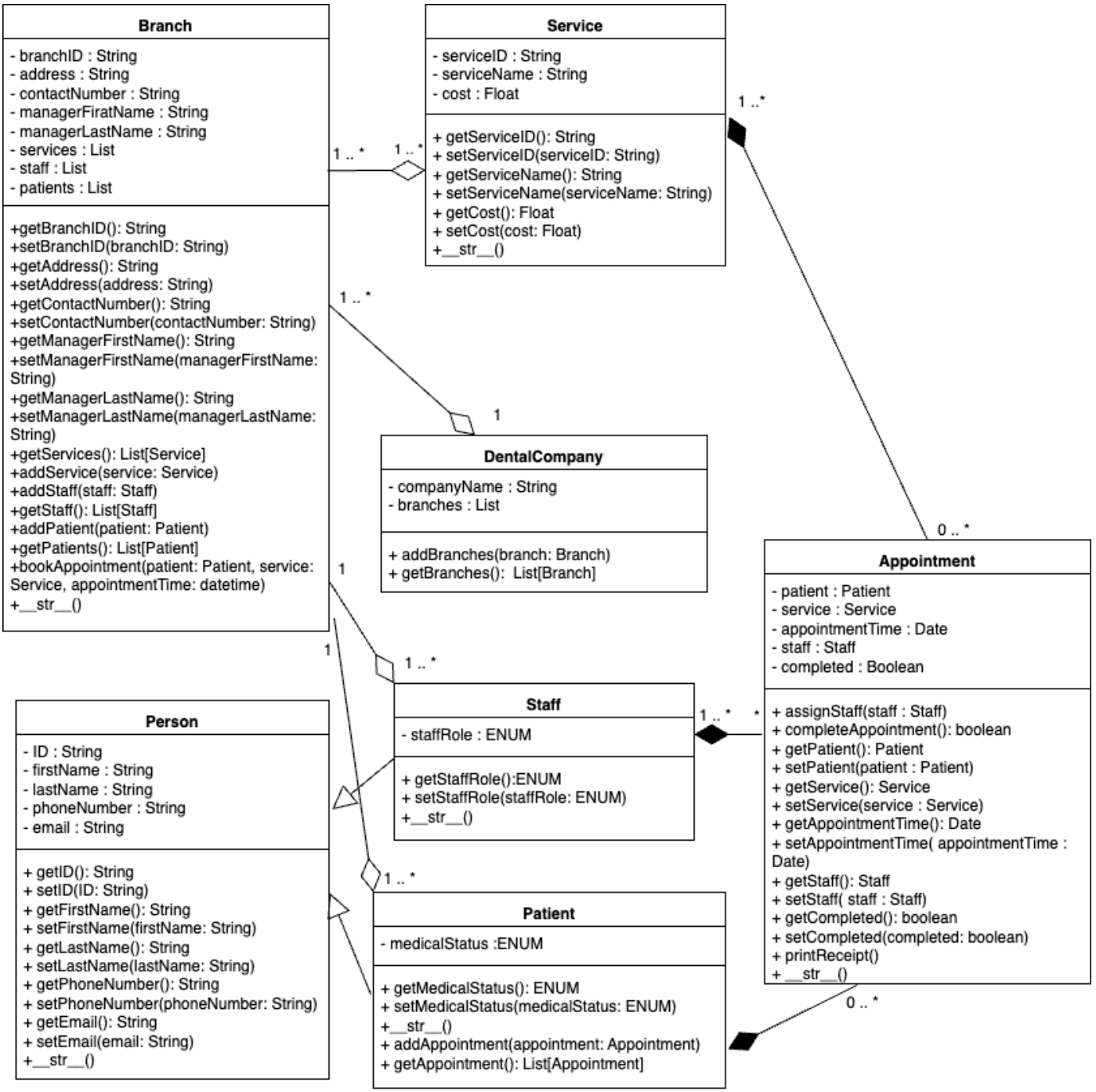
Github : https://github.com/MariamAlwars/Assignment-2-Programming

The Classes:

- DentalCompany: A class that represents a branch-based dental company.
- Branch: A class that represents a dental practice's branch, complete with services, personnel, and clients.
- Person: A class that represents a person and includes basic data like an ID, a first and last name, a phone number, and an email address.
- Staff: A class that denotes a member of the dental branch staff that derives from the Person class and adds the staff role as an additional attribute.
- Patient is a class that represents a patient in the dental branch. It derives from the Person class and adds a medical status attribute.
- Service: A category that represents a dental service with a name and duration that is provided in a branch.
- Appointment: A class that represents a patient's appointment for dental care and includes information about the patient, the dental service, and the appointment time.

The classes have numerous functions to set and get attributes, perform operations like adding services, staff, patients, and appointments to branches, as well as schedule patient appointments. Most classes have the __str__ method defined to give an object's string representation.

UML class diagram and description



The relationships

DentalCompany - Branch (Aggregation):

The "Branch" class and the "DentalCompany" class have an aggregation relationship in between. This indicates even if a "DentalCompany" object is destroyed, a "Branch" object that is associated with it can continue to function on its own because the attributes are not linked together. So, even if we delete DentalCompany from the code, Branch will still exist. One Branch is linked to one DentalCompany, and one DentalCompany is linked to one and many Branch.

Branch - Service (Aggregation):

The "Service" class and the "Branch" class are connected through aggregation. This indicates that a Branch can have multiple Service objects associated with it, and these Service objects can exist independently even if the Branch is deleted. Although the Service objects are regarded as belonging to the Branch, they are not dependent on it. A "Service" is offered by a branch, but it may be connected to more than one branch.

Branch - Staff (Aggregation):

The "Staff" class and the "Branch" class are connected through aggregation. The Branch class and the Staff class are related, and a Branch may be associated with multiple staff members. However, the Staff object's lifecycle is independent of the Branch object's.

Branch - Patient (Aggregation):

The " Patient " class and the "Branch" class are connected through aggregation. The Patient class and the Branch class are related, and a Branch object may be associated with multiple Patients, but the Patient objects' lifecycles are independent of the Branch object's.

Person - Staff (Inheritance):

The "Staff" class is an inherited version of the "Person" class because it is descended from the "Person" class. The "Staff" class can have additional attributes like "role" to represent the staff member's role in the branch in addition to inheriting the methods and attributes of the "Person" class.

Person - Patient (Inheritance):

The "Patient" class is a specialized version of the "Person" class because it derives from the "Person" class. The "Patient" class can have additional attributes like "medicalStatus" to represent the patient's medical status in addition to inheriting the methods and attributes of the "Person" class.

Appointment - Patient (Composition):

There is a composition between the "Appointment" class and the "Patient" class. This means that the Patient class and the Appointment class are composed of one another. This implies that a Patient object is a component of an Appointment object, and that the Patient object is necessary for the Appointment object's lifecycle. In other words, the associated Appointment object is also deleted when a patient object is deleted.

Appointment - Service (Composition):

The Appointment class and the Service class have a composition relationship, indicating that a Service object is a necessary component of the Appointment object's lifecycle and that an Appointment object is made up of one. In other words, a service object is necessary for an appointment object to exist, and vice versa. As such, when an appointment object is deleted, the corresponding service object should also be eliminated.

Appointment - Staff (Composition):

The Appointment class is composed of a Staff object, and the lifecycle of the Staff object is tightly coupled with the Appointment object. Because of their close ownership relationships, when the Staff object is deleted, the related Appointment object is also deleted. The associated Appointment object is also deleted when the Staff object is.

Python code to implement the UML diagram

```
from enum import Enum

# Enum for Medical Status
```

```
class MedicalStatus(Enum):
    UNDETERMINED = "Undetermined"
    GOOD = "Good"
    FAIR = "Fair"
    SERIOUS = "Serious"
    CRITICAL = "Critical"

# Enum for Staff Role
class StaffRole(Enum):
    RECEPTIONIST = "Receptionist"
    HYGIENIST = "Hygienist"
    DENTIST = "Dentist"
    MANAGER = "Manager"

# Time class for representing duration
class Time:
    def __init__(self, hours, minutes): #Initializing the attributes
        self.__hours = hours
        self.__minutes = minutes

    def __str__(self):
        return f"{self.__hours} hours {self.__minutes} minutes"

# DentalCompany class
class DentalCompany:
    def __init__(self, companyName): #Initializing the attributes
        self.__companyName = companyName
        self.__branches = []

#create functions for the class including setters and getters

    def addBranches(self, branch):
        self.__branches.append(branch)

    def getBranches(self):
        return self.__branches

# Branch class
class Branch:
    def __init__(self, branchID, address, contactNumber, managerFirstName,
managerLastName): #Initializing the attributes
        self.__branchID = branchID
        self.__address = address
        self.__contactNumber = contactNumber
        self.__managerFirstName = managerFirstName
```

```
        self.__managerLastName = managerLastName
        self.__services = []
        self.__staff = []
        self.__patients = []

#create functions for the class including setters and getters

    def getBranchID(self):
        return self.__branchID

    def setBranchID(self, branchID):
        self.__branchID = branchID

    def getAddress(self):
        return self.__address

    def setAddress(self, address):
        self.__address = address

    def getContactNumber(self):
        return self.__contactNumber

    def setContactNumber(self, contactNumber):
        self.__contactNumber = contactNumber

    def getManagerFirstName(self):
        return self.__managerFirstName

    def setManagerFirstName(self, managerFirstName):
        self.__managerFirstName = managerFirstName

    def getManagerLastName(self):
        return self.__managerLastName

    def setManagerLastName(self, managerLastName):
        self.__managerLastName = managerLastName

    def addService(self, service):
        self.__services.append(service)

    def getServices(self):
        return self.__services

    def addStaff(self, staff):
        self.__staff.append(staff)
```

```
    def getStaff(self):
        return self.__staff

    def addPatient(self, patient):
        self.__patients.append(patient)

    def getPatients(self):
        return self.__patients

    def bookAppointment(self, patient, service, appointmentTime):
        appointment = Appointment(patient, service, appointmentTime)
        return appointment

    def __str__(self):
        return "Branch ID: {}\nAddress: {}\nContact Number: {}\nManager
Name: {}".format(
            self.__branchID, self.__address, self.__contactNumber,
self.__managername
        )


# Person class
class Person:
    def __init__(self, ID, firstName, lastName, phoneNumber, email):
#Initializing the attributes
        self.__ID = ID
        self.__firstName = firstName
        self.__lastName = lastName
        self.__phoneNumber = phoneNumber
        self.__email = email

#create functions for the class including setters and getters

    def getID(self):
        return self.__ID

    def setID(self, ID):
        self.__ID = ID

    def getFirstName(self):
        return self.__firstName

    def setFirstName(self, firstName):
        self.__firstName = firstName
```

```
    def getLastName(self):
        return self.__lastName

    def setLastName(self, lastName):
        self.__lastName = lastName

    def getPhoneNumber(self):
        return self.__phoneNumber

    def setPhoneNumber(self, phoneNumber):
        self.__phoneNumber = phoneNumber

    def getEmail(self):
        return self.__email

    def setEmail(self, email):
        self.__email = email

    def __str__(self):
        return f"ID: {self.__ID}, First Name: {self.__firstName}, Last
Name: {self.__lastName}, Phone Number: {self.__phoneNumber}, Email:
{self.__email}"
    #def __str__(self):
        #return f"ID: {self.__ID}, Name: {self.__name}, Phone Number:
{self.__phoneNumber}, Email: {self.__email}"

# Staff class inheriting from Person
class Staff(Person):
    def __init__(self, ID, firstName, lastName, phoneNumber, email,
staffRole): #Initializing the attributes
        super().__init__(ID, firstName, lastName, phoneNumber, email)
#using super function  to inherit and invoke methods from its superclass
        self.__staffRole = staffRole

#create functions for the class including setters and getters

    def getStaffRole(self):
        return self.__staffRole

    def setStaffRole(self, staffRole):
        self.__staffRole = staffRole

    def __str__(self):
```

```
        return f"ID: {self.__getID()}, First Name: {self.__firstName},
Last Name: {self.__lastName}, Phone Number: {self.getPhoneNumber()},
Email: {self.getEmail()}, Staff Role: {self.__staffRole.value}"


# Patient class inheriting from Person
class Patient(Person):
    def __init__(self, ID, firstName, lastName, phoneNumber, email,
medicalStatus): #Initializing the attributes
        super().__init__(ID, firstName, lastName, phoneNumber, email)
#using super function  to inherit and invoke methods from its superclass
        self.__medicalStatus = medicalStatus
        self.__appointments = []

#create functions for the class including setters and getters

    def getMedicalStatus(self):
        return self.__medicalStatus

    def setMedicalStatus(self, medicalStatus):
        self.__medicalStatus = medicalStatus

    def __str__(self):
        appointment_str = "\n".join([str(appointment) for appointment in
self.__appointments])
        return f"ID: {self.getID()}, First Name: {self.getFirstName()},
Last Name: {self.getLastName()}, Phone Number: {self.getPhoneNumber()},
Email: {self.getEmail()}, Medical Status: {self.__medicalStatus.value},
Appointments:{appointment_str}"

    def addAppointment(self, appointment):
        self.__appointments.append(appointment)

    def getAppointment(self):
        return self.__appointments


# Service class
class Service:
    def __init__(self, serviceID, serviceName, cost): #Initializing the
attributes
        self.__serviceID = serviceID
        self.__serviceName = serviceName
        self.__cost = cost
```

```
#create functions for the class including setters and getters

    def getServiceID(self):
        return self.__serviceID

    def setServiceID(self, serviceID):
        self.__serviceID = serviceID

    def getServiceName(self):
        return self.__serviceName

    def setServiceName(self, serviceName):
        self.__serviceName = serviceName

    def getCost(self):
        return self.__cost

    def setCost(self, cost):
        self.__cost = cost

    def __str__(self):
        return f"Service ID: {self.__serviceID}, Service Name:
{self.__serviceName}, Cost: {self.__cost}"


class Appointment:
    def __init__(self, patient, service, appointmentTime): #Initializing
the attributes
        self.__patient = patient
        self.__service = service
        self.__appointmentTime = appointmentTime
        self.__staff = None
        self.__completed = False

#create functions for the class including setters and getters

    def assignStaff(self, staff):
        self.__staff = staff

    def completeAppointment(self):
        self.__completed = True

    def printReceipt(self):
        if self.__completed:
            vat = self.__service.getCost() * 0.05  #calculating the 5% vat
```

```
            totalCost = self.__service.getCost() + vat
            print("Receipt for Appointment")
            print("Service: ", self.__service.getServiceName())
            print("Staff: ", self.__staff.getFirstName())
            print("Patient: ", self.__patient.getFirstName())
            print("Appointment Time: ",
self.__appointmentTime.strftime("%Y-%m-%d %H:%M:%S"))
            print("Service Cost: ", self.__service.getCost())
            print("VAT: ", vat)
            print("Total Cost: ", totalCost)
        else:
            print("Error: Appointment not completed")

    def getPatient(self):
        return self.__patient

    def setPatient(self, patient):
        self.__patient = patient

    def getService(self):
        return self.__service

    def setService(self, service):
        self.__service = service

    def getAppointmentTime(self):
        return self.__appointmentTime

    def setAppointmentTime(self, appointmentTime):
        self.__appointmentTime = appointmentTime

    def getStaff(self):
        return self.__staff

    def setStaff(self, staff):
        self.__staff = staff

    def getCompleted(self):
        return self.__completed

    def setCompleted(self, completed):
        self.__completed = completed

    def __str__(self):
```

```
        return f"Appointment Details:\nService:
{self.__service.getServiceName()}\nStaff:
{self.__staff.getFirstName()}\nPatient:
{self.__patient.getFirstName()}\nAppointment Time:
{self.__appointmentTime.strftime('%Y-%m-%d %H:%M:%S')}\nTotal Cost:
${self.__service.getCost()}\nCompleted: {self.__completed}"
```

Test Case

```
import datetime


# Test Case of: the addition of patients booking appointments
dentalCompany = DentalCompany(companyName="Bright Smiles")
branch1 = Branch(branchID="B1", address="Dubai",
contactNumber="05533993399", managerFirstName="Fatima", managerLastName =
"Alrzy")
service1 = Service(serviceID="990", serviceName="fillings", cost=1200.0)
service2 = Service(serviceID="980", serviceName="cleaning", cost=200.0)
staff1 = Staff(ID="717", firstName="Dr. Aisha", lastName = "Al Ali",
phoneNumber="0567700045", email="Dr_Aisha@gmail.com",
staffRole=StaffRole.DENTIST)
patient1 = Patient(ID="1563", firstName="Maryam", lastName = "Alward",
phoneNumber="0509900000", email="maryam89@gmail.com",
medicalStatus=MedicalStatus.SERIOUS)
patient2 = Patient(ID="1574", firstName="Shamma", lastName = "Alameeri",
phoneNumber="0509900110", email="Shamma9@gmail.com",
medicalStatus=MedicalStatus.UNDETERMINED)


staff2 = Staff(ID="318", firstName="Dr. Ahmed", lastName= "Almatrooshi",
phoneNumber="0552223344", email="Dr_Ahmed@gmail.com",
staffRole=StaffRole.DENTIST)
branch1.addStaff(staff2)

# Book Appointments for Patients
appointmentTime1 = datetime.datetime(2023, 4, 15, 10, 30)
appointmentTime2 = datetime.datetime(2023, 4, 16, 15, 0)

appointment1 = Appointment(patient1, service1, appointmentTime1)
appointment1.assignStaff(staff1)
branch1.bookAppointment(patient1, service1, appointmentTime1)
patient1.addAppointment(appointment1)
print(patient1)
```

```
appointment2 = Appointment(patient2, service2, appointmentTime2)
appointment2.assignStaff(staff2)
branch1.bookAppointment(patient2, service2, appointmentTime2)
patient2.addAppointment(appointment2)
print(patient2)
```

Output

```
ID: 1563, First Name: Maryam,  Last Name: Alward, Phone Number:
0509900000, Email: maryam89@gmail.com, Medical Status: Serious,
Appointments:Appointment Details:
Service: fillings
Staff: Dr. Aisha
Patient: Maryam
Appointment Time: 2023-04-15 10:30:00
Total Cost: $1200.0
Completed: False
ID: 1574, First Name: Shamma,  Last Name: Alameeri, Phone Number:
0509900110, Email: Shamma9@gmail.com, Medical Status: Undetermined,
Appointments:Appointment Details:
Service: cleaning
Staff: Dr. Ahmed
Patient: Shamma
Appointment Time: 2023-04-16 15:00:00
Total Cost: $200.0
Completed: False
```

Test Case

```
import datetime


# Test Case 1: Create an appointment and print receipt
patient1 = Patient(ID="1563", firstName="Maryam", lastName = "Alward",
phoneNumber="0509900000", email="maryam89@gmail.com",
medicalStatus=MedicalStatus.SERIOUS)
service1 = Service(serviceID = "990", serviceName = "fillings", cost =
1200.0)
appointmentTime1 = datetime.datetime(2023, 4, 15, 10, 30)
staff1 = Staff(ID="717", firstName="Dr. Aisha", lastName = "Al Ali",
phoneNumber="0567700045", email="Dr_Aisha@gmail.com",
staffRole=StaffRole.DENTIST)
appointment1 = Appointment(patient1, service1, appointmentTime1)
appointment1.assignStaff(staff1)
```

```
appointment1.completeAppointment()
appointment1.printReceipt()
```

Output

```
Receipt for Appointment
Service:  fillings
Staff:  Dr. Aisha
Patient:  Maryam
Appointment Time:  2023-04-15 10:30:00
Service Cost:  1200.0
VAT:  60.0
Total Cost:  1260.0
```

Test Case

```
# Test Case of: the addition of branches to the dental company

dentalCompany = DentalCompany(companyName="Bright Smiles")

branch1 = Branch(branchID="B1", address="Dubai",
contactNumber="05533993399", managerFirstName="Fatima", managerLastName =
"Alrzy")
branch2 = Branch(branchID = "B2", address="Ajman", contactNumber =
"05677001145",  managerFirstName = "Reem", managerLastName = "AlAhli")
branch3 = Branch(branchID = "B3", address="Sharjah", contactNumber =
"0569375932",  managerFirstName = "Mohammed", managerLastName = "Alwars")

dentalCompany.addBranches(branch1)
dentalCompany.addBranches(branch2)
dentalCompany.addBranches(branch3)

branches = dentalCompany.getBranches()
assert len(branches) == 3
```

Output

```
No error
```

Test Case

```
# Test Case of:  the addition of dental services, staff, and patients to a
branch.

dentalCompany = DentalCompany(companyName="Bright Smiles")
branch1 = Branch(branchID="B1", address="Dubai",
contactNumber="05533993399", managerFirstName="Fatima", managerLastName =
"Alrzy")
service1 = Service(serviceID="990", serviceName="fillings", cost=1200.0)
service2 = Service(serviceID="980", serviceName="cleaning", cost=200.0)
staff1 = Staff(ID="717", firstName="Dr. Aisha", lastName = "Al Ali",
phoneNumber="0567700045", email="Dr_Aisha@gmail.com",
staffRole=StaffRole.DENTIST)
patient1 = Patient(ID="1563", firstName="Maryam", lastName = "Alward",
phoneNumber="0509900000", email="maryam89@gmail.com",
medicalStatus=MedicalStatus.SERIOUS)
patient2 = Patient(ID="1574", firstName="Shamma", lastName = "Alameeri",
phoneNumber="0509900110", email="Shamma9@gmail.com",
medicalStatus=MedicalStatus.UNDETERMINED)


dentalCompany.addBranches(branch1)
branch1.addService(service1)
branch1.addService(service2)
branch1.addStaff(staff1)
branch1.addPatient(patient1)
branch1.addPatient(patient2)

services = branch1.getServices()
staff = branch1.getStaff()
patients = branch1.getPatients()

assert len(services) == 2
assert len(staff) == 1
assert len(patients) == 2
```

Output

```
No error
```

Summary of learning

As a student, this assignment has given me worthwhile opportunities to learn about various facets of object-oriented programming using Python and the creation of UML class diagrams. I now have a firm grasp on the foundations of UML class diagrams, including how to draw and read them. The definition of attributes, methods, setters, and getters for classes in Python is another skill I've picked up.

Using inheritance to create classes that inherit traits and behaviors from parent classes is one of the key ideas I've learned. Additionally, I now understand how access modifiers like private and public can be used to restrict access to class members while still ensuring encapsulation in object-oriented programming.

I have developed my problem-solving abilities through this assignment by converting actual-world situations into UML class diagrams and putting them into practice in Python code. Common Python programming concepts like classes, objects, methods, attributes, and encapsulation are also something I've learned about. I now have a solid understanding of object-oriented programming principles and syntax.

In order to comprehend the organization and connections between classes in a system, I have also improved my reading and interpretation skills for UML class diagrams. My understanding of design patterns, code organization, and best practices for object-oriented programming has improved as a result of this.

By creating and analyzing UML class diagrams, I have also enhanced my presentation and communication abilities. I've gained practical experience using Python's classes, objects, methods, and attributes to model real-world scenarios as a result of this assignment.

Overall, completing this assignment has been a worthwhile educational experience that has improved my knowledge of Python class implementation, UML class diagrams, and object-oriented programming concepts. I can become a more skilled and productive programmer by using the knowledge and skills I've learned from this assignment in my future software development projects.