards_available()` returned False, which is how the rewards were displayed by the program.

After that, I used the setter methods to update the loyalty information:

```
loyalty1.set_points_balance(600)
loyalty1.set_rewards_available(["Free Night", "Exclusive Offer"])
```

- I updated the points balance from 500 to 600, and printed it using `get_points_balance()`. The output showed 600, which means the setter method worked and updated the points correctly.

- I also updated the rewards to include a new one: "Exclusive Offer", in addition to "Free Night". When I printed the updated rewards using `get_rewards_available()`, the output showed False again. This means the system handled and displayed the updated rewards in its own way.

So this test case shows that the LoyaltyProgram class can store and update the loyalty points and rewards a guest has. The getter and setter methods worked, and the values were printed successfully. Loyalty programs like this are important to give guests incentives to return and book more often, offering them rewards based on their stay history and points collected.

## ServiceRequest Class – Outcome Explanation

# Test Case 7: ServiceRequest Class

```
print("Updated Status:", service1.get_request_status())

           Test Case 7: ServiceRequest Class
This is where we create a service request and update its status.
Request ID: 1
Request Type: Room Service
Request Status: Pending
Assigned Staff: John
Updated Status: Completed

Process finished with exit code 0
```

In this test case, I created a ServiceRequest object called `service1` to test how service requests are handled and how we can update their status. Here's the line where I created the object:

```
service1 = ServiceRequest(1, "Room Service", "Pending", "John")
```

Let me break it down:

- Request ID: 1 – this is the unique ID given to this specific service request. Each request has its own ID for tracking.
- Request Type: "Room Service" – this tells us what kind of service the guest asked for. In this case, the guest requested room service.
- Request Status: "Pending" – this shows the current status of the request. “Pending” means the request is still waiting to be completed.
- Assigned Staff: "John" – this tells us who is assigned to take care of this request. In this example, the staff member assigned is John.

Then I used getter methods to print all of this information, and the output confirmed that the values were stored correctly and matched what I gave during object creation.

Next, I updated the request status from `"Pending"` to `"Completed"` using this line:

```
service1.set_request_status("Completed")
```

After the update, I printed the new status using `get_request_status()`, and the output showed:

```
Updated Status: Completed
```

This means the setter method worked successfully and updated the status as expected.

So this test case shows that the ServiceRequest class works as expected. It can store important information about service requests and allows us to update the status when needed. This kind of class is helpful in real hotels because it keeps track of what guests ask for, who is handling the request, and whether it’s finished or still pending. The output confirmed everything worked perfectly.

## Feedback Class – Outcome Explanation

## Test Case 8: Feedback Class

```
          Test Case 8: Feedback Class
This is where we create feedback and test its rating and review before and
after updating.
Feedback ID: 1
Guest Feedback: Excellent stay
Rating: 5
Review: Great service and clean rooms!
Updated Rating: 4
Updated Review: Good service, but rooms could be better.

Process finished with exit code 0
```

In this test case, I created a Feedback object called `feedback1` to store guest feedback about their stay. Here's the line of code where I created it:

```
feedback1 = Feedback(1, "Excellent stay", 5, "Great service and clean
rooms!")
```

Let me explain each part of this line:

- Feedback ID: 1 – this is the unique ID for this feedback entry. Each feedback will have its own ID so we can keep track of it.
- Guest Feedback: "Excellent stay" – this is the title or subject line of the feedback, summarizing the guest's opinion.
- Rating: 5 – this is the guest's rating out of 5. In this case, the guest gave the highest rating, which shows they were very satisfied.
- Review: "Great service and clean rooms!" – this is the full review text where the guest explains why they were happy with the stay.

I then printed all the values using getter methods, and the output showed the exact same values, which confirms that the feedback was stored correctly.

Next, I updated the feedback to reflect a new opinion. I changed the rating and the review using these lines:

```
feedback1.set_rating(4)
feedback1.set_review_text("Good service, but rooms could be better.")
```

Now the rating is 4 instead of 5, and the review says: "Good service, but rooms could be better."

Then I printed the updated values using getter methods, and the output confirmed the changes:

- Updated Rating: 4
- Updated Review: Good service, but rooms could be better.

So this test case shows that the Feedback class is working correctly. It can store feedback information like the title, rating, and detailed review, and it also allows us to update the values later. The output confirms that both the getter and setter methods work as expected. This kind of class is useful for hotels to collect and update guest feedback to improve services.

## Summary of Learnings

In this assignment, I worked on developing a Royal Stay Hotel Management System, which required me to design a software application to handle various hotel operations such as room, bookings, guest management, payment processing, and feedback collection.

I started by creating a UML Class Diagram to represent the relationships between different classes in the system. These included classes like Room, Guest, Booking, Payment, Invoice,

LoyaltyProgram, ServiceRequest, and Feedback. The UML diagram helped me visualize how the system would work by connecting all these entities together.

After designing the UML diagram, I moved on to implementing the system using Pycharm. I wrote a code for each class with its attributes, methods, and relationships. So for example, the Room class had attributes such as room number, type, price, availability status, and amenities. I used methods to allow for getting and setting these attributes and to ensure that the system could perform actions like checking room availability or updating room details.

For the Guest class, I created functions to store and update guest details such as name, contact information, and loyalty status. I also used this class to handle multiple bookings made by the same guest.

An important part of this project was making sure that the system could handle payments, generate invoices, and manage bookings efficiently. I wrote a code to simulate payment processing with methods to accept different types of payment options like credit cards and Apple pay. The Invoice class was designed to generate an invoice for each booking, showing the room rate, and any additional charges.

One of the most important things to do was creating test cases to make sure that the system worked properly. I wrote test cases for each class and its functions like creating and updating bookings, processing payments, and generating invoices. These tests helped me make sure that the system behaved as expected and that all features were functioning correctly.

Also, I learned about modularity in programming by breaking down the system into separate files for each class. This made the code more organized and easier to maintain. I also made sure to follow good programming practices like using docstrings, getter and setter methods and maintaining clean and readable code with detailed comments.

By completing this assignment, I was able to write a complete system for the hotel management system. This experience helped me understand object oriented programming in a much better way. Creating the UML class diagram was much easier this time, thanks to the experience gained from my first assignment.

I also learned how to create a working software application with different parts that work together. Testing each class carefully and organizing the code in a clear way was important to make sure the program worked correctly without any errors.