 **Hospital Management System**

Github link: https://github.com/aanyasan/OOPS-project---Hospital-Management-System

1. ***INTRODUCTION***

The development of the Hospital Management System (HMS), as specified in the assignment criteria was to design a system that helps in allowing various users, such as Patients, doctors, pharmacists and administrator to use the app by accessing its various features which includes patient management, staff management, appointment scheduling, inventory control and more. This project is focused on utilising the OOP concepts(Abstraction, encapsulation/information hiding, inheritance and polymorphism) and the 5 SOLID principles to create a HMS with the core of Java programming.

1. ***INITIAL APPROACH, MODIFICATIONS AND ADOPTION OF KEY/EXTRA FEATURES***

***Initial approach taken***:

Java was the programming language of use while developing the HMS. A Command Line Interface approach was taken to simply adhere to the project guidelines as the primary method of interaction. Thus, our project has incorporated a CLI-based menu system which provides the menu for various task managements.

Initially, our group had to focus mainly on ordering the given task functionalities to make the building of the app easier. We started this by trying to build a UML class diagram that split the classes into the sectors of parent class, sub classes and standalone classes. The categorising of the class in this way done by us initially is given below for reference. However, we soon realised that there were multiple loopholes in this model and the way we started working with our project.

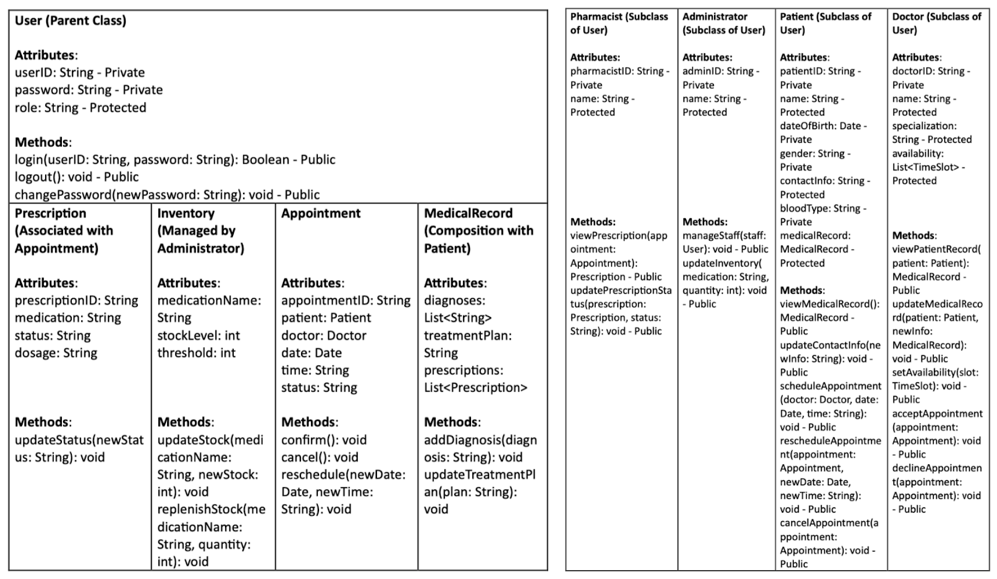
******

Figure : Initial approach to segregating the classes

***How did we progress from this initial sketch/layout and add in new key features?***

**I.**   **Adoption of ECB architecture**

In the initial model, the design was primarily focused on 4 subclasses(Pharmacist, Administrator, Patient, Doctor) to the User parent class and 4 standalone classes(Prescription, Inventory, Appointment, MedicalRecord). To make it more scalable and maintainable, the Entity-Control-Boundary (ECB) architecture was adopted. This approach allowed our system to introduce specialised Data manager classes such as (PatientDataManager, DoctorDataManager, StaffDataManager) to do their work and execute functions to enhance system efficiency and reduce function overlapping as compared to the subclasses method being used initially. This also makes the system more maintainable by showing clearer modularization which can make future modifications much easier.

To maintain the Hospital Management System, several classes were utilised. The classes were broadly split into 3 class stereotypes, namely, entity, boundary, and control classes.

**Entity classes:**

The entity classes used in the HMS include the following classes. These classes were categorised as entity classes as they contain the information with respect to the user types accessing the system to be maintained consistently. They ensure that they only concentrate on data storage and consistency, avoiding logic processing, by encapsulating attributes like staffID, specialisation, and patientID.

User: this base class contains shared attributes that include the user ID the role of the user in the HMS and also the username.

Administrator: Its status as an entity class is confirmed by the presence of attributes such as staffID, name, and staff management techniques.

Doctor: Contains methods like viewSchedule() and properties like specialisation that are related to the doctor's duties, with an emphasis on storing and displaying data unique to the doctor.

Pharmacist: Holds pharmacist-specific user information as well as inventory-related features like manageInventory().

Patient: This class encapsulates patient-related data without executing logic operations, as confirmed by attributes such as patientID, name, and medicalHistory.

Inventory: has attributes like the stock level and the threshold, this was a newly added feature to help detect low stock levels.

AppointmentOutcome: Has details and functions that help take care of the consultation process, medication details and also the type of service provided to the user.

**Control classes:**

As for the control classes, we have the classes below to help coordinate the interaction between the other entity classes and aid in controlling the overall logic of the use case. Like scheduling and patient data management, these classes are managed by these classes.

Login: handles the process of/ coordinates the process of user authentication/verification and redirects the users to their roles.

PatientDataManager: This class includes methods for patient records management. It mediates interactions between the entity and boundary classes and engages with the Patient class.

DoctorDataManager: similar to PatientDataManager but for doctors and has information about their availability for scheduling appointments and  so on.

StaffDataManager: coordinates all staff related information and operations (both general staff and the administrators).

**Boundary classes:**

To allow the users to interact with the HMS, the boundary classes are created which will allow the users to type in commands that can then be sent to the control classes which will then use the entity classes to perform the logic operations required by administering the whole process. Importantly, the boundary class HMSApp serves as the entry point to the HMS.

HMSApp: serves as main entry point to the HMS and allows user to get redirected to various sectors.

PharmacistMenu: Allows pharmacists to interact in the HMS with respect to the inventories and the prescriptions.

Doctor(as a menu): The initializeMenu() helps provide menu functions to the users

Patient(as a menu): allows the patients to interact in the HMSvia the CLI-based menu

**II.**    **Introducing the AppointmentOutcome class as a separate feature**

The whole process of appointment handling was carried out more generally in the initial method with the help of some basic functions such as confirming, rescheduling and cancelling of the appointments. However, we realised that this could be done in a more efficient and elegant manner using a separate class called AppointmentOutcome that can focus solely on this(note: usage of single responsibility principle-more explanation in later parts) to provide the outcomes regarding appointments. It includes functions like keeping track of the types of services, handing prescribed medications and more for each interaction. This new separate feature integrates with the two other classes: Doctor and Patient, thereby ensuring transparency and doctors to maintain accurate medical records of the patients.

**III.**    **Inventory management: addition of the low stock alerts update**

Apart from basic stock updates, the viewInventory method was designed to generate flags whenever the stock levels were observed when checking against a threshold, thereby improving the efficiency of the HMS.

**IV.**    **User authentication: Overcoming redundancy while logging in as a user**

Initially, each of the four user types were subclasses of the user class which was the parent class containing the login method. Later, this functionality was moved to the Login class to authenticate the users for uniform handling of the verification process, thereby reducing redundancy.

**VI.**    **Excel based data persistence using the help of Apache POI library**

The Apache POI library was used to help in developing the Excel file operations to manage the data constantly with ease as it was mentioned in the assignment instructions, a database wasn’t used. Without relying on any database, this way of working allowed us to provide the system with structured data storage. Hence, classes like PAtientDataManager, DoctorDataManager and StaffDataManager can all manage the read or write operations with ease using the help of the excel files, thereby aiding to the consistency across the system.

**VII.**    **Slot management done dynamically**

The method setAvailability helps in creating a system where the real-time availability of the doctors could be observed or noted. This will help the patients decide which slots to book to ensure no overlapping of the slots for appointments occur.the system update is designed to run dynamically using this dedicated method.

**VIII.**    **Error handling**

The system is designed in a way that its able to detect invalid inputs and other runtime errors that could potentially occur and performs exception handling for the same. The try-catch blocks are added in methods/operations like login, inventory updates and schedulingto handle these invalid user input, corrupted data that are passed on and even the handling of missing data files. This adds on to the reliability and robustness of the HMS.

**IX.**     **Inclusion of OOP principles and SOLID principles**

The initial model did make use of several OOP principles such as encapsulation/information hiding and inheritance, however, the use of SOLID principles were lacking. More details of this are illustrated later under the section: SOLID principles.

1. ***OOP PRINCIPLES***
2. **Encapsulation:** Encapsulation was a vital aspect in the creation of our project as it deals with sensitive and confidential data (eg: Diagnoses of patients, patient details). We assumed that the hospital already had baseline privacy standards put in place meaning the patients can view only their records and no one else’s and also maintains the doctor patient confidentiality. To do this we made sure that the classes Patient, Doctor and Pharmacist hid its attributes and only showed the necessary operations via public methods. Setters/getters were used if they were inherited by other classes so that they could have access to the information without it being visible to the users.

1. **Inheritance:** We used inheritance to avoid redundancy and were able to bring about a hierarchy where the user class was used as the base class and the other classes Patient, Doctor and Administrator were derived from. This allowed us to have a reusability of the shared methods and properties and also allowed us to alter or extend the features of the basic roles based on their specific needs.

1. **Polymorphism:** We used polymorphism in our project to help with code scalability and maintenance, it allowed us to improve the user experience. Method overloading and method overriding were used in our project to enable said polymorphism it gave us the chance to handle different data types and do various functions like for eg: the displayMenu function is made differently for the classes Doctor, Pharmacist and Administrator which caters to different properties each class requires.

1. **Abstraction:** Abstraction allowed us to hide the intricacy and difficulty of the data operations and only showed the required interfaces to the user roles. By using abstraction for the low level task like reading from and writing to the excel sheet we were able to supply a cleaner and a more straightforward UI. In classes like PatientDataManager and inventory the methods in them manage the complex file operations which allows the rest to interact with simple method calls. This aligns with our cause of not allowing other users to access confidential information not related to them.

***D. SOLID DESIGN PRINCIPLES***

**Single Responsibility Principle (SPR):**

The classes in the HMS were created keeping SPR in mind to serve as a good design metric so that if at all the system needs modification in future, it can be done so with ease. This principle aids in the classes not having responsibilities that are coupled but have one and only one responsibility that they accomplish well. The creation of the entity, control and boundary classes aid in serving this principle right. For instance, the role of the Patient class is only to store the information with respect to the user who is identified as a Patient. Furthermore, the Data manager classes, like PatientDataManager and DocotrDataManager also display the use of SRP as they ensure that the data storage logic is decoupled from other responsibilities. This ensures that the classes are all having high cohesion(perform their single duty very well) and low coupling as the links are minimised as much as possible.

**Open Closed Principle (OCP):**

The utilisation of OCP in the HMS is as follows: The Staff class acts as a superclass (generalisation) which has been extended by the other classes (via specialisation) such as Doctor, Patient, Administrator and Pharmacist. This supports the context/idea of keeping the design open for extensibility and closed for modification whenever the system is deemed to require a change to its current form. For instance, if there is a need to include a new staff other than Doctor, administrator or Pharmacist, for example a Nurse class, then through the OO concept of inheritance, the new class can be added to the system as a subclass of the superclass Staff. This shows the extensibility of the code and shows how the modification to the entire system can be prevented. Likewise, another instance in the HMS where OCP is used includes the generalisation of Patient class and Staff class to User class which is the superclass. Furthermore, the Data manager classes like PatientDataManager and DocotrDataManager help abstract the data operations, thus allowing the system to be open for extension including the need for inserting new storage formats without modifying the existing system.

**Liskov Substitution Principle (LSP):**

The subclasses created by extending their respective superclass *expect no more* from the user and *give no less*. The subclasses are all substitutable for their base classes. For instance, the subclass Doctor can completely replace the Staff class without providing any issues to the user.

The Doctor class has the provision to overwrite some functions from its superclass whereas the Pharmacist class has some add-on methods like updatePrescriptionStatus() that helps operate on the inventory-specific methods. Likewise for the Patient class and Staff class to their super class User class).

**Interface Segregation Principle (ISP):**

A class shouldn't be made to implement interfaces it doesn't use, according to the Interface Segregation Principle (ISP). This idea is used in our project making sure that every class only includes methods relevant to its particular function. The Doctor class, for instance, contains methods like viewAppointments() and updatePatientRecords(), but it does not have methods relating to inventory, which are only in the Pharmacist class. Similarly, the Patient class does not use any of the methods from the staff class; instead, it concentrates on patient-specific characteristics.Classes stay concentrated on their core goal because of this segregation. It increases maintainability and lowers the possibility of making mistakes when changing or expanding the system.

**Dependency Inversion Principle (DIP):**

High-level modules are guaranteed to rely on abstractions rather than low-level modules by the Dependency Inversion Principle (DIP). We implemented this in our HMS project through the use of abstract classes and interfaces. For instance, high-level modules like Login communicate with the abstract User class rather than particular implementations like Doctor, Patient or Pharmacist. This increases flexibility by separating the high-level reasoning from the low-level specifics. This abstraction encourages reuse and lessens module dependency. Since the high-level modules rely on the abstract user interface, they don't require any changes if a new user type is added in the future. In addition, the data manager classes introduced earlier help abstract the logic for data persistence, thereby proving that the details depend on the abstractions and abstractions don’t depend on the details. In the future, as opposed to the current way of using excel files as data sources, in the future if a need arises, other storage mechanisms can be can be added easily without the need for changing the lower level modules or higher level modules majorly.

***E. DETAILED UML CLASS DIAGRAM***

To better comprehend and visualise the code structure, we examined the links between the different classes in our system and included them in our UML Class Diagram. We created a fundamental architecture that successfully captures the linkages and dependencies between classes by using "is-a," "uses-a," and "has-a" relationships. By using an iterative process, we were able to improve our design and create a system that is well-organised and structured, guaranteeing future advancements' scalability and maintainability. The model mainly uses a “has-a” and “is-a” relationship between most of its classes.

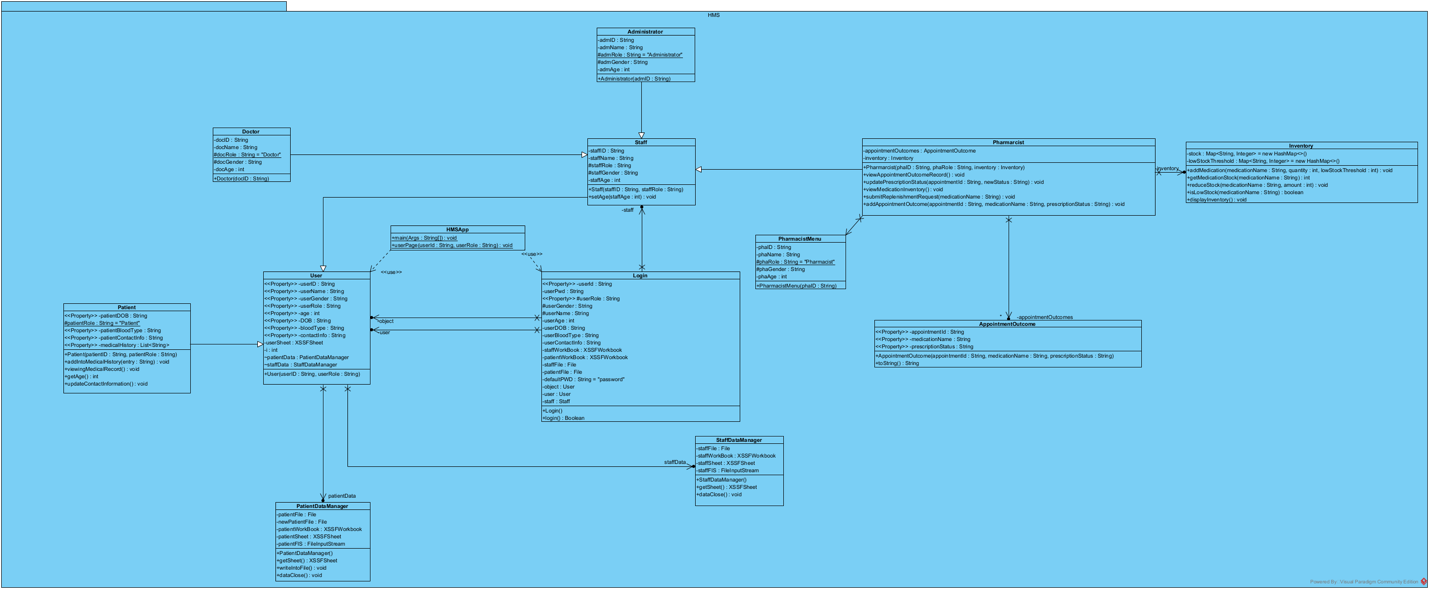


Figure 2: UML Class Diagram

***F. TRADE-OFFS AND DESIGN PATTERNS***

**Singleton pattern:**

We used the Singleton pattern mainly in the InventoryManager class. This pattern helps in ensuring that only a single instance of the class is there and also provides a global access to point to that particular instance of the class. In our HMS project this helped in avoiding duplication of the data in the inventory which is important in managing the system reliably. Hence, this instance of InventoryManager can be accessed throughout the system from different parts, which is helpful in updating the information constantly in tasks like replenishment requests in the system and handling of the stock levels. This helped to increase the efficiency of the system by avoiding the creation of duplicates in the HMS. However, a potential drawback of the pattern is that if it is accessed by multiple threads, concurrency issues might occur. That is, synchronisation keywords in Java will have to be used to avoid these types of race conditions via a set of mechanisms known as synchronisation mechanisms.

**Alternative factory pattern that can be considered:**

The Factory Pattern is another design pattern that we could've taken into account. This approach centralises the instantiation logic, making object creation easier. For instance, instances of Doctor, Pharmacist, or Administrator could be dynamically created using a factory class in response to user input. By abstracting the instantiation process, the Factory Pattern would further decouple the system and increase its maintainability and flexibility.

***G. TESTING***

1. Patient Actions

Test Case 1: View Medical Record: ***Test case passed***

Test Case 2: Update Personal Information: ***Test case passed***

Test Case 3: View Available Appointment Slots: ***Test case passed***

Test Case 4: Schedule an Appointment: ***Test case passed***

Test Case 5: Reschedule an Appointment: ***Test case passed***

Test Case 6: Cancel an Appointment: ***Test case passed***

Test Case 7: View Scheduled Appointments: ***Test case passed***

Test Case 8: View Past Appointment Outcome Records: ***Test case passed***

2. Doctor Actions

Test Case 9: View Patient Medical Records: ***Test case passed***

Test Case 10: Update Patient Medical Records: ***Test case passed***

Test Case 11: View Personal Schedule: ***Test case passed***

Test Case 12: Set Availability for Appointments: ***Test case passed***

Test Case 13: Accept or Decline Appointment Requests: ***Test case passed***

Test Case 14: View Upcoming Appointments: ***Test case passed***

Test Case 15: Record Appointment Outcome: ***Test case passed***

3. Pharmacist Actions

Test Case 16: View Appointment Outcome Record: ***Test case passed***

Test Case 17: Update Prescription Status: ***Test case passed***

Test Case 18: View Medication Inventory: ***Test case passed***

Test Case 19: Submit Replenishment Request: ***Test case passed***

4. Administrator Actions

Test Case 20: View and Manage Hospital Staff: ***Test case passed***

Test Case 21: View Appointments Details: ***Test case passed***

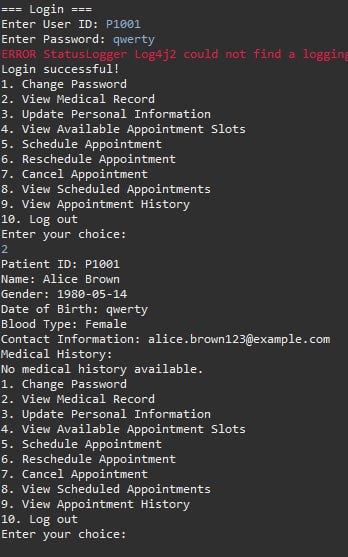
Test Case 22: View and Manage Medication Inventory: ***Test case passed***

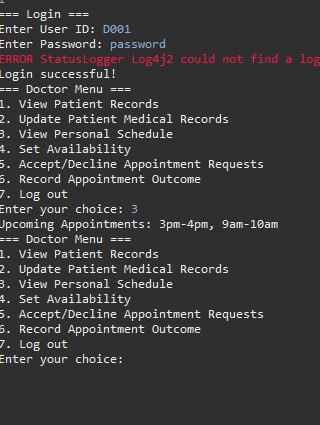
Test Case 23: Approve Replenishment Requests: ***Test case passed***

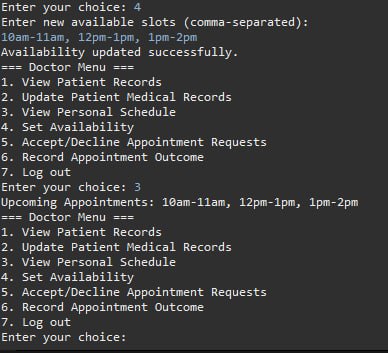
5. Login System and Password Management: ***Test case passed***

Test Case 25: First-Time Login and Password Change: ***Test case passed***

Test Case 26: Login with Incorrect Credentials: ***Test case passed***

******

******



***H. REFLECTION (challenges faced)***

Developing this project had been a long yet fulfilling journey, as students that are not proficient in java at all, this project turned out to be a challenge especially since we were navigating our way through vast uncharted territories, we tried our best with whatever knowledge we had to come up with our hospital management system.

Challenges faced:

1. While coding the project we realised the number of bugs we faced were endless. We were so concerned about how we would make it work as amateur coders in java, it was a humbling experience and very time consuming to deal with as there were always constant changes. Our persistence and resilience was what led us to solving the code we were pushed to our limits and learned to make improvements.
2. We even had trouble with finalising our UML diagram. It took us hours and weeks to perfect a UML diagram that worked for our cause. But after numerous considerations and brainstorming sessions we were finally able to create a UML with everything we needed.