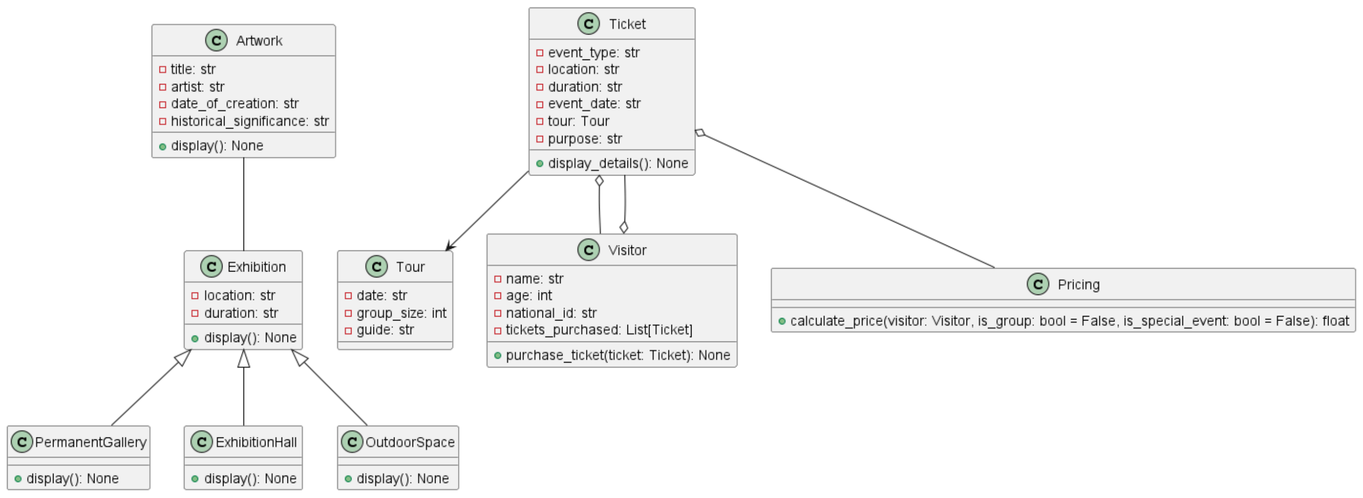
The Louvre Museum

UML Diagram for the Program



In this diagram:

- Artwork, Exhibition, PermanentGallery, ExhibitionHall, and OutdoorSpace represent the different aspects of the museum's collection and exhibition spaces.
- Visitor represents individuals who visit the museum and purchase tickets.
- Ticket represents the tickets purchased by visitors, which can be for different types of events or exhibitions.
- Pricing is a class responsible for calculating the price of tickets based on visitor categories and event types.
- There are associations between Artwork and Exhibition to show that artworks can be displayed in exhibitions, and exhibitions can be held in different locations such as permanent galleries, exhibition halls, or outdoor spaces.
- Exhibition has different types of exhibitions represented by PermanentGallery, ExhibitionHall, and OutdoorSpace through inheritance to represent the different types of exhibition spaces within the museum.
- There is an association between Visitor and Ticket to show that visitors can purchase tickets.
- There is a composition relationship between Ticket and Pricing to represent that the pricing logic is encapsulated within the ticket object.

Python Code for Classes

Artwork

```python
class Artwork:
    def __init__(self, title: str, artist: str, date_of_creation: str,
historical_significance: str):
        self.title = title
        self.artist = artist
        self.date_of_creation = date_of_creation
        self.historical_significance = historical_significance

    def display(self) -> None:
        print(f"Title: {self.title}, Artist: {self.artist}, Date of Creation:
{self.date_of_creation}, Historical Significance: {self.historical_significance}")
```

Exhibition

```python
class Exhibition:
    def __init__(self, location: str, duration: str):
        self.location = location
        self.duration = duration

    def display(self) -> None:
        print(f"Location: {self.location}, Duration: {self.duration}")

class PermanentGallery(Exhibition):
    def __init__(self, location: str, duration: str):
        super().__init__(location, duration)

class ExhibitionHall(Exhibition):
    def __init__(self, location: str, duration: str):
        super().__init__(location, duration)

class OutdoorSpace(Exhibition):
    def __init__(self, location: str, duration: str):
        super().__init__(location, duration)
```

Tour

```
class Tour:
    def __init__(self, date: str, group_size: int, guide: str):
        self.date = date
        self.group_size = group_size
        self.guide = guide
```

Ticket

```
from tour import Tour
from typing import Union

class Ticket:
    def __init__(self, event_type: str, location: str, duration: str, event_date:
str, tour: Union[None, Tour] = None, purpose: Union[None, str] = None):
        self.event_type = event_type
        self.location = location
        self.duration = duration
        self.event_date = event_date
        self.tour = tour
        self.purpose = purpose

    def display_details(self) -> None:
        print("Ticket Details:")
        print(f"Event Type: {self.event_type}")
        print(f"Location: {self.location}")
        print(f"Duration: {self.duration}")
        print(f"Event Date: {self.event_date}")
        if self.event_type == 'Tour':
            print(f"Tour Date: {self.tour.date}")
            print(f"Group Size: {self.tour.group_size}")
            print(f"Guide: {self.tour.guide}")
        elif self.event_type == 'Special Event':
            print(f"Purpose: {self.purpose}")
```

Visitor

```python
from typing import List
from ticket import Ticket

class Visitor:
    def __init__(self, name: str, age: int, national_id: str):
        self.name = name
        self.age = age
        self.national_id = national_id
        self.tickets_purchased: List[Ticket] = []

    def purchase_ticket(self, ticket: Ticket) -> None:
        self.tickets_purchased.append(ticket)

    def display(self) -> None:
        print("Visitor Details:")
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"National ID: {self.national_id}")
        print("Tickets Purchased:", self.tickets_purchased)
```

Pricing

```python
from typing import Union
from visitor import Visitor

class Pricing:
    def calculate_price(self, visitor: Visitor, is_group: bool = False,
is_special_event: bool = False) -> float:
        if visitor.age < 18 or visitor.age > 60:
            return 0.0  # Free ticket for children, seniors, and certain groups

        base_price = 63.0  # Base price for adults

        if is_group:
            base_price *= 0.5  # Apply 50% discount for groups

        if is_special_event:
            # Additional logic for special event pricing
            pass

        final_price = base_price * (1 + 0.05)  # Adding 5% VAT

        return final_price
```

Test Cases

```
from artwork import Artwork
from exhibition import PermanentGallery, ExhibitionHall, OutdoorSpace
from visitor import Visitor
from ticket import Ticket
from tour import Tour
from pricing import Pricing

def test_add_new_art():
    artwork1 = Artwork("Mona Lisa", "Leonardo da Vinci", "1503", "Iconic portrait")
    artwork2 = Artwork("The Starry Night", "Vincent van Gogh", "1889", "Post-Impressionist masterpiece")

    artwork1.display()
    artwork2.display()
    # Add more artwork objects as needed

def test_open_new_exhibition():
    permanent_gallery = PermanentGallery("Gallery 1", "Permanent")
    exhibition_hall = ExhibitionHall("Hall A", "2 months")
    outdoor_space = OutdoorSpace("Garden", "3 weeks")

    permanent_gallery.display()
    exhibition_hall.display()
    outdoor_space.display()
```

```python
def test_purchase_tickets():
    visitor1 = Visitor("Alice", 25, "12345")
    visitor2 = Visitor("Bob", 55, "67890")
    tour_guide = "John"
    tour_date = "2024-04-15"
    tour_group_size = 20
    tour = Tour(tour_date, tour_group_size, tour_guide)
    tour_ticket = Ticket("Tour", "Louvre Museum", "2 hours", "2024-04-15",
tour=tour)

    special_event_ticket = Ticket("Special Event", "Concert Hall", "3 hours",
"2024-05-01", purpose="Musical Concert")

    pricing = Pricing()
    tour_ticket_price = pricing.calculate_price(visitor1)
    special_event_ticket_price = pricing.calculate_price(visitor2,
is_special_event=True)

    visitor1.purchase_ticket(tour_ticket)
    visitor2.purchase_ticket(special_event_ticket)


    print(f"{visitor1.name} purchased a tour ticket for {tour_ticket_price} AED")
    print(f"{visitor2.name} purchased a special event ticket for
{special_event_ticket_price} AED")
```

```python
def test_display_payment_receipts():
    visitor1 = Visitor("Alice", 25, "12345")
    visitor2 = Visitor("Bob", 55, "67890")
    tour_guide = "John"
    tour_date = "2024-04-15"
    tour_group_size = 20
    tour = Tour(tour_date, tour_group_size, tour_guide)
    tour_ticket = Ticket("Tour", "Louvre Museum", "2 hours", "2024-04-15",
tour=tour)


    special_event_ticket = Ticket("Special Event", "Concert Hall", "3 hours",
"2024-05-01", purpose="Musical Concert")

    pricing = Pricing()
    tour_ticket_price = pricing.calculate_price(visitor1)
    special_event_ticket_price = pricing.calculate_price(visitor2,
is_special_event=True)

    visitor1.purchase_ticket(tour_ticket)
    visitor2.purchase_ticket(special_event_ticket)

    total_price = tour_ticket_price + special_event_ticket_price
    print(f"Total price for both tickets: {total_price} AED")

# Run test cases
test_add_new_art()
test_open_new_exhibition()
test_purchase_tickets()
test_display_payment_receipts()
```

Summary

The implementation of the Louvre Museum software application provides a comprehensive system for managing artworks, exhibitions, visitors, tickets, and pricing. Here's a summary of the key learnings from this exercise:

1. **Modular Design**: The program is designed with modularity in mind, with each class representing a specific aspect of the museum management system. Separating classes into different files enhances readability and maintainability.

2. **Class Relationships**: The relationships between classes are carefully defined to model real-world interactions effectively. For example, the composition relationship between `Ticket` and `Pricing` encapsulates pricing logic within ticket objects.

3. **Inheritance and Polymorphism**: The use of inheritance allows for code reuse and polymorphism, as demonstrated by the `Exhibition` superclass and its subclasses representing different exhibition spaces.

4. **Type Annotations**: Type annotations are used extensively throughout the code to improve code clarity and facilitate type checking.

5. **Test Cases**: Test cases are defined to showcase various program features, including adding new artwork, opening exhibitions, purchasing tickets, and displaying payment receipts. These test cases ensure that the program functions as expected and can be easily verified and validated.

Overall, this exercise demonstrates the application of object-oriented principles, modular design, and test-driven development techniques to create a robust and flexible software application for managing museum operations.