Small Clinic Management System Documentation

Author: Trần Lê Dũng
ID: 24110084
Date: September 9, 2025

Object-Oriented Analysis (OOA)

The Small Clinic Management System streamlines operations in a small medical clinic by managing doctors, patients (regular and chronic), and appointments. The system supports:

- **Doctor Management**: Store and update details (name, ID, specialty) and display information.
- **Patient Management**: Track patient details (name, ID, age, medical history) with specialized handling for chronic patients (condition type, last checkup).
- **Appointment Management**: Schedule, update, cancel, and complete appointments, linking doctors and patients.
- **Clinic Operations**: Manage collections of doctors, patients, and appointments, ensuring data consistency and memory safety.

Key Entities and Relationships:

- **Doctor**: Attributes include name, ID, and specialty. Associated with appointments.
- **Patient**: Base entity with name, ID, age, and medical history. Supports polymorphism for chronic patients.
- **ChronicPatient**: Extends `Patient` with condition type and last checkup, requiring specialized appointment scheduling.
- **Appointment**: Links a doctor and patient, with attributes for ID, date, time, reason, and status (Scheduled, Cancelled, Completed).
- **Clinic**: Aggregates doctors, patients, and appointments, providing methods to add, display, and select entities.

Requirements:

- **Functional**: Create/update/display entities, schedule appointments, maintain medical histories.
- **Non-functional**: Ensure modularity, extensibility, and user-friendly output.

Class Design and Inheritance

The system comprises five classes, designed with OOP principles for encapsulation, modularity, and extensibility:

- **Doctor**: Encapsulates name, ID, and specialty with private attributes. Provides getters (`getName`, `getID`, `getSpecialty`), setters, and `displayInfo`. No inheritance, as doctors have uniform behavior.
- **Patient**: Base class with private attributes for name, ID, age, and a `vector<string>` for medical history. Includes history management methods (`addHistory`, `removeHistory`, `showHistory`) and a virtual `scheduleAppointment` method for polymorphism. A virtual destructor ensures proper cleanup.
- **ChronicPatient**: Inherits publicly from `Patient`, adding `conditionType` and `lastCheckup`. Overrides `scheduleAppointment` for chronic-specific behavior, demonstrating polymorphism. Inheritance reuses

`Patient` functionality.

- **Appointment**: Encapsulates ID, date, time, reason, status, and pointers to `Doctor` and `Patient`. Includes getters, setters, status updates (`cancel`, `complete`), and `displayInfo`. Default status is "Scheduled."
- **Clinic**: Manages `vector`s of `Doctor*`, `Patient*`, and `Appointment*`. Provides methods to add (`addDoctor`, `addPatient`, `addAppointment`), display lists, and select entities (`chooseDoctor`, `choosePatient`, `chooseAppointment`). A destructor frees memory.

Inheritance and Polymorphism:

- `ChronicPatient` inherits from `Patient` to model the "is-a" relationship, reusing attributes and methods while adding specialized behavior.
- Polymorphism is achieved via the virtual `scheduleAppointment` method, allowing different scheduling logic for regular and chronic patients.
- Encapsulation restricts direct attribute access, and `Clinic` uses aggregation to manage entity relationships.

Code Walkthrough

The C++ code in `smallclinic.cpp` implements the system with clear OOP principles. Key components include:

- **Patient Class Definition**:

```
class Patient {
private:
    string name;                    // Tên bệnh nhân
    string ID;                      // Mã số định danh bệnh nhân
    int age;                        // Tuổi bệnh nhân
    vector<string> medicalHistory; // Danh sách các lần khám bệnh (lịch sử)
public:
    Patient(string _name, string _ID, int _age) {
        name = _name;
        ID = _ID;
        age = _age;
    }
```

  Encapsulates patient data and initializes objects. The `vector<string>` stores dynamic medical history records.

- **ChronicPatient Polymorphism**:

```
class ChronicPatient : public Patient {
private:
    string conditionType; // Loại bệnh mãn tính
    string lastCheckup;   // Ngày khám gần nhất
public:
    ChronicPatient(string _name, string _ID, int _age, string
_conditionType, string _lastCheckup)
```

```
        : Patient(_name, _ID, _age) {
        conditionType = _conditionType;
        lastCheckup = _lastCheckup;
    }
    void scheduleAppointment(string appID, string date, string time, string
reason) override {
        cout << "Chronic patient requires regular check-up...!" << endl;
```

The constructor reuses `Patient`'s initialization, and the overridden `scheduleAppointment` adds chronic-specific output and history entries.

- **Appointment Status Management**:

```
Appointment(string _appID, string _date, string _time, string _reason,
string _status, Doctor* _doctor, Patient* _patient) {
    appID = _appID;
    date = _date;
    time = _time;
    reason = _reason;
    doctor = _doctor;
    patient = _patient;
    status = "Scheduled";
}
```

Initializes appointments with a default "Scheduled" status, linking doctor and patient.

- **Clinic Memory Management**:

```
~Clinic() {
    for (int i = 0; i < patients.size(); i++) delete patients[i];
    for (int i = 0; i < doctors.size(); i++) delete doctors[i];
    for (int i = 0; i < appointments.size(); i++) delete appointments[i];
}
```

Ensures no memory leaks by deallocating dynamically allocated objects.

- **Main Function Test Cases**:

```
int main() {
    Clinic clinic;
    cout << "=== Test Case 1: Adding Doctors ===" << endl;
    Doctor* doc1 = new Doctor("Dr. Smith", "D001", "Cardiology");
    Doctor* doc2 = new Doctor("Dr. Johnson", "D002", "Neurology");
    clinic.addDoctor(doc1);
    clinic.addDoctor(doc2);
    clinic.displayDoctorList();
```

Tests cover entity creation, appointment scheduling, status updates, and history management.

The code uses `<bits/stdc++.h>` for simplicity and pointers in `Clinic` to support polymorphism.

Test Results

The `main` function includes 10 test cases. Sample outputs demonstrate functionality:

- **Test Case 1 (Adding Doctors)**:

```
=== Test Case 1: Adding Doctors ===
========== Doctors List ==========
#1. Doctor's Information:
Name: Dr. Smith
ID: D001
Specialty: Cardiology

#2. Doctor's Information:
Name: Dr. Johnson
ID: D002
Specialty: Neurology
```

  Validates `Doctor` creation and `Clinic::addDoctor`.

- **Test Case 4 (Adding Medical History)**:

```
=== Test Case 4: Adding Medical History ===
========== Medical history of John Doe (ID: P001) ==========
#1. 2025-09-01 - Diagnosed with hypertension

========== Medical history of Jane Smith (ID: P002) ==========
#1. 2025-08-01 - Blood sugar test
```

  Confirms `addHistory` and `showHistory` functionality.

- **Test Case 5 (Scheduling via Patient)**:

```
=== Test Case 5: Scheduling Appointment via Patient ===
Patient John Doe scheduled appointment on 2025-09-12 at 09:00 for Follow-up.
Chronic patient requires regular check-up...!
Appointment set on 2025-09-13 at 11:00 for Blood test (Condition: Diabetes).
```

  Demonstrates polymorphism, with `ChronicPatient` adding specialized output.

These results validate entity management, polymorphic behavior, and data consistency.

LLM Usage

I wrote all the code myself, ensuring full understanding and personalization. ChatGPT was used solely to provide ideas, verify syntax, and suggest comments to improve code clarity. Specific interactions included:

- **Idea Generation**: I asked ChatGPT for suggestions on methods for the `Appointment` class, such as canceling or completing appointments. It proposed ideas like `cancelAppointment` and `completeAppointment`, which I adapted into `cancel` and `complete` methods to fit the system's design, ensuring they integrated with the `status` attribute.
- **Syntax Verification**: I used ChatGPT to confirm the correctness of C++ syntax, particularly for virtual functions and destructors. For example, I asked, "Is my virtual destructor in the Patient class correct?" ChatGPT verified the syntax and suggested adding it to ensure proper cleanup of derived classes, which I implemented.
- **Code Comments**: I requested suggestions for clear and concise comments to enhance readability. ChatGPT provided examples of descriptive comments in Vietnamese, which I adapted to document each class and method, ensuring clarity for both myself and readers.