Clinic Management System Documentation
Author: Tran Le Dung, Student ID: 24110084

1 Object-Oriented Analysis (OOA)

The clinic management system manages doctors, patients, appointments, and prescriptions in a small clinic. The OOA identifies key entities and their relationships:

- **Entities**: Doctors (provide care), Patients (receive care, with a subtype for chronic conditions), Appointments (link doctors and patients), Medicines (prescribed drugs), Prescriptions (list of medicines for a patient), and Clinic (central management).
- **Relationships**: A Clinic contains multiple Doctors, Patients, and Appointments. An Appointment links one Doctor to one Patient. A Prescription, issued by a Doctor for a Patient, contains multiple Medicines with quantities.
- **Behaviors**: Doctors and Patients manage attributes (name, ID, etc.) and display information. Patients track medical history. Appointments handle status updates. Prescriptions manage medicines and compute costs.
- **Constraints**: Prevent duplicate medicines in prescriptions, ensure proper memory management, and support chronic patients with specialized scheduling.

The system emphasizes simplicity, extensibility, and robust memory handling for small-scale clinic operations.

2 Class Design and Inheritance

The system uses C++ classes to model entities with clear responsibilities:

- **Doctor**: Stores name, ID, and specialty with getters, setters, and `displayInfo`. No inheritance, as its standalone.
- **Patient**: Manages name, ID, age, and a `vector<string>` for medical history. Includes `addHistory`, `removeHistory`, and a virtual `scheduleAppointment`.
- **ChronicPatient**: Inherits from `Patient`, adding `conditionType` and `lastCheckup`. Overrides `scheduleAppointment` for chronic-specific behavior.
- **Appointment**: Links a `Doctor*` and `Patient*` with attributes (ID, date, time, reason, status). Supports status updates (e.g., cancel, complete).
- **Clinic**: Manages `vectors` of `Patient*`, `Doctor*`, and `Appointment*`. Provides add/display methods and index-based selection. Its destructor ensures memory cleanup.
- **Medicine**: Stores name, dosage, and price.
- **PrescribedMedicine**: Associates a `Medicine*` with a quantity, calculating total cost.

- **Prescription**: Links a `Doctor*` and `Patient*`, managing a `vector<PrescribedMedicine*>`. Prevents duplicate medicines and tracks total cost.

Inheritance Rationale: `ChronicPatient` inherits from `Patient` to enable polymorphic `scheduleAppointment` behavior, supporting chronic-specific logic without code duplication. Other classes remain standalone for simplicity.

3 Code Walkthrough

Key components of the C++ code (`smallclinic.cpp`) include:

- **Patient and ChronicPatient**: `Patient` uses a `vector<string>` for medical history, with `addHistory` and `removeHistory`. `ChronicPatient` overrides `scheduleAppointment` to append "[Chronic]" to history entries, e.g., `addHistory(date + " - " + time + ":  " + reason + " [Chronic]")`.

- **Prescription Management**: `Prescription` uses `hasMedicine` to prevent duplicates. `addMedicine` updates `totalCost` and deletes duplicate `Medicine` objects to avoid leaks.

- **Clinic Memory Management**: The `Clinic` destructor deletes all `Patient*`, `Doctor*`, and `Appointment*` objects, ensuring no memory leaks.

- **Test Case 12 (New)**: Creates a prescription for the chronic patient (`Jane Smith`), adds two medicines (Metformin, Insulin), removes one, and displays the result. This tests `Prescription` integration with `ChronicPatient`.

The code uses `<bits/stdc++.h>` for brevity and standard C++ libraries (`vector`, `string`). It compiles and runs without errors, with proper memory management verified through destructors.

4 Test Results

The `main` function in `smallclinic.cpp` executes 12 test cases. Sample outputs include:

- **Test Case 1 (Adding Doctors)**: "Name: Dr. Smith, ID: D001, Specialty: Cardiology", confirming `Doctor` creation and `Clinic` storage.

- **Test Case 5 (Scheduling via Patient)**: For `ChronicPatient`, "Chronic patient requires regular check-up...! Appointment set on 2025-09-13 at 11:00 for Blood test (Condition: Diabetes)", showing polymorphism.

- **Test Case 11 (Prescription)**: "Medicine: Paracetamol | Dosage: 500mg x 2/day | Unit Price: 20000 VND | Quantity: 10 | Total: 200000 VND", with duplicate rejection ("Error: Medicine 'Paracetamol' already exists").

- **Test Case 12 (Prescription for Chronic Patient)**:

```
=== Test Case 12: Prescription for Chronic Patient ===
Medicine 'Metformin' added successfully.
```

```
Medicine 'Insulin' added successfully.
========== Prescription PR002 ==========
Doctor: Dr. Johnson
Patient: Jane Smith
Medicines List:
- Metformin | Dosage: 1000mg x 1/day | Unit Price: 30000 VND | Quantity: 30 | Total
- Insulin | Dosage: 10 units/day | Unit Price: 100000 VND | Quantity: 5 | Total: 50
Total Cost: 1400000 VND
Remove Insulin: Removed
========== Prescription PR002 ==========
Doctor: Dr. Johnson
Patient: Jane Smith
Medicines List:
- Metformin | Dosage: 1000mg x 1/day | Unit Price: 30000 VND | Quantity: 30 | Total
Total Cost: 900000 VND
```

This demonstrates prescription management for a chronic patient, including adding and removing medicines.

All test cases, including Test Case 12, execute successfully, confirming robust functionality and memory management.

5 LLM Usage

I used Grok (created by xAI) to brainstorm class methods and test cases. For example, I prompted: "Suggest methods for an Appointment class in a clinic system." Grok suggested `cancel`, `complete`, and `updateStatus`, which I adapted for consistency. For Test Case 12, I prompted: "Suggest a test case for a prescription involving a chronic patient in a clinic system." Grok proposed creating a prescription with multiple medicines and removing one, which I implemented to test `Prescription` with `ChronicPatient`. I wrote all code myself, using the LLM only for ideas and validation. No specific prompt/response pairs are included as appendices, as they were integrated iteratively.