

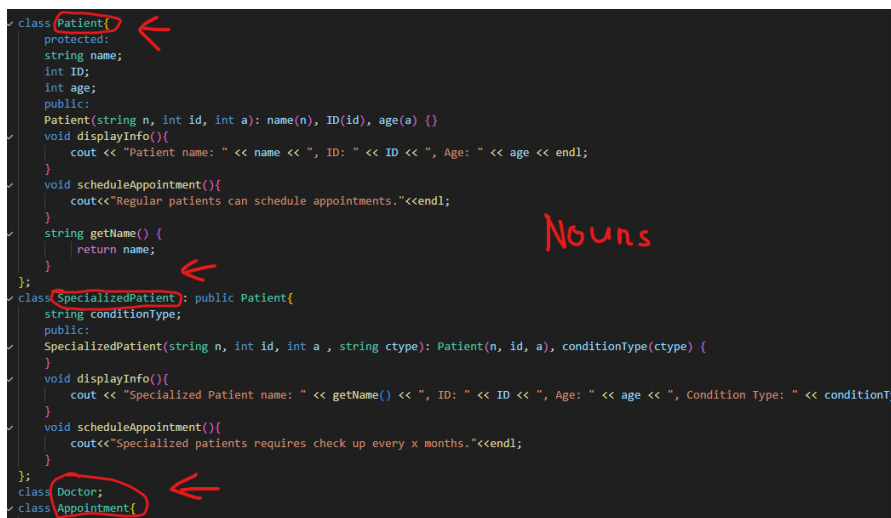
Documentation for Mini Clinic Management System

- Written by student ID:24110073 -

I. Object-Oriented Analysis (OOA)

The objects also known as nouns are the main factors/classes that define the code. I have identified the “nouns” as:

- Patient
- SpecializedPatient
- Doctor
- Appointment



```
class Patient{
protected:
    string name;
    int ID;
    int age;
public:
    Patient(string n, int id, int a): name(n), ID(id), age(a) {}
    void displayInfo(){
        cout << "Patient name: " << name << ", ID: " << ID << ", Age: " << age << endl;
    }
    void scheduleAppointment(){
        cout<<"Regular patients can schedule appointments."<<endl;
    }
    string getName() {
        return name;
    }
};
class SpecializedPatient: public Patient{
    string conditionType;
public:
    SpecializedPatient(string n, int id, int a, string ctype): Patient(n, id, a), conditionType(ctype) {}
    void displayInfo(){
        cout << "Specialized Patient name: " << getName() << ", ID: " << ID << ", Age: " << age << ", Condition Type: " << conditionType << endl;
    }
    void scheduleAppointment(){
        cout<<"Specialized patients requires check up every x months."<<endl;
    }
};
class Doctor;
class Appointment;
```

Nouns

II. Attributes for objects (descriptive nouns)

Attributes are the “children” of the classes that represents a type of info in which the class will use to display later.

Attributes for each object:

- Patient:
 - i. Name
 - ii. ID
 - iii. Age

```
class Patient{
protected:
    string name;
    int ID;
    int age;
```

- SpecializedPatient (Inherited Name,ID,Age from Patient):
 - i. conditionType

```
class SpecializedPatient : public Patient{
    string conditionType;
```

- Doctor
 - i. Name
 - ii. ID

- iii. Specialty
- iv. Appointments

```
class Doctor{
    string name;
    int ID;
    string specialty;
    Appointment* appointments[5];
}
```

- Appointment
 - i. Date
 - ii. Time
 - iii. Reason
 - iv. Status
 - v. Assigned patient/doctor

```
class Appointment{
    string date,time,reason,status;
    Patient* patient;
    Doctor* doctor;
}
```

III. Methods (verbs)

Methods are the functions that serves as a base for display info and update statuses.

Methods for each object:

- Patient:
 - i. displayInfo()
 - ii. scheduleAppointment()
 - iii. getName()

```
void displayInfo(){
    cout << "Patient name: " << name << ", ID: " << ID << ", Age: " << age << endl;
}
void scheduleAppointment(){
    cout<<"Regular patients can schedule appointments."<<endl;
}
string getName() {
    return name;
}
};
```

- SpecializedPatient:
 - i. displayInfo()
 - ii. scheduleAppointment()

```

class SpecializedPatient : public Patient{
    string conditionType;
public:
    SpecializedPatient(string n, int id, int a , string ctype): Patient(n, id, a), conditionType(ctype) {}
}
void displayInfo(){
    cout << "Specialized Patient name: " << name << ", ID: " << ID << ", Age: " << age << ", Condition Type: " << conditionType << endl;
}
void scheduleAppointment(){
    cout<<"Specialized patients requires check up every x months."<<endl;
}
};

```

- Doctor
 - i. displayInfo()
 - ii. AssignAppointment
 - iii. getname()

```

class Doctor{
    string name;
    int ID;
    string specialty;
    Appointment* appointments[5];
    int count;
public:
    Doctor(string n, int id, string spec): name(n), ID(id), specialty(spec), count(0) {}
    void AssignAppointment(Appointment* a){
        appointments[count++] = a;
    }
    void displayInfo(){
        cout<<"Name: "<<name<<" , ID: "<<ID<<" , Special Field: "<<specialty<<endl;
        cout<<"Appointments: "<<endl;
        for (int i=0; i<count; i++){
            appointments[i]->displayInfo();
        }
    }
    string getName() {
        return name;
    }
};

```

- Appointment
 - i. displayInfo()
 - ii. updateStatus

```

class Appointment{
    string date,time,reason,status;
    Patient* patient;
    Doctor* doctor;
public:
    Appointment(string d, string t, string r, Patient* p, Doctor* d) {}
    void displayInfo();
    void updateStatus(string newStatus){
        status = newStatus;
    }
};

```

IV. Inheritance relationship

- SpecializedPatient class inherited directly from the Patient class which overrides displayInfo() and scheduleAppointment() making them display separate and different infos.

CLASS DESIGN EXPLANATIONS

1. Patient

- Class represents normal patients visiting the clinic.
- Contains basic attributes (name, ID, age).
- Contains methods to display patient's info and schedule appointments.
- Serves as the base class for specialized patient class to inherit.

2. SpecializedPatient

- Represents patients with special cases in the clinic.
- Inherits all of the attributes from the Patient class.
- Adds extra attribute conditionType.
- Overrides displayInfo and scheduleAppointment methods which shows extra condition details and extended check up frequency demonstrating inheritance and polymorphism respectively.

3. Doctor

- Represents doctor within the clinic.
- Includes basic attributes (name, id, special field) and assigned appointments
- Methods allowing doctors to assign appointments and display info

4. Appointment

- Represents scheduled appointments between doctor and patient
- Includes time, date, reason and status with pointers linking to the patient and doctor
- Implemented methods to display appointment detail (displayInfo) and update the appointment's status (updateStatus)
- Serves as a connector between the doctor and patient

→ Inheritance is used to define SpecializedPatient from Patient, with extra details and requirements helping the system clear and realistic, making managing the clinic easier

KEY PARTS OF THE CODE

- Patient store personal info
- SpecializedPatient contains more info than patient (chronic conditions)
- Appointment links doctor and patient
- Doctors manage the appointments
- SpecializedPatient overrides behavior of Patient showing polymorphism and inheritance

TEST RESULTS

The code properly displayed the patient info, special patients info and condition along with the doctors' information and correctly assigned the appointments between the doctors and patient printing out the correct result while constantly updating the status of the appointments.

```
Patient name: John Roblox, ID: 46, Age: 26
Specialized Patient name: Jane Minecraft, ID: 682, Age: 46, Condition Type: Obesity
Name: Dr. Slime, ID: 420, Special Field: Neurology
Appointments:
Date: 08-09-2025, Time: 3:00 PM, Reason: General Checkup, Patient: John Roblox, Doctor: Dr. Slime, Status: Available
Name: Dr. Pork, ID: 1337, Special Field: Cardiology
Appointments:
Date: 09-09-2025, Time: 12:45 PM, Reason: Obesity Checkup, Patient: Jane Minecraft, Doctor: Dr. Pork, Status: Available
Name: Dr. Slime, ID: 420, Special Field: Neurology
Appointments:
Date: 08-09-2025, Time: 3:00 PM, Reason: General Checkup, Patient: John Roblox, Doctor: Dr. Slime, Status: Completed
Name: Dr. Pork, ID: 1337, Special Field: Cardiology
Appointments:
Date: 09-09-2025, Time: 12:45 PM, Reason: Obesity Checkup, Patient: Jane Minecraft, Doctor: Dr. Pork, Status: Cancelled
```

LLM USAGE

The assignment wouldn't have been possible without the help of ChatGPT and GitHub Copilot. I used GitHub Copilot to assist with correcting minor typos and errors that I often make while writing codes. I then used ChatGPT to assist with correcting pointers for the assignments between the doctor and patient by prompting "I couldn't figure out why was my code giving errors while setting up assignments, give me a quick and simple fix to connect the doctor and the patient in the appointments". The majority of the code were written by myself from setting up base classes and linking them (inheritance) to creating object examples (doctor, patient and meeting details) based from what I've learned from examples written on class by the lecturer.