Javadoc : <file:///Users/gargamel724/Desktop/SC2002SCSDGrp4/HMS%20javadoc/package-summary.html>

Github repo: https://github.com/xushuwei281/SC2002-Hospital-Management-System

**Introduction**

The Command Line Interface (CLI) Hospital Management System (HMS) is a modular and object-oriented application developed to streamline the administrative, medical, and logistical operations of a hospital. Designed with scalability, maintainability, and efficiency in mind, the system adheres to core object-oriented programming (OOP) principles and design patterns, ensuring it remains adaptable to evolving requirements.

Key assumptions have shaped the development of this HMS:

1. **Patient-Centric Approach:** Patients access appointment scheduling by viewing available slots and selecting doctors via their unique IDs, promoting seamless and error-free interactions.
2. **Conflict Management:** Given the n-to-1 relationship between patients and doctors, doctors can confirm or reschedule appointments to optimize slot availability, ensuring service quality and minimizing scheduling conflicts.
3. **Role-Specific Functionalities:** Users—including patients, doctors, pharmacists, and administrators—interact with role-tailored menus to access specific features, enhancing usability and reducing complexity.
4. **Flexible Staff Management:** To accommodate future expansions, the system distinctly separates administrative views for doctors and pharmacists, providing administrators with intuitive controls for managing diverse staff roles.
5. **Security and Fair Use:** Doctors are empowered to cancel appointments as needed, safeguarding their rights and preventing system misuse.
6. **Enhanced Data Accessibility**: The system leverages Excel integration to streamline data management processes, allowing users to export or import data efficiently and ensuring compatibility with widely used software tools.

**1. Object-Oriented Principles and Concepts**

**1.1 Encapsulation**

Encapsulation hides the internal state of objects and exposes functionalities through controlled interfaces, ensuring data integrity and abstraction.

* **Patient Class**:
  + **Attributes like name, age, and medicalHistory are private**.
  + Access is controlled via **getter and setter methods**, ensuring that updates to sensitive data are validated.

public class Patient {

private String name;

private int age;

private List<String> medicalHistory;

public String getName() {

return name;

}

public void setName(String name) {

if (name != null && !name.isEmpty()) {

this.name = name;

} else {

throw new IllegalArgumentException("Name cannot be null or empty.");

}

} // Similar getters and setters for other fields

}

* **Inventory Class**:
  + Manages **stock levels and medication details**, ensuring no unauthorized access or modification.

public class Inventory {

private Map<String, Integer> stock;

public int getStock(String medicine) {

return stock.getOrDefault(medicine, 0);

}

public void updateStock(String medicine, int quantity) {

if (quantity >= 0) {

stock.put(medicine, quantity);

} else {

throw new IllegalArgumentException("Stock quantity cannot be negative.");

}

}

}

**1.2 Inheritance**

Inheritance avoids redundancy by allowing shared attributes and behaviors to be defined in a base class and reused in derived classes.

* **User Hierarchy**:
  + User is a **base class** for shared attributes like userID, password, and generic methods like login().
  + Specific roles like Doctor, Patient, Pharmacist, and Administrator **extend User** to add unique functionalities.

public abstract class User {

private String userId;

private String password;

public abstract void viewMenu();

}

public class Doctor extends User {

@Override

public void viewMenu() {

System.out.println("Doctor Menu Options");

}

}

**1.3 Polymorphism**

Polymorphism allows the system to handle different user roles or operations using a common interface or parent class.

* **Dynamic Menus**:
  + The viewMenu() method in the User class is overridden by subclasses to display role-specific options dynamically.

User user = new Doctor();

user.viewMenu(); // Displays "Doctor Menu Options"

* **Appointment Handling**:
  + Both Patient and Doctor interact with the Appointment class differently using polymorphic methods.

Appointment appointment = new Appointment();

patient.scheduleAppointment(appointment);

doctor.updateAppointmentOutcome(appointment);

**2. Design Principles (SOLID)**

**2.1 Single Responsibility Principle (SRP)**

**Example from AppointmentManager.java**

This class is responsible for managing appointment-related functionalities such as booking, rescheduling, and canceling appointments.

public class AppointmentManager {

private List<Appointment> appointments;

public AppointmentManager() {

this.appointments = new ArrayList<>();

}

public void addAppointment(Appointment appointment) {

appointments.add(appointment);

}

public void cancelAppointment(String appointmentId) {

appointments.removeIf(app -> app.getAppointmentId().equals(appointmentId));

}

public List<Appointment> getAppointmentsForDoctor(String doctorId) {

return appointments.stream()

.filter(app -> app.getDoctorId().equals(doctorId))

.collect(Collectors.toList());

}

}

* **Compliance with SRP**:
  + AppointmentManager is solely responsible for appointment-related operations.
  + It doesn’t handle user authentication, inventory, or other unrelated functionalities.

**2.2 Open/Closed Principle (OCP)**

**Example from User.java**

The base User class is designed to be extended by specific roles such as Doctor, Patient, Pharmacist, and Administrator.

public abstract class User {

private String userId;

private String password;

public User(String userId, String password) {

this.userId = userId;

this.password = password;

}

public abstract void displayMenu();

}

public class Doctor extends User {

public Doctor(String userId, String password) {

super(userId, password);

}

@Override

public void displayMenu() {

System.out.println("1. View Patient Records");

System.out.println("2. Update Medical Records");

}

}

* **Compliance with OCP**:
  + Adding new roles (e.g., Nurse) involves creating a new class that extends User and implements displayMenu(). No changes are required in existing code.

**2.3 Liskov Substitution Principle (LSP)**

**Example from ReplenishRequestManager.java**

The ReplenishRequestManager handles requests for inventory replenishment. It demonstrates LSP compliance by allowing any subclass of User to submit requests as long as the user has sufficient privileges.

public class ReplenishRequestManager {

private List<ReplenishRequest> requests;

public ReplenishRequestManager() {

this.requests = new ArrayList<>();

}

public void submitRequest(User user, String item, int quantity) {

if (user instanceof Pharmacist) {

requests.add(new ReplenishRequest(item, quantity));

System.out.println("Replenish request submitted for item: " + item);

} else {

throw new IllegalArgumentException("Only pharmacists can submit replenishment requests.");

}

}

}

* **Compliance with LSP**:
  + This function assumes that Pharmacist can be substituted for User without violating system correctness.
* **Improvements to make**:
  + Hardcoding checks like user instanceof Pharmacist can lead to brittle code if more roles (e.g., Inventory Manager) are added. Using an interface such as InventoryHandler would be a better solution.

**2.4 Interface Segregation Principle (ISP)**

**Example from CommonInventory.java**

The inventory management is logically separated from user-specific roles by defining specific operations that deal only with inventory management.

public class CommonInventory {

private Map<String, Integer> inventory;

public CommonInventory() {

this.inventory = new HashMap<>();

}

public void addItem(String item, int quantity) {

inventory.put(item, inventory.getOrDefault(item, 0) + quantity);

}

public void removeItem(String item, int quantity) {

if (inventory.containsKey(item) && inventory.get(item) >= quantity) {

inventory.put(item, inventory.get(item) - quantity);

} else {

throw new IllegalArgumentException("Not enough stock for item: " + item);

}

}

public int getStock(String item) {

return inventory.getOrDefault(item, 0);

}

}

* **Compliance with ISP**:
  + The CommonInventory class doesn’t enforce methods unrelated to inventory operations, adhering to ISP.
  + requestManager introduced to allow roles like Pharmacist and Administrator to interact with inventory while keeping their specific methods isolated.

**2.5. Dependency Inversion Principle (DIP)**

**Example from HospitalManagementSystem.java**

The system demonstrates DIP by depending on abstractions (User) rather than concrete implementations (Doctor, Patient).

public class HospitalManagementSystem {

private List<User> users;

public HospitalManagementSystem() {

this.users = new ArrayList<>();

}

public void addUser(User user) {

users.add(user);

}

public User authenticate(String userId, String password) {

return users.stream()

.filter(user -> user.getUserId().equals(userId) && user.verifyPassword(password))

.findFirst()

.orElseThrow(() -> new IllegalArgumentException("Invalid credentials"));

}

}

* **Compliance with DIP**:
  + HospitalManagementSystem interacts with User abstractions, making it easy to add new roles without modifying this class.

**3. UML Class Diagram**

****

**3.1 Control Classes:**

Manager classes coordinate and manage domain-specific entities, serving as control classes in the system.

1. **StaffManager.java**:
   * Manages the hospital staff, including doctors and pharmacists.
   * Functions:
     + Add, update, and remove staff members.
     + Filter staff based on attributes like role, gender, and age​.
2. **ReplenishRequestManager.java**:
   * Oversees medication inventory replenishment.
   * Functions:
     + Handles requests submitted by pharmacists for restocking low inventory.
     + Approves or denies requests​​.
3. **AppointmentManager.java**:
   * Deals with scheduling, rescheduling, and cancelling appointments.
   * Functions:
     + Tracks appointment statuses (e.g., confirmed, cancelled, completed).
     + Manages time slot availability updates​​.
4. **CommonInventory.java**:

* Manages hospital inventory, particularly medications.
* Functions:
  + Tracks stock levels.
  + Updates inventory and low-stock alerts​​.

**3.2 Utility Classes:**

Interface classes define contracts that other classes implement, promoting loose coupling and flexibility in the system.

1. **ReplenishRequest Interface**: Defines the structure for handling inventory requests.
2. **Appointment Interface**: Standardizes appointment-related functionality across roles (patients, doctors, administrators).

**3.3 Entity classes:**

The **entity classes** for the users in the Hospital Management System (HMS) represent the core data structures holding user-specific information and encapsulating user-related behavior.

**Users: Patient, Doctor, Pharmacists, Administrator**

**3.4 Boundary classes:**

The UserInterface handles user interactions, including displaying menus, capturing input, and providing feedback. It ensures intuitive and error-free communication between the user and the system.

1. **User Interaction in Main**

The main function acts as the system's entry point, presenting menus through the UserInterface, capturing user actions, and delegating tasks to relevant controllers or managers. It orchestrates the application workflow in a loop until the user exits.

1. **Excel Reader/Writer**

The ExcelReader and Writer classes manages data import/export between the system and Excel files, enabling efficient storage and retrieval of hospital data like schedules, staff details, and reports.

**Test Cases:**

**A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generated**

test case2

test case 1

A screenshot of a computer

Description automatically generated

test case3

Test case 4 , 7 and 13

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Test case 5 , 11 and 14:

Test case 6

A screenshot of a computer program

Description automatically generatedupdated slot is reflected on doctor side

A screenshot of a computer program

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

Test cases 8, 9, 15(10)A screenshot of a computer

Description automatically generated

A screenshot of a computer program

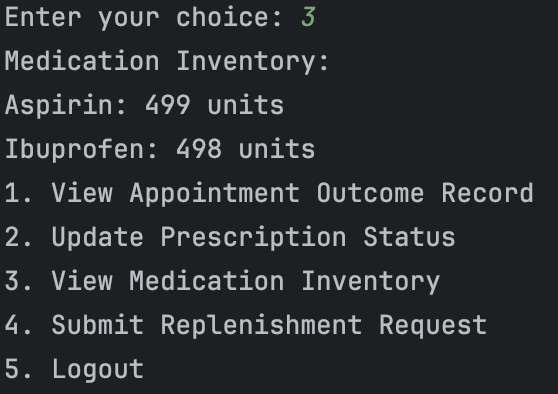
Description automatically generatedA screenshot of a computer

Description automatically generatedTest case 16, 17

A screenshot of a computer

Description automatically generatedTest case 18

Test case 20



Test case 19&23

Test case 20 & 26

A screenshot of a computer program

Description automatically generatedA computer screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

Test case 22

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

Test case 25