ICS220 22873 Program. Fund.

Assignment 2: Software Implementation

Prof Kuhail

Fri 14 April 2023

Shamsa Rasehd

multiple branches
Each dental branch has an address, phone number, and a manager.

Scenario:

Consider the following problem statement:

A dental company, “Bright Smiles”, has multiple branches (i.e., dental clinics) and would like you to create a software application to help manage the business processes and dental services. Each dental branch has an address, phone number, and a manager. A dental branch offers dental services to patients. Examples of services include cleaning, implants, crowns, fillings, and more. Each of the services has a cost. The clinic keeps track of its patients and staff. The staff includes managers, receptionists, hygienists, and dentists. The patient needs to book an appointment before coming to the clinic. Upon checkout, the clinic charges the patient depending on the services she/he has received. Also, a 5% value-added tax (VAT) is added to the final bill.

1. DentalCompany:
    i. Name: string
    ii. branch:List[Branch]
2. Branch:
    - name :string
    - address:string
    - phone number:string
    - manager:Manger
    - dental services:List[Services]
    - patients:List[Patient]
    - staff:List[Staff]

3. Person:
- firstName: string
- lastName: string
- phoneNumber:string
- gender:Enum
- dateOfBirth:string

a. Staff (inherits from Person):
    - employeeID:string

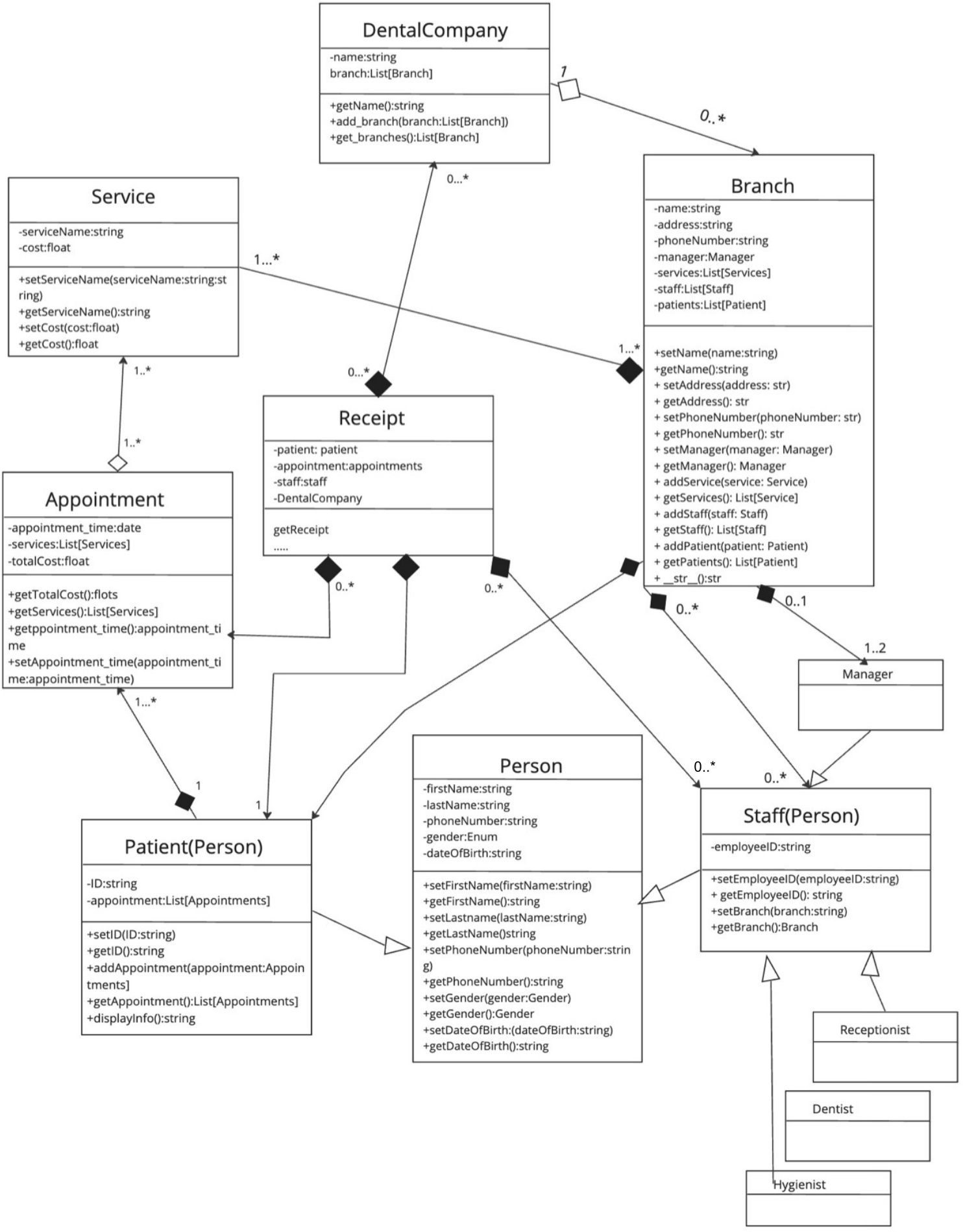
    i. Manager (inherits from StaffMember):
    ii. Receptionist (inherits from StaffMember):
    iii. Hygienist (inherits from StaffMember):
    iv. Dentist (inherits from StaffMember):

b. Patient (inherits from Person):
    - appointments: List[Appointment]
    - ID:string

4. Appointment:
- services:List[Services]
- dateTime:datetime
- Method: totalCost:float

5. Services:
- serviceName:string
- cost:float

6. Receipt:
- patient:patient
- appointments:appointments
- Staff:staff
- DentalCompany:DentalCompany


DentalCompany
-name:string
branch:List[Branch]
+getName():string
+add_branch(branch:List[Branch])
+get_branches():List[Branch]
1
0..*
Service
-serviceName:string
-cost:float
+setServiceName(serviceName:string:string)
+getServiceName():string
+setCost(cost:float)
+getCost():float
1...*
0...*
Branch
-name:string
-address:string
-phoneNumber:string
-manager:Manager
-services:List[Services]
-staff:List[Staff]
-patients:List[Patient]
+setName(name:string)
+getName():string
+ setAddress(address: str)
+ getAddress(): str
+ setPhoneNumber(phoneNumber: str)
+ getPhoneNumber(): str
+ setManager(manager: Manager)
+ getManager(): Manager
+ addService(service: Service)
+ getServices(): List[Service]
+ addStaff(staff: Staff)
+ getStaff(): List[Staff]
+ addPatient(patient: Patient)
+ getPatients(): List[Patient]
+ __str__():str
1...*
1..*
1..*
Receipt
-patient: patient
-appointment:appointments
-staff:staff
-DentalCompany
getReceipt
.....
0...*
Appointment
-appointment_time:date
-services:List[Services]
-totalCost:float
+getTotalCost():flots
+getServices():List[Services]
+getppointment_time():appointment_time
+setAppointment_time(appointment_time:appointment_time)
0..*
0..*
0..*
0..1
1..2
Manager
1...*
1
1
Person
-firstName:string
-lastName:string
-phoneNumber:string
-gender:Enum
-dateOfBirth:string
+setFirstName(firstName:string)
+getFirstName():string
+setLastname(lastName:string)
+getLastName()string
+setPhoneNumber(phoneNumber:string)
+getPhoneNumber():string
+setGender(gender:Gender)
+getGender():Gender
+setDateOfBirth:(dateOfBirth:string)
+getDateOfBirth():string
0..*
0..*
Patient(Person)
-ID:string
-appointment:List[Appointments]
+setID(ID:string)
+getID():string
+addAppointment(appointment:Appointments]
+getAppointment():List[Appointments]
+displayInfo():string
Staff(Person)
-employeeID:string
+setEmployeeID(employeeID:string)
+ getEmployeeID(): string
+setBranch(branch:string)
+getBranch():Branch
Receptionist
Dentist
Hygienist

I have created a more organized, modular, and efficient codebase by making use of relationships such as association, composition, and inheritance. Let me elaborate on these connections and describe how they improve the code's readability and efficiency:

1. Aggregation between DentalCompany and Branch:A dental company has multiple branches, but the branches can exist independently of the dental company. If the dental company is deleted, the branches can still exist on their own.
2. the association between Appointment and Service is an example of a many-to-many aggregation association. Aggregation represents a "whole/part" relationship where one class (the whole) contains or is composed of multiple instances of another class (the parts). In this case, an Appointment can have multiple Service instances (the parts), and each Service instance can be associated with multiple Appointment instances.
3. Composition between Branch and Manager:As explained earlier, a branch has a manager, and the manager's lifecycle is dependent on the branch's lifecycle.
4. Composition between Branch and Staff:A branch has a list of staff members, and each staff member is associated with a specific branch. The staff's lifecycle is dependent on the branch's lifecycle.
5. Composition between Branch and Patient:A branch has a list of patients registered at that branch. The patients are associated with a specific branch, and their lifecycle is dependent on the branch's lifecycle.
6. Inheritance between Person and its subclasses (Staff, Patient):The Person class serves as the base class for both Staff and Patient classes, and they inherit attributes and methods from the Person class.
7. The Receipt class has a patient ,appointments,staff ,DentalCompany attribute that holds a reference to a Patient , appointments ,staff and DentalCompany objects. This relationship can be described as a "has-a" "composition" relationship .This means that a Receipt object is composed of or has references to objects from these classes to represent the details of a specific receipt, such as the patient who received the services, the appointments when the services were provided, the staff member who provided the services, and the dental company where the services were provided.

I find that composition relationships better represent real-world relationships and provide stronger "has-a" connections.Using Composition aids me in ensuring that my code is consistent and that my data is correct. Composition relationships also help me model "part-whole" relationships in the real world, which makes my code more accurate to life and enhances its quality. Inheritance offers additional benefits, such as code reusability and accurate modeling of real-world relationships. By allowing a subclass to inherit properties and methods from a superclass, I can encourage code reuse and reduce the need for duplication. Subclasses can inherit properties and behaviors from their superclasses, just as real-world objects inherit properties and behaviors from their parents. This facilitates the creation of a more realistic and effective code structure.

In the program, I structured multiple classes, each representing a specific aspect of the clinic.

DentalCompany: Represents the dental company, containing the name of the company and a list of branches. I added methods to add branches and retrieve the list of branches.

Branch: Represents a clinic branch, with attributes such as name, address, phone number, and manager. I included lists of services, patients, and staff. I provided methods to add and retrieve services, patients, and staff.

Person: A base class representing a person, with attributes like first name, last name, phone number, gender, and date of birth. I created setter and getter methods for each attribute.

Staff: A subclass of Person, representing a staff member in the clinic. It has an additional attribute, employee ID, and a method to set and get the associated branch.

Manager, Receptionist, Hygienist, Dentist: Subclasses of Staff, representing different roles in the clinic.

Patient: A subclass of Person, representing a patient in the clinic. It has an additional attribute, ID, and a list of appointments. I provided methods to book appointments and retrieve the list of appointments.

Appointment: Represents an appointment, with attributes like services and appointment time. I included methods to calculate the total cost of services, and set and get the services and appointment time.

Service: Represents a dental service, with attributes like service name and cost. I created setter and getter methods for each attribute.

Receipt: Represents a receipt, with attributes like patient, appointments, staff, and dental company. I added a method to print the receipt, including details like services, costs, and VAT.

In the main function of the program, I demonstrated how to create instances of these classes and establish relationships between them. For example, I created a dental company, branches, services, staff members, patients, and appointments. I then associated staff members with branches, added patients and appointments, and finally generated receipts for the appointments.

I made sure the code effectively showcases the use of inheritance, composition, and association relationships to model a real-life scenario, making it easier to understand and maintain. The classes are well-organized, and methods and attributes are appropriately named, contributing to the code's readability.

In the program, I carefully structured multiple classes to make the dental clinic management system comprehensive and organized. In the Branch class, I included services, patients, and staff lists to facilitate archiving all necessary information for each branch. This design choice makes it easier to manage and maintain the system and allows for efficient access to relevant data.By incorporating these lists into the Branch class, the system can quickly retrieve information related to services offered, patient records, and staff members working at a specific branch. This level of organization not only improves the efficiency of the dental clinic management system but also streamlines the process of adding or updating information within the system.

During the process of making a management system for a dental clinic using object-oriented programming, I learned a number of important ideas and methods, including:

- I learned to design classes and objects that represent key components of the system, such as DentalCompany, Branch, Person, Staff, Manager, Receptionist, Hygienist, Dentist, Patient, Appointment, Service, and Receipt.

- I learned to implement inheritance by creating subclasses (e.g., Staff, Manager, Receptionist, Hygienist, Dentist) that inherit properties and methods from a parent class (e.g., Person).

- I learned to employ encapsulation, concealing a class's internal implementation details through private attributes (e.g., __name, __address, __phoneNumber) and offering public methods (getters and setters) to access and modify these attributes.

- I learned to establish composition and association relationships between classes, such as adding services, staff members, and patients to a Branch or linking appointments with patients.

- I learned to utilize enumeration (Enum) to define a collection of named values, like Gender in this case, which aids in assigning valid values to attributes.

- I learned to work with the datetime module for handling dates and times, such as creating and formatting appointment times.

- I learned to conduct calculations and manipulations with objects, like determining the total cost of appointments and producing receipts.

- I learned to create a main function (main()) to execute the code, showcasing the system's functionality by generating instances of the classes, setting their attributes, and invoking their methods. This function also tests the implemented features and verifies their proper operation.

- In conclusion, the experience of constructing a dental clinic management system using object-oriented programming principles has deepened my comprehension of these concepts and enhanced my capacity to apply them in practical situations.