Project Documentation: Medical Clinic Management Using Design Patterns

Introduction

The project is a Medical Clinic Management System that uses various software design patterns to make the development process more efficient and maintainable. The system aims to provide an interface for adding patients, scheduling appointments, and viewing medical records in an organized and effective manner using appropriate design patterns.

---

1. Design Patterns Used

In this project, the following design patterns have been implemented:

Singleton Pattern

Factory Method Pattern

Proxy Pattern

Command Pattern

Builder Pattern

1.1 Singleton Pattern

Purpose: Ensures that a class has only one instance and provides a global access point to it.

Components:

PatientDatabaseManager: Manages the patient database.

AppointmentScheduler: Schedules patient appointments.

Reason for Use: We need only one instance to manage the patients and appointments across the system. Therefore, the Singleton pattern ensures that there is only one object to handle this functionality.

Implementation:

java

Copy code

```
public class PatientDatabaseManager {
    private static PatientDatabaseManager instance;

    private PatientDatabaseManager() {
        // Initialization code
    }

    public static synchronized PatientDatabaseManager getInstance() {
        if (instance == null) {
            instance = new PatientDatabaseManager();
        }
        return instance;
    }

    // Methods for managing patients
}
```

1.2 Factory Method Pattern

Purpose: Allows for creating objects without specifying the exact class of the object that will be created.

Components:

DoctorFactory: Creates doctor objects based on specialty.

Reason for Use: We need to create different types of doctors based on their specialties (such as Cardiologist, Neurologist, General Practitioner). The Factory pattern helps us create doctors without having to specify the exact type every time.

Implementation:

java

Copy code

```
public class DoctorFactory {
    public Doctor createDoctor(String specialty) {
        if ("Cardiologist".equalsIgnoreCase(specialty)) {
            return new Cardiologist();
        } else if ("Neurologist".equalsIgnoreCase(specialty)) {
            return new Neurologist();
        } else {
            return new GeneralPractitioner();
        }
    }
}
```

1.3 Proxy Pattern

Purpose: Provides an object representing another object to control access to it, often for reasons such as security or lazy initialization.

Components:

PatientRecordAccess: A proxy for accessing patient medical records, where user permissions are verified before access.

Reason for Use: Only doctors should have access to patient medical records, so we used the Proxy pattern to verify the user's role before allowing access to the records.

Implementation:

java

Copy code

```
public class ProxyPatientRecord implements PatientRecordAccess {
```

```
    private Patient patient;
    private String userRole;

    public ProxyPatientRecord(Patient patient, String userRole) {
        this.patient = patient;
        this.userRole = userRole;
    }

    @Override
    public void accessPatientRecord() {
        if ("Doctor".equalsIgnoreCase(userRole)) {
            patient.viewRecord();
        } else {
            System.out.println("Access denied.");
        }
    }
}
```

1.4 Command Pattern

Purpose: Separates the request for an action from the object that performs the action, allowing for parameterized actions and decoupling invocations from the actions themselves.

Components:

AddPatientCommand: Adds a new patient to the database.

ScheduleAppointmentCommand: Schedules an appointment for a patient.

Reason for Use: We need to decouple the user interface (buttons) from the logic that executes the actual operations (such as adding a patient or scheduling an appointment).

Implementation:

java

Copy code

```
public class AddPatientCommand implements Command {

    private PatientDatabaseManager patientDatabaseManager;

    private Patient patient;



    public AddPatientCommand(PatientDatabaseManager patientDatabaseManager, Patient patient) {

        this.patientDatabaseManager = patientDatabaseManager;

        this.patient = patient;

    }



    @Override

    public void execute() {

        patientDatabaseManager.addPatient(patient);

    }

}
```

1.5 Builder Pattern

Purpose: Allows the creation of complex objects step by step, without requiring a large number of constructors or forcing clients to provide many parameters.

Components:

AppointmentBuilder: Used to build appointments for patients in a flexible and organized manner.

Reason for Use: An Appointment contains multiple attributes such as the patient, doctor specialty, and time slot. Using the Builder pattern allows us to create an appointment step by step without passing all the parameters through the constructor.

Implementation:

java

Copy code

```
public class AppointmentBuilder {

    private Patient patient;

    private String doctorSpecialty;

    private String timeSlot;


    public AppointmentBuilder setPatient(Patient patient) {

        this.patient = patient;

        return this;

    }


    public AppointmentBuilder setDoctorSpecialty(String doctorSpecialty) {

        this.doctorSpecialty = doctorSpecialty;

        return this;

    }


    public AppointmentBuilder setTimeSlot(String timeSlot) {

        this.timeSlot = timeSlot;
```

```
        return this;
    }

    public Appointment build() {
        return new Appointment(patient, doctorSpecialty, timeSlot);
    }
}
```

Usage of Builder to create an appointment:

java

Copy code

```
Appointment appointment = new AppointmentBuilder()
    .setPatient(selectedPatient)
    .setDoctorSpecialty(selectedSpecialty)
    .setTimeSlot(timeSlot)
    .build();
```

---

2. Project Structure

The project structure consists of key classes that manage the patients, appointments, doctors, and medical records:

PatientDatabaseManager: Manages the patient database.

AppointmentScheduler: Schedules appointments.

Doctor and DoctorFactory: Creates doctors based on their specialty.

Patient: Represents a patient.

Appointment: Represents an appointment with a patient.

PatientRecordAccess and ProxyPatientRecord: Handles access to medical records.

---

3. User Interface

The user interface is built using Swing components, providing buttons for:

Adding a patient.

Scheduling an appointment.

Viewing medical records.

Event handling is done through ActionListeners that execute commands like adding a patient or scheduling an appointment using the Command Pattern.

---

4. Interaction Between Patterns

Singleton:

PatientDatabaseManager and AppointmentScheduler are used as singletons, ensuring that only one instance manages the patient database and appointment schedules across the application.

Factory:

The Factory pattern is used to create doctors based on their specialties.

Proxy:

The Proxy pattern ensures that only doctors can access patient medical records.

Command:

The Command pattern decouples the user interface from the execution logic of operations such as adding a patient or scheduling an appointment.

Builder:

The Builder pattern is used to create appointments in a flexible and step-by-step manner without needing to pass a large number of parameters through the constructor.

---

5. Conclusion

Using these design patterns in the Medical Clinic Management project helps to keep the code clean, modular, and easier to maintain. The Builder Pattern improves how complex objects like appointments are created, while other patterns like Singleton, Factory, Proxy, and Command enhance data flow, security, and flexibility within the application. This structured approach ensures that the system remains scalable and maintainable over time.