Medical Clinic Management System - Documentation

📋 Project Overview

A Java desktop application that demonstrates **6 design patterns** in a real-world medical clinic scenario. The system manages patients, appointments, medical records, insurance, and notifications with a GUI interface.

Tech Stack: Java, Java Swing

🎯 Design Patterns & Justifications

1️⃣ Singleton Pattern

Purpose: Ensure only ONE instance of critical system components exists.

Classes:

- `PatientDatabaseManager` - Manages all patient records
- `AppointmentSchedulingSystem` - Manages all appointments
- *Why Singleton? **
- ✅ **Prevents duplicate databases** - All parts of the system access the same patient data
- ✅ **Centralized control** - One appointment scheduler prevents double-booking
- ✅ **Thread-safe** - Uses `synchronized` to handle multiple users
- ✅ **Memory efficient** - Only one instance in memory

Key Methods:

- `getInstance()` - Returns the single instance
- `addPatient(Patient)` - Adds patient to database
- `bookAppointment(Appointment)` - Books appointment if slot available

2️⃣ Factory Pattern

Purpose: Create objects without specifying their exact class type.

Classes:

Medical Records:

- `MedicalRecord` (interface)
- `MedicalRecordFactory` - Creates records
- `Prescription` - Medication prescriptions
- `LabResult` - Laboratory test results
- `PatientHistory` - Patient medical history

Doctors:

- `Doctor` (interface)
- `DoctorFactory` - Creates doctors
- `Cardiologist` - Heart specialist
- `Neurologist` - Brain/nerve specialist
- `GeneralPractitioner` - General health

Why Factory?

- ✅ **Flexible object creation** - Add new record types without changing code
- ✅ **Simplifies code** - Client doesn't need to know which class to instantiate
- ✅ **Easy to extend** - Add new doctor specializations easily
- ✅ **Centralized logic** - All creation logic in one place

Usage Example:

```
MedicalRecord record = MedicalRecordFactory.createRecord("Prescription");
Doctor doctor = DoctorFactory.createDoctor("Cardiologist");
```

3️⃣ Decorator Pattern

Purpose: Add features to appointments dynamically without modifying the base class.

Classes:

- `Appointment` (interface) - Base contract
- `BasicAppointment` - Basic consultation ($100)
- `AppointmentDecorator` (abstract) - Base decorator
- `LabTestDecorator` - Adds lab test (+$50)
- `XRayDecorator` - Adds X-ray (+$150)
- `MRIScanDecorator` - Adds MRI scan (+$500)

Why Decorator?

- ✅ **Flexible pricing** - Combine services in any order
- ✅ **No class explosion** - Don't need separate classes for every combination
- ✅ **Open/Closed Principle** - Add new services without modifying existing code
- ✅ **Runtime flexibility** - Add/remove services dynamically

Usage Example:

```
Appointment appt = new BasicAppointment();          // $100
appt = new LabTestDecorator(appt);                  // $150
appt = new XRayDecorator(appt);                     // $300
appt = new MRIScanDecorator(appt);                  // $800
```

4️⃣ Observer Pattern

Purpose: Notify multiple parties automatically when an event occurs.

Classes:

- `Observer` (interface) - Observer contract
- `AppointmentSubject` - Manages observers and sends notifications
- `PatientObserver` - Receives patient notifications

- `DoctorObserver` - Receives doctor notifications
- `ReceptionistObserver` - Receives admin notifications

Why Observer?

- ✅ **Automatic notifications** - No need to manually notify each party
- ✅ **Loose coupling** - Subject doesn't need to know about observers
- ✅ **Easy to extend** - Add new observer types (e.g., SMS, Email) easily
- ✅ **Real-time updates** - Everyone stays informed immediately

How it works:

1. Observers register with the subject (`attach()`)
2. When appointment is booked, subject calls `notifyObservers()`
3. All observers receive the notification automatically

5️⃣ Adapter Pattern

Purpose: Convert incompatible interfaces to work together.

Classes:

- `InsuranceService` (interface) - Modern insurance interface
- `InsuranceAdapter` - Converts legacy format to modern format
- `LegacyInsuranceSystem` - Old system with different data format
- `InsuranceCoverage` - Modern coverage data model

Why Adapter?

- ✅ **Legacy integration** - Works with old insurance systems without changing them
- ✅ **Data format conversion** - Converts `"ID|Percentage|Limit"` → Object
- ✅ **Protects existing code** - No need to modify legacy system
- ✅ **Future-proof** - Easy to swap legacy system later

Data Flow:

```
Legacy System → Returns:  "P001|75|10000"
     ↓
  Adapter → Converts to: InsuranceCoverage object
     ↓
Modern System → Uses: coverage.getCoveragePercentage()
```

6️⃣ Proxy Pattern

Purpose: Control access to sensitive data and log all operations.

Classes:

- `MedicalRecordAccess` (interface) - Access contract
- `MedicalRecordProxy` - Controls access and logs
- `RealMedicalRecordAccess` - Actual record access

Why Proxy?

- ✅ **Security & Audit** - Logs who accessed what and when
- ✅ **Access control** - Can add permission checks before allowing access
- ✅ **Lazy loading** - Creates real object only when needed
- ✅ **HIPAA compliance** - Medical records require access tracking

Logging Example:

```
[2025-12-18 10:30:45] User: Dr. Smith | Operation: VIEW | Record: REC001
[2025-12-18 10:31:12] User: Dr. Smith | Operation: VIEW | Record: REC002
```

📂 Project Structure

```
src/
├── models/
│   ├── Patient.java            # Patient data model
│   └── Appointment.java        # Appointment data model
```

```
│
├── singleton/
│   ├── PatientDatabaseManager. java      # Shared patient database
│   └── AppointmentSchedulingSystem.java  # Shared appointment scheduler
│
├── factory/
│   ├── MedicalRecord.java       # Medical record interface
│   ├── MedicalRecordFactory.java # Creates medical records
│   ├── Prescription.java        # Prescription record
│   ├── LabResult.java           # Lab result record
│   ├── PatientHistory.java      # Patient history record
│   ├── Doctor.java              # Doctor interface
│   ├── DoctorFactory.java       # Creates doctors
│   ├── Cardiologist.java        # Heart specialist
│   ├── Neurologist.java         # Brain specialist
│   └── GeneralPractitioner.java  # General doctor
│
├── decorator/
│   ├── Appointment. java         # Appointment interface
│   ├── BasicAppointment.java     # Basic consultation
│   ├── AppointmentDecorator.java # Base decorator
│   ├── LabTestDecorator.java     # Lab test add-on
│   ├── XRayDecorator.java        # X-ray add-on
│   └── MRIScanDecorator.java     # MRI add-on
│
├── observer/
│   ├── Observer.java            # Observer interface
│   ├── AppointmentSubject.java   # Manages notifications
│   ├── PatientObserver.java      # Patient notifications
│   ├── DoctorObserver. java      # Doctor notifications
│   └── ReceptionistObserver. java # Admin notifications
│
├── adapter/
│   ├── InsuranceService.java     # Modern insurance interface
│   ├── InsuranceAdapter.java     # Converts legacy data
│   ├── LegacyInsuranceSystem.java # Old insurance system
```

```
│   └── InsuranceCoverage.java    # Coverage data model
│
├── proxy/
│   ├── MedicalRecordAccess.java  # Access interface
│   ├── MedicalRecordProxy.java   # Logs access
│   └── RealMedicalRecordAccess.java # Actual access
│
└── MedicalClinicGUI.java         # Main GUI application
```

🔑 Key Class Descriptions

Models

Patient

- Stores patient information (ID, name, age, phone)
- Used throughout the system for patient identification

Appointment (in models package)

- Stores appointment details (ID, patient, doctor, time slot)
- Used by AppointmentSchedulingSystem

Singleton Classes

PatientDatabaseManager

- **Responsibility:** Centralized patient data storage
- **Key Feature:** Thread-safe singleton with synchronized methods
- **Methods:** `getInstance()`, `addPatient()`, `getAllPatients()`, `getPatientCount()`

AppointmentSchedulingSystem

- **Responsibility:** Manages appointment bookings and time slots
- **Key Feature:** Prevents double-booking by tracking slot availability
- **Methods:** `bookAppointment()`, `cancelAppointment()`, `checkAvailability()`

Factory Classes

MedicalRecordFactory

- **Responsibility:** Creates different types of medical records
- **Returns:** Prescription, LabResult, or PatientHistory based on input

DoctorFactory

- **Responsibility:** Creates different specialist doctors
- **Returns:** Cardiologist, Neurologist, or GeneralPractitioner based on input

Decorator Classes

BasicAppointment

- **Cost:** $100
- **Description:** "Basic Consultation"

LabTestDecorator

- **Adds:** $50 to cost
- **Adds:** " + Lab Test" to description

XRayDecorator

- **Adds:** $150 to cost
- **Adds:** " + X-Ray" to description

MRIScanDecorator

- **Adds:** $500 to cost
- **Adds:** " + MRI Scan" to description

Observer Classes

AppointmentSubject

- **Responsibility:** Manages list of observers
- **Methods:** `attach()`, `detach()`, `notifyObservers()`

PatientObserver, DoctorObserver, ReceptionistObserver

- **Responsibility:** Receive and display notifications
- **Method:** `update(String message)` - Called when notification sent

Adapter Classes

InsuranceAdapter

- **Responsibility:** Converts legacy format to modern format
- **Conversion:** `"P001|75|10000"` → `InsuranceCoverage(P001, 75, 10000)`

LegacyInsuranceSystem

- **Responsibility:** Simulates old insurance system
- **Format:** Returns pipe-separated string `"ID|Coverage%|Limit"`

Proxy Classes

MedicalRecordProxy

- **Responsibility:** Logs all access to medical records
- **Log Format:** `[Timestamp] User: X | Operation: Y | Record: Z`
- **Feature:** Lazy initialization of real access object

✅ Summary

Pattern	Problem Solved	Main Benefit
Singleton	Multiple database instances	One shared source of truth
Factory	Complex object creation	Flexible and extendable
Decorator	Rigid appointment pricing	Dynamic service combinations
Observer	Manual notifications	Automatic updates to all parties
Adapter	Incompatible legacy system	Seamless integration
Proxy	Untracked data access	Security and audit trail

End of Documentation