In [1]:
```python
#Part B:
# Class 1
class Room:
    # Constructor
    def __init__(self, room_number, room_type, amenities=None, price_per_night=0.0):
        # Private attributes
        self._room_number = room_number
        self._type = room_type
        self._amenities = amenities if amenities else []
        self._price_per_night = price_per_night
        self._available = True

#This class does not directly reference other classes in its attributes
#It is referenced by the Booking class (composition relationship - a booking contains a room)


    # Getter and setter methods
    def get_room_number(self):
        return self._room_number

    def get_type(self):
        return self._type

    def set_type(self, room_type):
        self._type = room_type

    def get_amenities(self):
        return self._amenities

    def set_amenities(self, amenities):
        self._amenities = amenities

    def get_price_per_night(self):
        return self._price_per_night

    def set_price_per_night(self, price):
```

```
        self._price_per_night = price

    def is_available(self):
        return self._available

    def set_available(self, status):
        self._available = status

    # Methods
    def update_availability(self):
        # Update room availability based on current bookings
        # This would typically check the booking database
        pass

    def get_details(self):
        # Return a string representation of room details
        amenities_str = ", ".join(self._amenities)
        return f"Room {self._room_number} ({self._type}): ${self._price_per_night:.2f}/night, Amenities: {ameniti

    # String representation
    def __str__(self):
        status = "Available" if self._available else "Booked"
        return f"Room {self._room_number} - Type: {self._type} - Price: ${self._price_per_night:.2f}/night - Stat
```

In [2]:
```
#Class 2:
class Guest:
    # Constructor
    def __init__(self, guest_id, name, contact_info, loyalty_points=0):
        # Private attributes
        self._guest_id = guest_id
        self._name = name
        self._contact_info = contact_info
        #The _loyalty_points attribute indicates an aggregation relationship with LoyaltyProgram
        self._loyalty_points = loyalty_points
        #The Guest "has" loyalty points, but doesn't "own" or create the LoyaltyProgram itself
        #If the LoyaltyProgram were to be deleted, the Guest could still exist independently
        #The loyalty points are a "part" of the Guest that can be separated without destroying the Guest
```

```
#This class is referenced by the Booking class (association - a booking is made by a guest)
#This class is referenced by the ServiceRequest class (association - a guest makes service requests)
#This class is referenced by the Feedback class (aggregation - a guest provides feedback)

    # Getter and setter methods
    def get_guest_id(self):
        return self._guest_id

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def get_contact_info(self):
        return self._contact_info

    def set_contact_info(self, contact_info):
        self._contact_info = contact_info

    def get_loyalty_points(self):
        return self._loyalty_points

    def set_loyalty_points(self, points):
        self._loyalty_points = points

    # Methods
    def update_profile(self, name=None, contact_info=None):
        # Update guest profile information
        if name:
            self._name = name
        if contact_info:
            self._contact_info = contact_info
        return "Profile updated successfully"

    def view_reservation_history(self):
```

```
        # In a real system, this would fetch booking history from a database
        # For simplicity, we return a placeholder message
        return f"Reservation history for {self._name}"

    def redeem_points(self, points_to_redeem):
        # Redeem loyalty points for rewards
        if points_to_redeem <= self._loyalty_points:
            self._loyalty_points -= points_to_redeem
            return f"Redeemed {points_to_redeem} points. Remaining points: {self._loyalty_points}"
        else:
            return f"Insufficient points. Current balance: {self._loyalty_points}"

    # String representation
    def __str__(self):
        return f"Guest ID: {self._guest_id} - Name: {self._name} - Contact: {self._contact_info} - Loyalty Points
```

In [3]:

```
#Class 3:
from datetime import datetime

class Booking:
    # Constructor
    def __init__(self, booking_id, guest, room, check_in_date, check_out_date):
        # Private attributes
        self._booking_id = booking_id
        self._guest = guest # Association with Guest (1 to many)
        self._room = room # Composition with Room (1 to many)
        self._check_in_date = check_in_date
        self._check_out_date = check_out_date
        self._status = "Confirmed"  # Default status

#In here if the Guest object is deleted or modified, the Booking object continues to exist
#The Booking simply holds a reference to the Guest
#If you try to access booking._guest after the Guest object is garbage collected, you might get a reference to
#a non-existent object
```

```
    # Getter and setter methods
    def get_booking_id(self):
        return self._booking_id

    def get_guest(self):
        return self._guest

    def get_room(self):
        return self._room

    def get_check_in_date(self):
        return self._check_in_date

    def set_check_in_date(self, date):
        self._check_in_date = date

    def get_check_out_date(self):
        return self._check_out_date

    def set_check_out_date(self, date):
        self._check_out_date = date

    def get_status(self):
        return self._status

    def set_status(self, status):
        self._status = status

    # Methods
    def calculate_total_price(self):
        # Calculate number of days
        delta = self._check_out_date - self._check_in_date
        num_days = delta.days

        # Calculate total price
        total_price = num_days * self._room.get_price_per_night()
        return total_price
```

```
    def confirm_booking(self):
        # Confirm the booking and update room availability
        self._status = "Confirmed"
        self._room.set_available(False)
        return f"Booking {self._booking_id} confirmed for {self._guest.get_name()}"

    def cancel_booking(self):
        # Cancel the booking and update room availability
        self._status = "Cancelled"
        self._room.set_available(True)
        return f"Booking {self._booking_id} cancelled for {self._guest.get_name()}"

    # String representation
    def __str__(self):
        return (f"Booking ID: {self._booking_id} - Guest: {self._guest.get_name()} - "
                f"Room: {self._room.get_room_number()} - Check-in: {self._check_in_date.strftime('%Y-%m-%d')} - "
                f"Check-out: {self._check_out_date.strftime('%Y-%m-%d')} - Status: {self._status}")
```

In [4]:

```
#Class 4:
from datetime import datetime

class Payment:
    # Constructor
    def __init__(self, payment_id, booking, amount, payment_method):
        # Private attributes
        self._payment_id = payment_id
        self._booking = booking # Composition with Booking
        self._amount = amount
        self._payment_date = datetime.now()
        self._payment_method = payment_method

# In python, when the containing object is deleted, the contained object should also be deleted
#The contained object cannot exist without its container
# Python doesn't enforce this automatically through garbage collection
# Typically implemented by creating the contained object inside the container's constructor

    # Getter and setter methods
```

```
    def get_payment_id(self):
        return self._payment_id

    def get_booking(self):
        return self._booking

    def get_amount(self):
        return self._amount

    def set_amount(self, amount):
        self._amount = amount

    def get_payment_date(self):
        return self._payment_date

    def get_payment_method(self):
        return self._payment_method

    def set_payment_method(self, method):
        self._payment_method = method

    # Methods
    def process_payment(self):
        # Process the payment
        # In a real system, this would connect to a payment gateway
        return True

    def generate_receipt(self):
        # Generate a receipt for the payment
        receipt = (f"Receipt for Payment #{self._payment_id}\n"
                   f"Date: {self._payment_date.strftime('%Y-%m-%d %H:%M:%S')}\n"
                   f"Amount: ${self._amount:.2f}\n"
                   f"Method: {self._payment_method}\n"
                   f"Booking ID: {self._booking.get_booking_id()}\n"
                   f"Guest: {self._booking.get_guest().get_name()}")
        return receipt

    def refund_payment(self):
```

```
        # Process a refund
        # In a real system, this would connect to a payment gateway
        return True

    # String representation
    def __str__(self):
        return (f"Payment ID: {self._payment_id} - Amount: ${self._amount:.2f} - "
                f"Date: {self._payment_date.strftime('%Y-%m-%d')} - Method: {self._payment_method}")
```

In [5]:
```
#Class 5:
class LoyaltyProgram:
    # Constructor
    def __init__(self, program_id, program_name, points_per_dollar, rewards=None):
        # Private attributes
        self._program_id = program_id
        self._program_name = program_name
        self._points_per_dollar = points_per_dollar
        self._rewards = rewards if rewards else []

    # Getter and setter methods
    def get_program_id(self):
        return self._program_id

    def get_program_name(self):
        return self._program_name

    def set_program_name(self, name):
        self._program_name = name

    def get_points_per_dollar(self):
        return self._points_per_dollar

    def set_points_per_dollar(self, points):
        self._points_per_dollar = points

    def get_rewards(self):
        return self._rewards
```

```
    def add_reward(self, reward):
        self._rewards.append(reward)

    # Methods
    def earn_points(self, amount):
        # Calculate points earned based on amount spent
        points_earned = int(amount * self._points_per_dollar)
        return points_earned

    def redeem_reward(self, points, reward_name):
        # Find the reward with the given name
        for reward in self._rewards:
            if reward.get("name") == reward_name:
                if points >= reward.get("points"):
                    return reward.get("reward")
                else:
                    return "Insufficient points for this reward"
        return "Reward not found"

    def get_available_rewards(self, points):
        # Return a list of rewards that can be redeemed with the given points
        available_rewards = []
        for reward in self._rewards:
            if points >= reward.get("points"):
                available_rewards.append(reward)
        return available_rewards

    # String representation
    def __str__(self):
        return (f"Loyalty Program: {self._program_name} (ID: {self._program_id}) - "
                f"Points per dollar: {self._points_per_dollar} - Rewards: {len(self._rewards)}")
```

In [6]:
```
#Class 6:
from datetime import datetime

class ServiceRequest:
```

```
    # Constructor
    def __init__(self, request_id, guest, request_type, status="Pending"):
        # Private attributes
        self._request_id = request_id
        self._guest = guest # Association with Guest
        self._request_type = request_type
        self._request_date = datetime.now()
        self._status = status

#The relationship is "uses" rather than "owns" or "is part of"
#The relationship simply establishes that a guest is associated with a service request
#If the Guest object were deleted, the ServiceRequest could still exist (lose access to guest information)

    # Getter and setter methods
    def get_request_id(self):
        return self._request_id

    def get_guest(self):
        return self._guest

    def get_request_type(self):
        return self._request_type

    def set_request_type(self, request_type):
        self._request_type = request_type

    def get_request_date(self):
        return self._request_date

    def get_status(self):
        return self._status

    def set_status(self, status):
        self._status = status

    # Methods
    def submit_request(self):
        # Submit a new service request
```

```
        self._status = "Submitted"
        return f"Request {self._request_id} submitted successfully"

    def update_status(self, status):
        # Update the status of the request
        self._status = status
        return f"Request {self._request_id} status updated to {status}"

    def assign_to_staff(self, staff_id):
        # Assign the request to a staff member
        # In a real system, this would update a staff assignment field
        return f"Request {self._request_id} assigned to staff member {staff_id}"

    # String representation
    def __str__(self):
        return (f"Service Request ID: {self._request_id} - Type: {self._request_type} - "
                f"Guest: {self._guest.get_name()} - Status: {self._status} - "
                f"Date: {self._request_date.strftime('%Y-%m-%d %H:%M')}")
```

In [7]:
```
#Class 7:
from datetime import datetime

class Invoice:
    # Constructor
    def __init__(self, invoice_id, booking, total_amount):
        # Private attributes
        self._invoice_id = invoice_id
        self._booking = booking
        self._total_amount = total_amount
        self._issue_date = datetime.now()

    # Getter and setter methods
    def get_invoice_id(self):
        return self._invoice_id

    def get_booking(self):
        return self._booking
```

```
    def get_total_amount(self):
        return self._total_amount

    def set_total_amount(self, amount):
        self._total_amount = amount

    def get_issue_date(self):
        return self._issue_date

    # Methods
    def generate_invoice(self):
        # This method demonstrates the composition relationship
        # by directly accessing and using Booking's internal data

# Invoice cannot exist without a Booking - it's completely dependent on it, relies on and directly uses
#Booking's internal data and relationships
#f the Booking were deleted, the Invoice would be meaningless and couldn't function


        # Generate an invoice with details
        guest = self._booking.get_guest()
        room = self._booking.get_room()

        invoice_details = (
            f"INVOICE #{self._invoice_id}\n"
            f"Date: {self._issue_date.strftime('%Y-%m-%d')}\n"
            f"Guest: {guest.get_name()}\n"
            f"Room: {room.get_room_number()} ({room.get_type()})\n"
            f"Check-in: {self._booking.get_check_in_date().strftime('%Y-%m-%d')}\n"
            f"Check-out: {self._booking.get_check_out_date().strftime('%Y-%m-%d')}\n"
            f"Total Amount: ${self._total_amount:.2f}\n"
            f"Thank you for choosing Royal Stay Hotel!"
        )

        return invoice_details

    def email_invoice(self):
```

```
        # Send invoice via email
        # In a real system, this would use an email service
        guest = self._booking.get_guest()
        return f"Invoice #{self._invoice_id} emailed to {guest.get_name()} successfully"

    # String representation
    def __str__(self):
        return (f"Invoice ID: {self._invoice_id} - Booking: {self._booking.get_booking_id()} - "
                f"Amount: ${self._total_amount:.2f} - Date: {self._issue_date.strftime('%Y-%m-%d')}")
```

In [8]:
```
#Class 8:
class Feedback:
    # Constructor
    def __init__(self, feedback_id, guest, booking, rating, comments=""):
        # Private attributes
        self._feedback_id = feedback_id
        self._guest = guest # Aggregation with Guest
        self._booking = booking
        self._rating = rating
        self._comments = comments

#Similar to association at code level
#The difference is conceptual - aggregation suggests the guest is "part of" the feedback collection
#If the Guest object is deleted, the Feedback object still exists but loses access to guest information
#Often implemented with collections like lists (e.g., self._guests = []) when the relationship is one-to-many

    # Getter and setter methods
    def get_feedback_id(self):
        return self._feedback_id

    def get_guest(self):
        return self._guest

    def get_rating(self):
        return self._rating

    def set_rating(self, rating):
```

```
        self._rating = rating

    def get_comments(self):
        return self._comments

    def set_comments(self, comments):
        self._comments = comments

    # Methods
    def submit_feedback(self):
        # Submit the feedback
        return f"Feedback {self._feedback_id} submitted successfully with rating {self._rating}/5"

    def view_feedback(self):
        # Return a string representation of the feedback
        return (f"Feedback from {self._guest.get_name()} - Rating: {self._rating}/5\n"
                f"Comments: {self._comments}\n"
                f"For booking: {self._booking.get_booking_id()}")

    # String representation
    def __str__(self):
        return (f"Feedback ID: {self._feedback_id} - Guest: {self._guest.get_name()} - "
                f"Rating: {self._rating}/5 - Comments: {self._comments[:20]}...")
```

In [9]:
```python
from datetime import datetime, timedelta

#Part C:
# 1. Guest Account Creation:
def test_guest_account_creation():
    print("\n=== Testing Guest Account Creation ===")

    # Test Case 1: Creating a new guest account with valid information
    # Using assertions to verify object creation and property values
    # Assertions are good for validating expected conditions in unit tests
    try:
        guest1 = Guest(1001, "Asma Mohamed", "Asma.Mohamed@zu.ac.ae", 0)
        assert guest1 is not None, "Guest object should be created"
        assert guest1.get_guest_id() == 1001, "Guest ID should be 1001"
        assert guest1.get_name() == "Asma Mohamed", "Name should be Asma Mohamed"
        print(f"Test Case 1: {guest1}")
        print("Guest account created successfully!")
    except AssertionError as ae:
        print(f"Test Case 1 failed: {ae}")

    # Test Case 2: Creating a guest account and updating information
    # Using exception handling to catch potential errors in the update process
    # Exception handling is better for operations that might fail due to external factors
    try:
        guest2 = Guest(1002, "Mohamed Ali", "Mohamed Ali@gmail.com", 100)
        print(f"Test Case 2 (before update): {guest2}")

        # Update guest information
        result = guest2.update_profile("Mohamed Ali", "Mohamed.Ali@gmail.com")
        print(f"Update result: {result}")
        print(f"Test Case 2 (after update): {guest2}")

        # Assertion after update to verify changes took effect
        assert guest2.get_name() == "Mohamed Ali", "Name should be updated"
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [10]:
```python
# 2. Searching for Available Rooms:
def test_room_search():
    print("\n=== Testing Room Search Functionality ===")

    # Create rooms with different properties
    rooms = [
        Room(101, "Single", ["Wi-Fi", "TV"], 100.0),
        Room(102, "Double", ["Wi-Fi", "TV", "Mini-bar"], 150.0),
        Room(103, "Suite", ["Wi-Fi", "TV", "Mini-bar", "Jacuzzi"], 250.0),
        Room(104, "Single", ["Wi-Fi", "TV"], 100.0),
        Room(105, "Double", ["Wi-Fi", "TV", "Mini-bar"], 150.0)
    ]

    # Set some rooms as unavailable
    rooms[1].set_available(False)
    rooms[3].set_available(False)

    # Test Case 1: Search for available single rooms
    # Using assertions to verify the search results
    # Assertions are appropriate here to validate the filtering logic
    print("Test Case 1: Search for available single rooms")
    room_type = "Single"
    available_singles = [room for room in rooms if room.get_type() == room_type and room.is_available()]

    assert len(available_singles) == 1, "Should find exactly 1 available single room"
    assert available_singles[0].get_room_number() == 101, "Room 101 should be the available single"

    print(f"Found {len(available_singles)} available single rooms:")
    for room in available_singles:
        print(f"  {room}")

    # Test Case 2: Search for rooms with specific amenities
    # Using exception handling to protect against potential errors in the search process
    # This demonstrates handling unexpected conditions in the search filter
    try:
        print("\nTest Case 2: Search for rooms with Mini-bar")
        required_amenity = "Mini-bar"
        rooms_with_minibar = [room for room in rooms if required_amenity in room.get_amenities() and room.is_avai
```

```
        print(f"Found {len(rooms_with_minibar)} available rooms with mini-bar:")
        for room in rooms_with_minibar:
            print(f"  {room}")

        # Adding assertions to verify the search results
        assert len(rooms_with_minibar) == 2, "Should find 2 rooms with minibar"
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [11]:
```
#3. Making a Room Reservation:
def test_room_reservation():
    print("\n=== Testing Room Reservation Process ===")

    # Create test data
    guest = Guest(1001, "Asma Mohamed", "Asma.Mohamed@zu.ac.ae", 0)
    room = Room(101, "Single", ["Wi-Fi", "TV"], 100.0)
    check_in = datetime.now() + timedelta(days=1)
    check_out = datetime.now() + timedelta(days=3)

    # Test Case 1: Make a successful reservation
    # Using a combination of exception handling and assertions
    # Exception handling is used for the overall process, while assertions validate specific outcomes
    try:
        print("Test Case 1: Making a new reservation")
        booking = Booking(5001, guest, room, check_in, check_out)
        confirmation = booking.confirm_booking()

        print(f"Reservation details: {booking}")
        print(f"Confirmation: {confirmation}")

        # Assert that the room is now unavailable
        assert not room.is_available(), "Room should be marked as unavailable after booking"
        print(f"Room availability after booking: {'Available' if room.is_available() else 'Not Available'}")
    except AssertionError as ae:
        print(f"Test Case 1 failed: {ae}")
    except Exception as e:
```

```
        print(f"Test Case 1 failed: {e}")

    # Test Case 2: Try to book an unavailable room
    # Using exception handling since this is testing an error condition
    # Exception handling is better for testing error paths
    try:
        print("\nTest Case 2: Attempting to book an unavailable room")
        # Room is now unavailable from previous booking
        if not room.is_available():
            print("Room is not available for booking.")

            # Create another room for testing
            available_room = Room(102, "Double", ["Wi-Fi", "TV", "Mini-bar"], 150.0)
            new_booking = Booking(5002, guest, available_room, check_in, check_out)
            confirmation = new_booking.confirm_booking()

            print(f"Alternative booking made: {new_booking}")
            print(f"Confirmation: {confirmation}")

            # Assert that the new room is now unavailable
            assert not available_room.is_available(), "Alternative room should be unavailable after booking"
        else:
            print("Room is unexpectedly available!")
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [12]:

```
#4. Booking Confirmation Notification:
def test_booking_notification():
    print("\n=== Testing Booking Confirmation Notification ===")

    # Create test data
    guest = Guest(1001, "Asma Mohamed", "Asma.Mohamed@zu.ac.ae", 0)
    room = Room(101, "Single", ["Wi-Fi", "TV"], 100.0)
    check_in = datetime.now() + timedelta(days=1)
    check_out = datetime.now() + timedelta(days=3)

    # Test Case 1: Send booking confirmation email
```

```
    # Using assertions to verify the booking object is in the expected state
    # Assertions are appropriate for validating object properties
    try:
        print("Test Case 1: Sending email notification")
        booking = Booking(5001, guest, room, check_in, check_out)
        booking.confirm_booking()

        # Assert that the booking status is confirmed
        assert booking.get_status() == "Confirmed", "Booking status should be 'Confirmed'"

        # In a real system, this would send an actual email
        # Here we simulate the notification
        print(f"Email sent to {guest.get_contact_info()}")
        print(f"Subject: Royal Stay Hotel - Booking #{booking.get_booking_id()} Confirmation")
        print("Body: Thank you for your booking at Royal Stay Hotel...")
    except AssertionError as ae:
        print(f"Test Case 1 failed: {ae}")
    except Exception as e:
        print(f"Test Case 1 failed: {e}")

    # Test Case 2: Send booking confirmation as in-app notification
    # Using exception handling to catch potential errors in the notification process
    # Exception handling is better for external communication operations
    try:
        print("\nTest Case 2: Sending in-app notification")
        booking = Booking(5002, guest, room, check_in, check_out)
        booking.confirm_booking()

        # Simulate in-app notification
        print(f"In-app notification sent to user ID: {guest.get_guest_id()}")
        print(f"Notification: Your booking #{booking.get_booking_id()} is confirmed!")
        print(f"Details: {room.get_type()} room for {(check_out - check_in).days} nights")
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [13]:
```
#5. Invoice Generation for a Booking:
def test_invoice_generation():
```

```
print("\n=== Testing Invoice Generation ===")

# Create test data
guest = Guest(1001, "Asma Mohamed", "Asma.Mohamed@zu.ac.ae", 0)
room = Room(101, "Single", ["Wi-Fi", "TV"], 100.0)
check_in = datetime.now() + timedelta(days=1)
check_out = datetime.now() + timedelta(days=3)
booking = Booking(5001, guest, room, check_in, check_out)

# Test Case 1: Generate a simple invoice
# Using assertions to verify the invoice amount is calculated correctly
# Assertions are good for validating mathematical calculations
try:
    print("Test Case 1: Generating a basic invoice")
    total_amount = booking.calculate_total_price()
    # Assert that the total amount is correct (2 nights * 100.0 per night)
    assert total_amount == 200.0, f"Expected total amount of 200.0, got {total_amount}"

    invoice = Invoice(9001, booking, total_amount)

    print(f"Invoice created: {invoice}")
    print("\nInvoice Details:")
    print(invoice.generate_invoice())
except AssertionError as ae:
    print(f"Test Case 1 failed: {ae}")
except Exception as e:
    print(f"Test Case 1 failed: {e}")

# Test Case 2: Generate invoice with additional charges
# Using exception handling to catch potential errors in the invoice generation process
# Exception handling is better for complex processes with multiple potential failure points
try:
    print("\nTest Case 2: Invoice with additional charges")
    total_amount = booking.calculate_total_price()
    # Add additional charges (e.g., room service, minibar)
    additional_charges = 45.50
    total_with_charges = total_amount + additional_charges
```

```
        invoice = Invoice(9002, booking, total_with_charges)

        print(f"Invoice created: {invoice}")
        print("\nInvoice Details (with additional charges):")
        print(f"Room charges: ${total_amount:.2f}")
        print(f"Additional services: ${additional_charges:.2f}")
        print(f"Total amount: ${total_with_charges:.2f}")

        # Assert the total amount is calculated correctly
        assert abs(total_with_charges - 245.50) < 0.01, "Total amount should be 245.50"
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [14]:

```
#6. Processing Different Payment Methods:
def test_payment_processing():
    print("\n=== Testing Payment Processing ===")

    # Create test data
    guest = Guest(1001, "Asma Mohamed", "Asma.Mohamed@zu.ac.ae", 0)
    room = Room(101, "Single", ["Wi-Fi", "TV"], 100.0)
    check_in = datetime.now() + timedelta(days=1)
    check_out = datetime.now() + timedelta(days=3)
    booking = Booking(5001, guest, room, check_in, check_out)
    total_amount = booking.calculate_total_price()

    # Test Case 1: Process payment with credit card
    # Using assertions to verify the payment process and status
    # Assertions are appropriate for verifying the payment state
    try:
        print("Test Case 1: Processing credit card payment")
        payment = Payment(7001, booking, total_amount, "Credit Card")

        result = payment.process_payment()
        # Assert that the payment was successful
        assert result == True, "Payment processing should return True"

        print("Payment processed successfully!")
```

```
        print(payment.generate_receipt())
    except AssertionError as ae:
        print(f"Test Case 1 failed: {ae}")
    except Exception as e:
        print(f"Test Case 1 failed: {e}")

    # Test Case 2: Process payment with mobile wallet
    # Using exception handling to catch potential errors in the payment process
    # Exception handling is better for external payment processing operations
    try:
        print("\nTest Case 2: Processing mobile wallet payment")
        payment = Payment(7002, booking, total_amount, "Mobile Wallet")

        if payment.process_payment():
            print("Payment processed successfully!")
            print(payment.generate_receipt())
        else:
            print("Payment processing failed!")

        # Assert that the payment has the correct status
        assert payment.get_status() == "Completed", "Payment status should be 'Completed'"
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [15]:
```
#7. Displaying Reservation History:
def test_reservation_history():
    print("\n=== Testing Reservation History Display ===")

    # Create test data
    guest = Guest(1001, "Asma Mohamed", "Asma.Mohamedh@zu.ac.ae", 0)
    room1 = Room(101, "Single", ["Wi-Fi", "TV"], 100.0)
    room2 = Room(102, "Double", ["Wi-Fi", "TV", "Mini-bar"], 150.0)

    # Add some bookings with different dates
    check_in1 = datetime(2024, 2, 15)
    check_out1 = datetime(2024, 2, 17)
    booking1 = Booking(5001, guest, room1, check_in1, check_out1)
```

```
booking1.set_status("Completed")

check_in2 = datetime(2024, 3, 10)
check_out2 = datetime(2024, 3, 12)
booking2 = Booking(5002, guest, room2, check_in2, check_out2)
booking2.set_status("Completed")

# Future booking
check_in3 = datetime.now() + timedelta(days=30)
check_out3 = datetime.now() + timedelta(days=33)
booking3 = Booking(5003, guest, room1, check_in3, check_out3)
booking3.set_status("Confirmed")

# Test Case 1: View all bookings
# Using assertions to verify the booking collection
# Assertions are appropriate for validating collection properties
try:
    print("Test Case 1: Viewing all bookings")
    # In a real system, this would fetch from a database
    bookings = [booking1, booking2, booking3]

    # Assert that we have the correct number of bookings
    assert len(bookings) == 3, "Should have 3 bookings"

    print(f"Reservation history for {guest.get_name()}:")
    for booking in bookings:
        print(f"  {booking}")
except AssertionError as ae:
    print(f"Test Case 1 failed: {ae}")
except Exception as e:
    print(f"Test Case 1 failed: {e}")

# Test Case 2: Filter bookings by status
# Using a mix of exception handling and assertions
# Exception handling for the overall process, assertions for specific conditions
try:
    print("\nTest Case 2: Viewing completed bookings only")
    # Filter for completed bookings
```

```
        completed_bookings = [b for b in [booking1, booking2, booking3] if b.get_status() == "Completed"]

        # Assert that we have the correct number of completed bookings
        assert len(completed_bookings) == 2, "Should have 2 completed bookings"

        print(f"Completed stays for {guest.get_name()}:")
        for booking in completed_bookings:
            print(f"  {booking}")
            # Assert that each booking is actually completed
            assert booking.get_status() == "Completed", "Booking should have 'Completed' status"
    except AssertionError as ae:
        print(f"Test Case 2 failed: {ae}")
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [16]:
```
#8. Cancellation of a Reservation:
def test_reservation_cancellation():
    print("\n=== Testing Reservation Cancellation ===")

    # Create test data
    guest = Guest(1001, "Asma Mohamed", "Asma.Mohamed@zu.ac.ae", 0)
    room = Room(101, "Single", ["Wi-Fi", "TV"], 100.0)
    check_in = datetime.now() + timedelta(days=7)  # Booking for next week
    check_out = datetime.now() + timedelta(days=10)

    # Test Case 1: Standard cancellation
    # Using assertions to verify the cancellation effects
    # Assertions are good for validating state changes
    try:
        print("Test Case 1: Standard cancellation")
        booking = Booking(5001, guest, room, check_in, check_out)
        booking.confirm_booking()

        print(f"Initial booking status: {booking.get_status()}")
        print(f"Room availability before cancellation: {'Available' if room.is_available() else 'Not Available'}"

        # Process cancellation
```

```
        result = booking.cancel_booking()

        print(result)
        print(f"Booking status after cancellation: {booking.get_status()}")

        # Assert that the booking status is now cancelled
        assert booking.get_status() == "Cancelled", "Booking status should be 'Cancelled'"
        # Assert that the room is now available
        assert room.is_available(), "Room should be available after cancellation"

        print(f"Room availability after cancellation: {'Available' if room.is_available() else 'Not Available'}")
    except AssertionError as ae:
        print(f"Test Case 1 failed: {ae}")
    except Exception as e:
        print(f"Test Case 1 failed: {e}")

    # Test Case 2: Cancellation with refund
    # Using exception handling to catch potential errors in the refund process
    # Exception handling is better for financial operations with multiple potential failure points
    try:
        print("\nTest Case 2: Cancellation with refund processing")
        booking = Booking(5002, guest, room, check_in, check_out)
        booking.confirm_booking()

        # Create payment for this booking
        total_amount = booking.calculate_total_price()
        payment = Payment(7001, booking, total_amount, "Credit Card")
        payment.process_payment()

        print(f"Payment made: {payment}")

        # Process cancellation and refund
        result = booking.cancel_booking()
        print(result)

        refund_result = payment.refund_payment()
        if refund_result:
            print(f"Refund processed for payment {payment.get_payment_id()}")
```

```
            print(f"Refund amount: ${payment.get_amount():.2f}")

            # Assert that the payment status is now refunded
            assert payment.get_status() == "Refunded", "Payment status should be 'Refunded'"
        else:
            print("Refund processing failed!")
    except Exception as e:
        print(f"Test Case 2 failed: {e}")
```

In [17]:
```
# printing each of part c alone to make it more clearer.
# Run all test cases
if __name__ == "__main__":
    print("===== ROYAL STAY HOTEL MANAGEMENT SYSTEM - TEST CASES =====\n")

    test_guest_account_creation()
```

```
===== ROYAL STAY HOTEL MANAGEMENT SYSTEM - TEST CASES =====


=== Testing Guest Account Creation ===
Test Case 1: Guest ID: 1001 - Name: Asma Mohamed - Contact: Asma.Mohamed@zu.ac.ae - Loyalty Points: 0
Guest account created successfully!
Test Case 2 (before update): Guest ID: 1002 - Name: Mohamed Ali - Contact: Mohamed Ali@gmail.com - Loyalty Point
s: 100
Update result: Profile updated successfully
Test Case 2 (after update): Guest ID: 1002 - Name: Mohamed Ali - Contact: Mohamed.Ali@gmail.com - Loyalty Point
s: 100
```

In [18]:
```
    test_room_search()
```

```
=== Testing Room Search Functionality ===
Test Case 1: Search for available single rooms
Found 1 available single rooms:
  Room 101 - Type: Single - Price: $100.00/night - Status: Available

Test Case 2: Search for rooms with Mini-bar
Found 2 available rooms with mini-bar:
  Room 103 - Type: Suite - Price: $250.00/night - Status: Available
  Room 105 - Type: Double - Price: $150.00/night - Status: Available
```

In [19]:
```
test_room_reservation()
```

```
=== Testing Room Reservation Process ===
Test Case 1: Making a new reservation
Reservation details: Booking ID: 5001 - Guest: Asma Mohamed - Room: 101 - Check-in: 2025-03-19 - Check-out: 2025
-03-21 - Status: Confirmed
Confirmation: Booking 5001 confirmed for Asma Mohamed
Room availability after booking: Not Available

Test Case 2: Attempting to book an unavailable room
Room is not available for booking.
Alternative booking made: Booking ID: 5002 - Guest: Asma Mohamed - Room: 102 - Check-in: 2025-03-19 - Check-out:
2025-03-21 - Status: Confirmed
Confirmation: Booking 5002 confirmed for Asma Mohamed
```

In [20]:
```
test_booking_notification()
```

```
=== Testing Booking Confirmation Notification ===
Test Case 1: Sending email notification
Email sent to Asma.Mohamed@zu.ac.ae
Subject: Royal Stay Hotel - Booking #5001 Confirmation
Body: Thank you for your booking at Royal Stay Hotel...

Test Case 2: Sending in-app notification
In-app notification sent to user ID: 1001
Notification: Your booking #5002 is confirmed!
Details: Single room for 2 nights
```

In [21]:
```
test_invoice_generation()
```

```
=== Testing Invoice Generation ===
Test Case 1: Generating a basic invoice
Invoice created: Invoice ID: 9001 - Booking: 5001 - Amount: $200.00 - Date: 2025-03-18

Invoice Details:
INVOICE #9001
Date: 2025-03-18
Guest: Asma Mohamed
Room: 101 (Single)
Check-in: 2025-03-19
Check-out: 2025-03-21
Total Amount: $200.00
Thank you for choosing Royal Stay Hotel!

Test Case 2: Invoice with additional charges
Invoice created: Invoice ID: 9002 - Booking: 5001 - Amount: $245.50 - Date: 2025-03-18

Invoice Details (with additional charges):
Room charges: $200.00
Additional services: $45.50
Total amount: $245.50
```

In [22]:
```
test_payment_processing()
```

```
=== Testing Payment Processing ===
Test Case 1: Processing credit card payment
Payment processed successfully!
Receipt for Payment #7001
Date: 2025-03-18 21:23:08
Amount: $200.00
Method: Credit Card
Booking ID: 5001
Guest: Asma Mohamed

Test Case 2: Processing mobile wallet payment
Payment processed successfully!
Receipt for Payment #7002
Date: 2025-03-18 21:23:08
Amount: $200.00
Method: Mobile Wallet
Booking ID: 5001
Guest: Asma Mohamed
Test Case 2 failed: 'Payment' object has no attribute 'get_status'
```

In [23]:
```
test_reservation_history()
```

```
=== Testing Reservation History Display ===
Test Case 1: Viewing all bookings
Reservation history for Asma Mohamed:
  Booking ID: 5001 - Guest: Asma Mohamed - Room: 101 - Check-in: 2024-02-15 - Check-out: 2024-02-17 - Status: Co
mpleted
  Booking ID: 5002 - Guest: Asma Mohamed - Room: 102 - Check-in: 2024-03-10 - Check-out: 2024-03-12 - Status: Co
mpleted
  Booking ID: 5003 - Guest: Asma Mohamed - Room: 101 - Check-in: 2025-04-17 - Check-out: 2025-04-20 - Status: Co
nfirmed

Test Case 2: Viewing completed bookings only
Completed stays for Asma Mohamed:
  Booking ID: 5001 - Guest: Asma Mohamed - Room: 101 - Check-in: 2024-02-15 - Check-out: 2024-02-17 - Status: Co
mpleted
  Booking ID: 5002 - Guest: Asma Mohamed - Room: 102 - Check-in: 2024-03-10 - Check-out: 2024-03-12 - Status: Co
mpleted
```

In [24]:
```
    test_reservation_cancellation()

    print("\n===== ALL TESTS COMPLETED =====")
```

```
=== Testing Reservation Cancellation ===
Test Case 1: Standard cancellation
Initial booking status: Confirmed
Room availability before cancellation: Not Available
Booking 5001 cancelled for Asma Mohamed
Booking status after cancellation: Cancelled
Room availability after cancellation: Available

Test Case 2: Cancellation with refund processing
Payment made: Payment ID: 7001 - Amount: $300.00 - Date: 2025-03-18 - Method: Credit Card
Booking 5002 cancelled for Asma Mohamed
Refund processed for payment 7001
Refund amount: $300.00
Test Case 2 failed: 'Payment' object has no attribute 'get_status'

===== ALL TESTS COMPLETED =====
```

In [ ]:
```

```