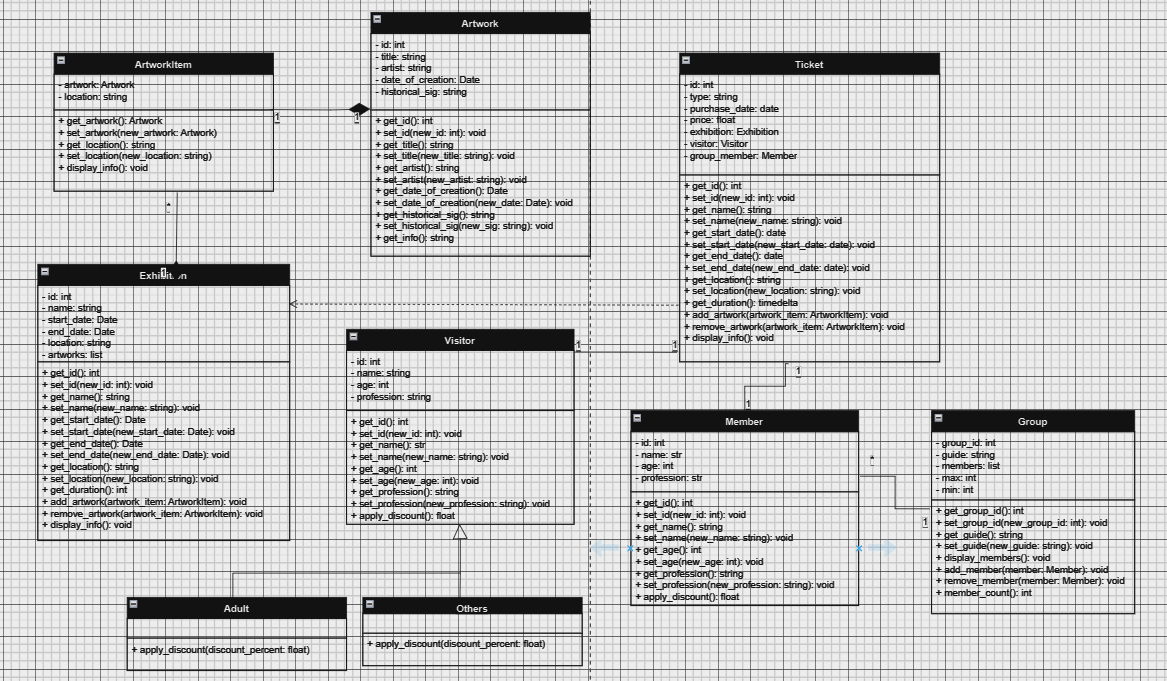
**Software Implementation - OOP Project**

**Report**

**Student Name:  
Student ID:**

**Submission Date:**

**UML Class Diagram**

****

There are nine classes in the diagram: Artwork, ArtworkItem, Ticket, Exhibition, Visitor, Member, Group, Adult, and Others. Artwork contains attributes like title, artist, date of creation, and historical significance and access methods for them. Artwork contains an instance of Artwork and stores the location of the artwork. In addition, to the setters and getters, the class contains a display function to display the art information of both classes. The Artwork and ArtworkItam classes do a single purpose of storing artwork and artifacts for the museum. It means that if the Artwork class is deleted ArtworkItem serves no purpose at all.

Then comes the Exhibition class which is where the artworks from the museum are displayed. An exhibition can be permanent galleries, exhibition halls, and outdoor spaces. It encapsulates essential attributes such as an ID for identification, a name to describe the exhibition, start and end dates to define its duration, a location where it takes place, and a list of artworks included in the exhibition. This class enables the management and organization of exhibitions, allowing for the addition and removal of artworks, as well as providing functionality to display information about the exhibition, such as its ID, name, dates, location, and the artworks featured in it. ArtworkItem and Exhibition have an association relation. 1 artwork item can be displayed in 1 exhibition whereas 1 exhibition can have multiple artwork items.

An exhibition or event ticket for entry into a museum or other cultural facility is represented by the Ticket class. A ticket's qualities include identity information such as an ID, the type of ticket (e.g. online, in-person), the price, the date of purchase, information about the exhibition or event for which the ticket is intended, the visitor or group member who is purchasing the ticket, and the group member, if applicable. The course covers techniques for verifying the ticket using the date of the exhibition, computing the ticket cost with any relevant savings, and presenting ticket facts such as ID, purchase date, cost, related guest or group member, and exhibition or event specifics. The connection between the Exhibition and Ticket classes is that a ticket is linked to an exhibition and serves as a pass for entry or access to the event or exhibition within a museum or other cultural institution. Specifically, the Ticket is dependent on the Exhibition class as indicated by the arrow in the UML diagram.

The Visitor class represents a generic visitor to a museum or cultural institution and includes attributes such as ID, name, age, and profession. The Others and Adult classes are subclasses of Visitor and inherit from it. Each of the Others and Adult classes has a specialized discount logic for ticket pricing based on age and profession. The Others class provides free tickets for visitors under 18 or over 60 years old, as well as for students and teachers. In contrast, the Adult class applies no discounts for visitors aged 18 to 60. The relation between the Visitor and Ticket classes is that a ticket can be associated with a visitor (1 to 1 relation) when purchased, indicating which visitor or group member has bought the ticket and allowing for specific discount calculations based on the visitor's age and profession or group membership.

A group of visitors in a museum is represented by the `Group` class, which has members, a list of members, and a group ID. A single visitor within a group is represented by the Member class, which has attributes like ID, name, age, and occupation. A minimum and maximum member limit is upheld by the Group class, which controls member addition and removal. 1 member has 1 group and 1 group has many members. The Member and Ticket classes are related in that when a ticket is purchased in a group setting, it can be linked to a member, identifying which group member purchased the ticket. These two classes have an association relation. 1 member has 1 ticket and vice versa.

**Python Code**

Following are the Python classes implementation of the above UML diagram:

1. **Artwork**

class Artwork:

def \_\_init\_\_(self, id, title, artist, date\_of\_creation, historical\_sig):

self.\_\_id = id

self.\_\_title = title

self.\_\_artist = artist

self.\_\_date\_of\_creation = date\_of\_creation

self.\_\_historical\_sig = historical\_sig

def get\_id(self):

return self.\_\_id

def set\_id(self, new\_id):

self.\_\_id = new\_id

def get\_title(self):

return self.\_\_title

def set\_title(self, new\_title):

self.\_\_title = new\_title

def get\_artist(self):

return self.\_\_artist

def set\_artist(self, new\_artist):

self.\_\_artist = new\_artist

def get\_date\_of\_creation(self):

return self.\_\_date\_of\_creation

def set\_date\_of\_creation(self, new\_date):

self.\_\_date\_of\_creation = new\_date

def get\_historical\_sig(self):

return self.\_\_historical\_sig

def set\_historical\_sig(self, new\_sig):

self.\_\_historical\_sig = new\_sig

def get\_info(self):

return f"ID: {self.\_\_id}\nTitle: {self.\_\_title}\nArtist: {self.\_\_artist}\nDate of Creation: {self.\_\_date\_of\_creation}\nHistorical Significance: {self.\_\_historical\_sig}"

1. **ArtworkItem**

class ArtworkItem:

def \_\_init\_\_(self, artwork, location):

self.\_\_artwork = artwork

self.\_\_location = location

def get\_artwork(self):

return self.\_\_artwork

def set\_artwork(self, new\_artwork):

self.\_\_artwork = new\_artwork

def get\_location(self):

return self.\_\_location

def set\_location(self, new\_location):

self.\_\_location = new\_location

def display\_info(self):

print("--Artwork--")

print(self.\_\_artwork.get\_info())

print(f"Location: {self.\_\_location}")

1. **Exhibition**

class Exhibition:

def \_\_init\_\_(self, id, name, start\_date, end\_date, location):

self.\_\_id = id

self.\_\_name = name

self.\_\_start\_date = start\_date

self.\_\_end\_date = end\_date

self.\_\_location = location

self.\_\_artworks = [] #creating a list to store the diffreent artworks in an exhibition

def get\_id(self):

return self.\_\_id

def set\_id(self, new\_id):

self.\_\_id = new\_id

def get\_name(self):

return self.\_\_name

def set\_name(self, new\_name):

self.\_\_name = new\_name

def get\_start\_date(self):

return self.\_\_start\_date

def set\_start\_date(self, new\_start\_date):

self.\_\_start\_date = new\_start\_date

def get\_end\_date(self):

return self.\_\_end\_date

def set\_end\_date(self, new\_end\_date):

self.\_\_end\_date = new\_end\_date

def get\_location(self):

return self.\_\_location

def set\_location(self, new\_location):

self.\_\_location = new\_location

def get\_duration(self):

return self.\_\_end\_date - self.\_\_start\_date

def add\_artwork(self, artwork\_item):

self.\_\_artworks.append(artwork\_item) #adding artwork in list

def remove\_artwork(self, artwork\_item):

self.\_\_artworks.remove(artwork\_item) #removing artwork from list

def display\_info(self):

print("--Exhibition--")

print("ID: ", self.\_\_id)

print("Name: ", self.\_\_name)

print("Start Date: ", self.\_\_start\_date)

print("End Date: ", self.\_\_end\_date)

print("Location: ", self.\_\_location)

print("Artworks in Exhibition: ", end=" ")

for artwork\_item in self.\_\_artworks:

print(artwork\_item.get\_artwork().get\_title(), end=" ")

print()

This class contains a get\_duration() function that returns the subtracted result from the end date and start date of the exhibition. The class contains a list that stores all artworks and contains an add and remove artwork function as well.

1. **Visitor**

class Visitor:

def \_\_init\_\_(self, id, name, age, profession):

self.\_\_id = id

self.\_\_name = name

self.\_\_age = age

self.\_\_profession = profession

def get\_id(self):

return self.\_\_id

def set\_id(self, new\_id):

self.\_\_id = new\_id

def get\_name(self):

return self.\_\_name

def set\_name(self, new\_name):

self.\_\_name = new\_name

def get\_age(self):

return self.\_\_age

def set\_age(self, new\_age):

self.\_\_age = new\_age

def get\_profession(self):

return self.\_\_profession

def set\_profession(self, new\_profession):

self.\_\_profession = new\_profession

def apply\_discount(self):

return 0

1. **Adult**

class Adult(Visitor):

def apply\_discount(self):

if 18 <= self.get\_age() <= 60:

return 0 #no discount

else:

return -1 #invalid age for adult

1. **Others**

class Others(Visitor):

def apply\_discount(self):

if self.get\_age() <18 or self.get\_age() >60 or self.get\_profession() == "student" or self.get\_profession() == "teacher":

return 1 #free ticket

else:

return 0 #no discount

The Others and Afult classes inherit from Visitor. They are created to handle the discounting process for these 2 groups. Adults between 18 and 60 are supposed to pay the full ticket price of 63 AED whereas visitors under 18 (young people) or over 60 (seniors) years old and students and teachers are all given a 100 percent discount, which means the final ticket price for them is zero.

1. **Group**

class Group:

def \_\_init\_\_(self, group\_id, guide):

self.\_\_group\_id = group\_id

self.\_\_guide = guide

self.\_\_members = [] #members in a group

if len(self.\_\_members) < 15:

print()

print("--Error!!: Cannot create group with less than 15 members--")

print()

self.\_\_max = 40

self.\_\_min = 15

def get\_group\_id(self):

return self.\_\_group\_id

def set\_group\_id(self, new\_group\_id):

self.\_\_group\_id = new\_group\_id

def get\_guide(self):

return self.\_\_guide

def set\_guide(self, new\_guide):

self.\_\_guide = new\_guide

def display\_members(self):

print("Group Members: " , end=" ")

for i in self.\_\_members:

print(i.get\_name(), end=" ")

print()

def add\_member(self, member):

if self.member\_count() <= self.\_\_max:

self.\_\_members.append(member)

def remove\_member(self, member):

if member in self.\_\_members and self.member\_count() >= self.\_\_min:

self.\_\_members.remove(member)

else:

print(f"{member} is not a member of this group.")

def member\_count(self):

return len(self.\_\_members) #count of mmebers

The Group class has a maximum and minimum member limit as well. The add\_member() and remove\_member() are catered to deal with this. Then is the member\_count() function that calculates the count of group members. A guide is also used as the leader of the group. Moreover, a display message is presented if a group is created for members less than forty. The message goes away as soon as the member count exceeds 15.

1. **Member**

class Member:

def \_\_init\_\_(self, id, name, age, profession):

self.\_\_id = id

self.\_\_name = name

self.\_\_age = age

self.\_\_profession = profession

def get\_id(self):

return self.\_\_id

def set\_id(self, new\_id):

self.\_\_id = new\_id

def get\_name(self):

return self.\_\_name

def set\_name(self, new\_name):

self.\_\_name = new\_name

def get\_age(self):

return self.\_\_age

def set\_age(self, new\_age):

self.\_\_age = new\_age

def get\_profession(self):

return self.\_\_profession

def set\_profession(self, new\_profession):

self.\_\_profession = new\_profession

def apply\_discount(self):

return 0.5 #50% discount

Each member of a group is given a 50 percent.

1. **Ticket**

class Ticket:

def \_\_init\_\_(self, id, type, purchase\_date, price, exhibition, visitor, group\_member):

self.\_\_id = id

self.\_\_type = type

self.\_\_purchase\_date = purchase\_date

self.\_\_price = price

self.\_\_exhibition = exhibition

self.\_\_visitor = visitor

self.\_\_group\_member = group\_member

def get\_id(self):

return self.\_\_id

def set\_id(self, new\_id):

self.\_\_id = new\_id

def get\_type(self):

return self.\_\_type

def set\_type(self, new\_type):

self.\_\_type = new\_type

def get\_purchase\_date(self):

return self.\_\_purchase\_date

def set\_purchase\_date(self, new\_purchase\_date):

self.\_\_purchase\_date = new\_purchase\_date

def get\_price(self):

return self.\_\_price

def set\_price(self, new\_price):

self.\_\_price = new\_price

def get\_exhibition(self):

return self.\_\_exhibition

def set\_exhibition(self, new\_exhibition):

self.\_\_exhibition = new\_exhibition

def get\_visitor(self):

return self.\_\_visitor

def set\_visitor(self, new\_visitor):

self.\_\_visitor = new\_visitor

def get\_group\_member(self):

return self.\_\_group\_member

def set\_group\_member(self, new\_group\_member):

self.\_\_group\_member = new\_group\_member

def validate\_ticket(self):

current = date.today()

if current >= self.\_\_exhibition.get\_start\_date() and current <= self.\_\_exhibition.get\_end\_date():

return True

else:

return False

def calculate\_price(self):

if self.\_\_visitor != None:

discount\_percent = self.\_\_visitor.apply\_discount()

else:

discount\_percent = self.\_\_group\_member.apply\_discount()

if discount\_percent == -1:

return None

elif isinstance(self.\_\_group\_member, Member): #group member

discount\_percent = self.\_\_group\_member.apply\_discount()

elif isinstance(self.\_\_visitor, Visitor): #visitor

discount\_percent = self.\_\_visitor.apply\_discount()

discounted\_price = self.\_\_price - (self.\_\_price \* discount\_percent)

final\_price = discounted\_price \* 1.05 # 5% VAT

return final\_price

def display\_info(self):

print("--Ticket--")

print("ID: ",self.\_\_id)

print("Purchase Date: ", self.\_\_purchase\_date)

print("Price: ", self.\_\_price)

if self.\_\_visitor != None:

print("Visitor Name: ",self.\_\_visitor.get\_name())

print("Visitor ID: ", self.\_\_visitor.get\_id())

if self.\_\_group\_member != None:

print("Group Member Name: ", self.\_\_group\_member.get\_name())

print("Group Member ID: ", self.\_\_group\_member.get\_id())

print("Exhibition Location: ", self.\_\_exhibition.get\_location()) #exhibition location

print("Exhibition Name: ", self.\_\_exhibition.get\_name()) #name of exhibition

print("Exhibition Duration: ", self.\_\_exhibition.get\_duration()) #time of exhibition

By comparing the current date with the beginning and ending dates of the associated exhibition, the validate\_ticket() method determines whether the ticket is still valid. If the ticket is legitimate, it returns True; if not, it returns False. The final ticket price is determined by the calculate\_price() method, which takes into account any applicable discounts based on the visitor or group member. It initially determines whether the ticket is linked to a visitor; if not, it applies the group member's discount. Calling the visitor's or group member's apply\_discount() method yields the discount. It computes the discounted price and adds 5% VAT to get the final price if the discount is valid (a value of -1 indicates no discount).

**Test Cases**

**1.**

**############(a) addition of new art############**

**new\_artwork = Artwork(1, "Starry Night", "Vincent van Gogh", date(1889, 1, 1), "Post-Impressionist masterpiece")**

**new\_artwork\_item = ArtworkItem(new\_artwork, "Gallery B")**

**new\_artwork\_item.display\_info()**

**print()**

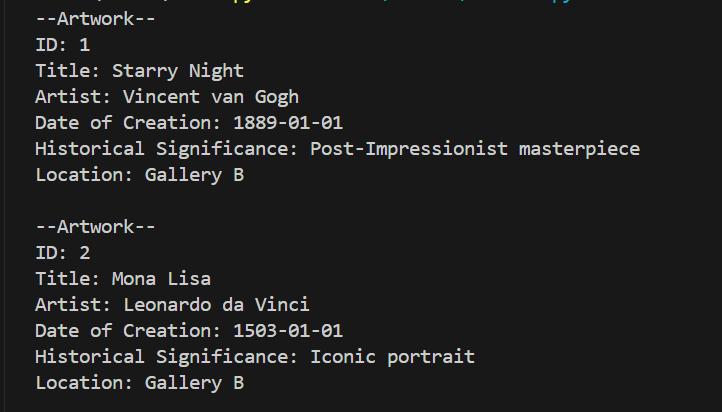
**new\_artwork1 = Artwork(2, "Mona Lisa", "Leonardo da Vinci", date(1503, 1, 1), "Iconic portrait")**

**new\_artwork\_item1 = ArtworkItem(new\_artwork1, "Gallery B")**

**new\_artwork\_item1.display\_info()**

**print()**

**Result**

****

**2.**

**############(b) opening a new museum exhibition############**

**new\_exhibition = Exhibition(1, "Impressionist Masterpieces", date(2024, 7, 1), date(2024, 9, 30), "East Wing")**

**#adding artworks to exhibition**

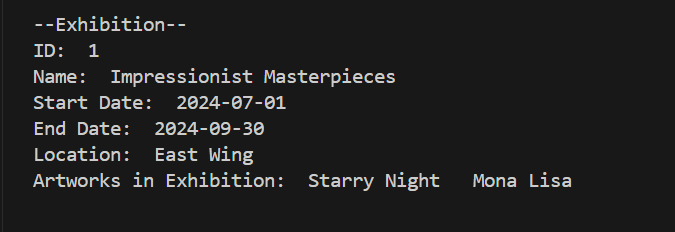
**new\_exhibition.add\_artwork(new\_artwork\_item)**

**new\_exhibition.add\_artwork(new\_artwork\_item1)**

**new\_exhibition.display\_info()**

**print()**

**Result**

****

**3.**

**############(c) purchase of ticket############**

**#adult purchases ticket**

**adult\_visitor = Adult(1, "Alice Johnson", 25, "engineer")**

**adult\_ticket = Ticket(1, "Inperson", date.today(), 63, new\_exhibition, adult\_visitor, None)**

**#student purchases ticket**

**student\_visitor = Others(1, "Adam Newman", 20, "student")**

**student\_ticket = Ticket(2, "Online", date.today(), 63, new\_exhibition, student\_visitor, None)**

**#group purchases ticket**

**group1 = Group(1, "tour guide")**

**member1 = Member(1, "Joden Dick", 30, "content creator")**

**member2 = Member(2, "Jane Smith", 25, "doctor")**

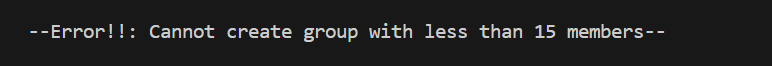
**group1.add\_member(member1)**

**group1.add\_member(member2)**

**member\_ticket1 = Ticket(3, "Inperson", date.today(), 63, new\_exhibition, None, member1)**

**member\_ticket2 = Ticket(4, "Online", date.today(), 63, new\_exhibition, None, member2)**

**Result**

****

The error appears because as of now the group members are lower than 15, the min limit.

**4.**

**############(d) ticket reciepts + final price############**

**adult\_ticket.display\_info()**

**print("ticket final price: ",adult\_ticket.calculate\_price())**

**print()**

**student\_ticket.display\_info()**

**print("ticket final price: ",student\_ticket.calculate\_price())**

**print()**

**member\_ticket1.display\_info()**

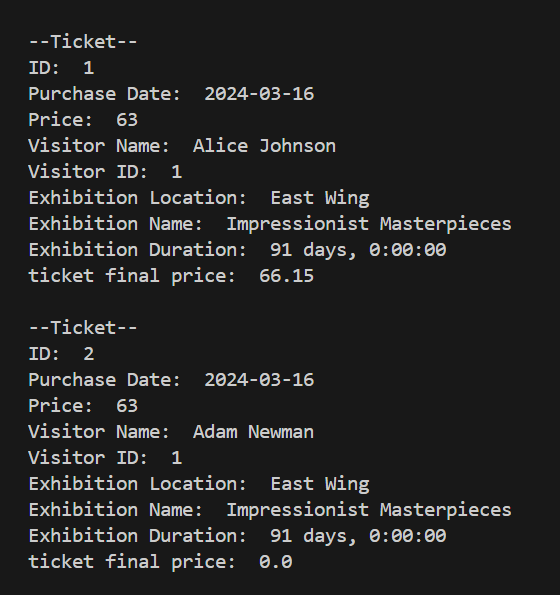
**print("ticket final price: ",member\_ticket1.calculate\_price())**

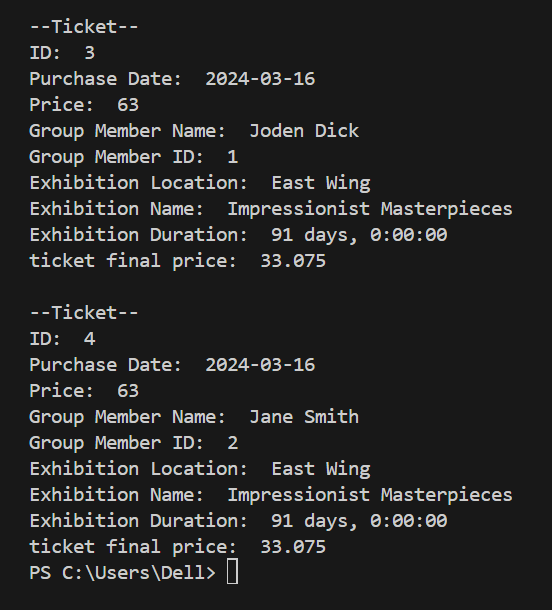
**print()**

**member\_ticket2.display\_info()**

**print("ticket final price: ",member\_ticket2.calculate\_price())**

**Result**

****

****

**Github Repository Link**

<https://github.com/beg-python/artwork>

**Summary of Learnings**

Through this project, I have gained valuable insights into object-oriented programming (OOP) principles and software design practices. One of the key learnings was the importance of encapsulation in OOP. Encapsulation allowed me to hide the internal details of classes, such as attributes and methods, and expose only the necessary functionalities through well-defined interfaces. I learned that this not only enhances the modularity of the code but also promotes code reusability.

Making the UML diagram also made it easier for me to see how various classes relate to one another and to comprehend how data moves through the system and interacts. In order to properly simulate real-world entities, it stressed the importance of creating classes with distinct roles and defining suitable interactions, such as affiliations, inheritance, and dependencies. My comprehension of software design concepts like as composition, polymorphism, inheritance, and abstraction has improved as a result of this training, and I am now able to create software that is more structured, arranged, and scalable. All things considered, this project gave me the chance to put the theoretical ideas of object-oriented programming and software design into practice in a real-world setting, which improved my system design and programming abilities. Also, the process of doing the process in recurring steps gave me an opportunity to repeat the code and find flaws that existed. This helped me in avoiding them the next time.