**National University of Sciences and Technology (NUST)**



**School of Electrical Engineering and Computer Science (SEECS)**

**Software Design and Architecture**

|  |  |
| --- | --- |
| **Group Members** | **CMS ID** |
| Ayesha Siddiqa | 407198 |
| Sameed Ilyas | 426125 |
| Amna Ahmed | 408099 |

**Submitted to: Zunera Zahid**

**Abstract**: This project entails a Hospital Management System, designed to facilitate various tasks within a hospital setting. It caters to multiple stakeholders, including doctors, nurses, administrative staff, and patients. Each stakeholder has distinct responsibilities and functionalities within the system. For instance, doctors can access patient records and prescribe medications, while nurses can manage patient care tasks and schedule appointments. The system offers features such as appointment scheduling, medical record management, and billing.

**Code before Refactoring:**

void patient::addPerson()

{

    //getting the basic details of patient from the user side;

    person::addPerson();

    //getting patient specific details;

    while (true) {

        try {

            cout << "Enter the height of the patient (in cms):\n";

            height=0;

            if (height < 0|| !(cin >> height)) {

                throw invalid\_argument("Enter valid height.");

            }

            break; // Break out of the loop if the input is valid

        } catch (exception& e) {

            cout << "Exception occurred: " << e.what() << endl;

            cin.clear();              // Clear error flags

            cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Ignore remaining input

        }

    }

    while (true) {

        try {

            cout << "Enter the weight of the patient (in pounds):\n";

           // cin >> weight;

            weight=0;

            if (weight < 0|| !(cin >> weight)) {

                throw invalid\_argument("Enter a Valid weight.");

            }

            break; // Break out of the loop if the input is valid

        } catch (exception& e) {

            cout << "Exception occurred: " << e.what() << endl;

            cin.clear();              // Clear error flags

            cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Ignore remaining input

        }

    }

    char tt;

    cout << "\nIs the patient being hospitalized? (Y = Yes || N = No)\n";

    cin >> tt;

    while (tt != 'Y' && tt != 'N')

        cout << "Y or N?\n", cin >> tt;

    hospitalized = (tt == 'Y');

    if (hospital::patientsList.rbegin() != hospital::patientsList.rend())

        id = ((hospital::patientsList.rbegin())->first) + 1;

    else

        id = 1;

    hospital::patientsList[id] = \*this;

    cout << "\n"

         << firstName << " " << lastName << " registered successfully!\n";

    cout << "Their ID is: " << id << "\n";

    return;

}

**Code after Refactoring:**

void patient::addPerson()

{

    // Getting the basic details of the patient

    person::addPerson();

    // Getting patient-specific details

    getPatientDetails();

    // Adding the patient to the hospital's list

    addToHospitalList();

    cout << "\n" << firstName << " " << lastName << " registered successfully!\n";

    cout << "Their ID is: " << id << "\n";

}

void patient::getPatientDetails()

{

    while (true) {

        try {

            cout << "Enter the height of the patient (in cms):\n";

            if (!(cin >> height) || height < 0) {

                throw invalid\_argument("Enter a valid height.");

            }

            break; // Break out of the loop if the input is valid

        } catch (exception& e) {

            cout << "Exception occurred: " << e.what() << endl;

            cin.clear();              // Clear error flags

            cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Ignore remaining input

        }

    }

    while (true) {

        try {

            cout << "Enter the weight of the patient (in pounds):\n";

            if (!(cin >> weight) || weight < 0) {

                throw invalid\_argument("Enter a valid weight.");

            }

            break; // Break out of the loop if the input is valid

        } catch (exception& e) {

            cout << "Exception occurred: " << e.what() << endl;

            cin.clear();              // Clear error flags

            cin.ignore(numeric\_limits<streamsize>::max(), '\n'); // Ignore remaining input

        }

    }

    char tt;

    cout << "\nIs the patient being hospitalized? (Y = Yes || N = No)\n";

    cin >> tt;

    while (tt != 'Y' && tt != 'N')

        cout << "Y or N?\n", cin >> tt;

    hospitalized = (tt == 'Y');

}

void patient::addToHospitalList()

{

    if (hospital::patientsList.rbegin() != hospital::patientsList.rend())

        id = ((hospital::patientsList.rbegin())->first) + 1;

    else

        id = 1;

    hospital::patientsList[id] = \*this;

}

**Explanation:**

* **Separation of Concerns:** The original code had a single addPerson() method in the patient class that handled multiple responsibilities, including getting patient details, adding the patient to the hospital list, and displaying registration information. The refactored code separates these concerns into three distinct functions: getPatientDetails(), addToHospitalList(), and the modified addPerson() method.
* **Improved Function Naming:** The refactored code uses more descriptive function names (getPatientDetails() and addToHospitalList()) that clearly indicate their purpose. This improves code readability and makes it easier for developers to understand the code's functionality.
* **Single Responsibility Principle (SRP):** Each function now follows the SRP more closely by focusing on a single task. getPatientDetails() is responsible for getting specific patient details, addToHospitalList() is responsible for adding the patient to the hospital's list, and the modified addPerson() method orchestrates these tasks.
* **Error Handling:** Error handling has been improved in the getPatientDetails() function by checking for valid inputs (height and weight) and throwing exceptions with informative error messages. This enhances the robustness of the code and provides better user feedback in case of invalid inputs.
* **Code Reusability:** The separation of concerns and improved function naming also promotes code reusability. Each function can be used independently and can be easily modified or extended without affecting other parts of the codebase.

**Code before Refactoring:**

void appointment::book()

{

    if (hospital::appointmentsList.size() >= 8 \* hospital::doctorsList.size())

    {

        cout << "\n\nSorry, no doctor is available for appointment today!\n\n";

        return;

    }

    cout << "\n\nIs the patient already registered (Y : Yes || N : No)?\n";

    char ans;

    cin >> ans;

    while (ans != 'Y' && ans != 'y' && ans != 'N'&& ans != 'n')

    {

        cout << "Y or N?\n";

        cin >> ans;

    }

    if (ans == 'N' ||ans == 'n' )

    {

        cout << "Register the patient:\n";

        P.addPerson();

    }

    else

    {

        cout << "Search for the required patient:\n\n";

        ans = 'Y';

        while (ans == 'Y')

        {

            P.getDetails();

            ans = 'K';

            if (P.id == -1)

            {

                cout << "Try again (Y : Yes || N : No)?\n";

                cin >> ans;

                while (ans != 'Y' && ans != 'y' && ans != 'N'&& ans != 'n')

                {

                    cout << "Y or N?\n";

                    cin >> ans;

                }

            }

        }

        if (ans == 'N')

        {

            return;

        }

    }

    cout << "\n\nNow, search for the required doctor:\n";

    ans = 'Y';

    while (ans == 'Y')

    {

        D.getDetails();

        ans = 'K';

        if (D.id == -1)

        {

            cout << "Try again (Y : Yes || N : No)?\n";

            cin >> ans;

            while (ans != 'Y' && ans != 'y' && ans != 'N'&& ans != 'n')

            {

                cout << "Y or N?\n";

                cin >> ans;

            }

        }

        else if (D.appointmentsBooked >= 8)

        {

            cout << "Sorry, selected doctor has no free slot left for the day!\n";

            cout << "Search again (Y : Yes || N : No)?\n";

            cin >> ans;

            while (ans != 'Y' && ans != 'y' && ans != 'N'&& ans != 'n')

            {

                cout << "Y or N?\n";

                cin >> ans;

            }

        }

    }

    if (ans == 'N'|| ans=='n')

    {

        cout<<"appointment not booked"<<endl;

        return;

    }

    if (hospital::appointmentsList.rbegin() != hospital::appointmentsList.rend())

        id = ((hospital::appointmentsList.rbegin())->first) + 1;

    else

        id = 1;

    hh = 9 + D.appointmentsBooked;

    hospital::appointmentsList[id] = \*this;

    hospital::doctorsList[D.id].appointmentsBooked++;

    cout << "\nAppointment of patient " + P.firstName + " " + P.lastName + " with doctor "

         << D.firstName << " " << D.lastName << " booked successfully!\n";

    printDetails();

    return;

}

**Code after Refactoring:**

void appointment::book() {

    if (isDoctorUnavailable()) {

        cout << "\n\nSorry, no doctor is available for appointment today!\n\n";

        return;

    }

    Patient patient;

    Doctor doctor;

    if (!isPatientRegistered(patient)) {

        cout << "Register the patient:\n";

        patient.addPerson();

    }

    if (!selectDoctor(doctor)) {

        cout << "Appointment not booked.\n";

        return;

    }

    if (!bookAppointment(patient, doctor)) {

        cout << "Failed to book appointment.\n";

        return;

    }

    cout << "\nAppointment of patient " + patient.firstName + " " + patient.lastName + " with doctor "

         << doctor.firstName << " " << doctor.lastName << " booked successfully!\n";

    printDetails();

}

bool appointment::isDoctorUnavailable() const {

    return hospital::appointmentsList.size() >= 8 \* hospital::doctorsList.size();

}

bool appointment::isPatientRegistered(Patient& patient) {

    char ans;

    cout << "\n\nIs the patient already registered (Y : Yes || N : No)?\n";

    cin >> ans;

    while (ans != 'Y' && ans != 'y' && ans != 'N' && ans != 'n') {

        cout << "Y or N?\n";

        cin >> ans;

    }

    if (ans == 'N' || ans == 'n') {

        return false;

    }

    else {

        cout << "Search for the required patient:\n\n";

        ans = 'Y';

        while (ans == 'Y') {

            patient.getDetails();

            ans = 'K';

            if (patient.id == -1) {

                cout << "Try again (Y : Yes || N : No)?\n";

                cin >> ans;

                while (ans != 'Y' && ans != 'y' && ans != 'N' && ans != 'n') {

                    cout << "Y or N?\n";

                    cin >> ans;

                }

            }

        }

        if (ans == 'N') {

            return false;

        }

        return true;

    }

}

bool appointment::selectDoctor(Doctor& doctor) {

    char ans = 'Y';

    cout << "\n\nNow, search for the required doctor:\n";

    while (ans == 'Y') {

        doctor.getDetails();

        ans = 'K';

        if (doctor.id == -1) {

            cout << "Try again (Y : Yes || N : No)?\n";

            cin >> ans;

            while (ans != 'Y' && ans != 'y' && ans != 'N' && ans != 'n') {

                cout << "Y or N?\n";

                cin >> ans;

            }

        }

        else if (doctor.appointmentsBooked >= 8) {

            cout << "Sorry, selected doctor has no free slot left for the day!\n";

            cout << "Search again (Y : Yes || N : No)?\n";

            cin >> ans;

            while (ans != 'Y' && ans != 'y' && ans != 'N' && ans != 'n') {

                cout << "Y or N?\n";

                cin >> ans;

            }

        }

    }

    if (ans == 'N' || ans == 'n') {

        cout << "Appointment not booked.\n";

        return false;

    }

    return true;

}

bool appointment::bookAppointment(const Patient& patient, Doctor& doctor) {

    if (hospital::appointmentsList.rbegin() != hospital::appointmentsList.rend())

        id = ((hospital::appointmentsList.rbegin())->first) + 1;

    else

        id = 1;

    hh = 9 + doctor.appointmentsBooked;

    hospital::appointmentsList[id] = \*this;

    hospital::doctorsList[doctor.id].appointmentsBooked++;

    return true;

}

**Explanation:**

* **Separation of Concerns:** The original book() function in the appointment class handled multiple responsibilities, including checking doctor availability, patient registration, doctor selection, and appointment booking. The refactored code divides these responsibilities into separate functions, each focusing on a specific task: checking doctor availability (isDoctorUnavailable()), checking patient registration status (isPatientRegistered()), selecting a doctor (selectDoctor()), and booking the appointment (bookAppointment()).
* **Improved Function Naming:** The refactored code uses descriptive function names that clearly indicate their purpose (isDoctorUnavailable(), isPatientRegistered(), selectDoctor(), bookAppointment()). This makes the code more readable and understandable.
* **Single Responsibility Principle (SRP):** Each function in the refactored code follows the SRP more closely by focusing on a single task. For example, isDoctorUnavailable() checks if a doctor is available for an appointment, isPatientRegistered() checks if the patient is already registered, selectDoctor() handles the selection of a doctor, and bookAppointment() books the appointment.
* **Error Handling:** Error handling has been improved in the isPatientRegistered() and selectDoctor() functions by checking for valid user inputs and providing appropriate feedback. This enhances the robustness of the code and improves the user experience.
* **Code Reusability:** The separation of concerns and improved function naming also promotes code reusability. Each function can be used independently and can be easily modified or extended without affecting other parts of the codebase.