
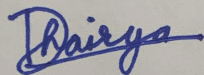





### **Declaration of Original Work for CE/CZ2002 Assignment**

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature/Date
Dahiya Adit	SC2002	SCSC	 19-November-2024
Kakkar Dhairya	SC2002	SCSC	 19-November-2024
Dahiya Advik	SC2002	SCSC	 19-November-2024
Gautham Krishna Manoj	SC2002	SCSC	 19-November-2024
Gunda Sai Venkata Aaditya	SC2002	SCSC	 19-November-2024

Important notes:

1. Name must EXACTLY MATCH the one printed on your Matriculation Card.

2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

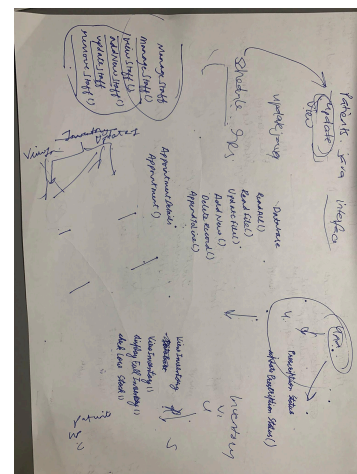
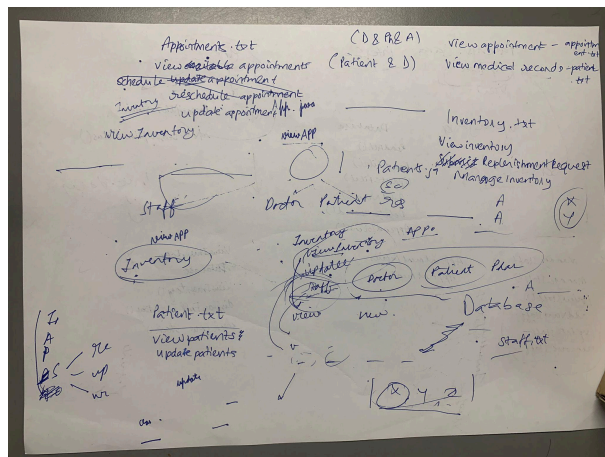
## GITHUB LINK-

<https://github.com/Advikdahiya/HMSFinal-2002-Group2.git>

## Approach Taken

The process of creating and implementing the Hospital Management System started with a thorough brainstorming session. The team worked together to pinpoint essential needs, discuss possible obstacles, and establish a preliminary framework. In these meetings, we developed basic drawings and design concepts that laid the groundwork for our project. These preliminary designs helped us see the general structure of the system and pinpoint important classes and modules.

Initially, our process began with exploring the issue before diving into creating the Hospital Management System through collaborative brainstorming sessions. At first, our focus was on comprehending the problem statement, identifying essential features, and making rough drafts of the system's structure. These preliminary sketches allowed us to see how different parts were connected and laid the groundwork for our design. We assigned responsibilities to team members to ensure we were making progress simultaneously and had frequent discussions to merge our ideas.



As we dug further into the project, we researched the basics of effective object-oriented design, specifically focusing on the SOLID principles. After revisiting, we discovered that our original design did not follow all of these principles, resulting in inefficiencies and repetition in the code. Originally, we were writing our code solely according to the user menu and not based on other dependent classes. This led us to go back and enhance our method. We collaborated once more,

this time concentrating on improving our design with the concepts of encapsulation, abstraction, inheritance, and polymorphism. By methodically utilising these fundamental Java principles, we improved our system's modularity, flexibility, and overall cohesiveness by defining specific scopes and dividing functionalities into modular components. Initially, we prioritised following the SOLID design principles to guarantee scalability and maintainability. Several versions of the UML diagram were created, deliberated upon, and improved to match our goals.

Moreover, some of us even took SC2006 Software Engineering this semester, so we even incorporated principles like Agile and Singleton.

Agile basically means that the program and its source code are flexible and efficient, able to adapt and respond quickly to change. Essentially, Agile involves constructing the system in small, controllable segments to facilitate updates and adjustments, emphasising users' needs. This is applied in various contexts in our project for example, classes such as `ScheduleAppointment`, `CancelAppointments`, and `RescheduleAppointment` are responsible for managing distinct tasks. An appointment can be scheduled by `ScheduleAppointment`, which checks availability and books it, whereas `CancelAppointments` specifically deals with cancellations only. This modularity permits incorporating additional appointment-related functions without modifying the current code.

Specialised menus such as `PatientMenu` and `PharmacistMenu` are created for specific user roles. Patients can book appointments, while pharmacists can review inventory. This guarantees that the system efficiently caters to various user requirements.

Interfaces such as `UpdateAppointments` and `ViewAppointments` enable adding new appointment-related features, such as mass appointment updates, by incorporating these interfaces into new classes without altering the current ones.

Singleton ensures code clarity by having each component perform a single task, limiting complexity (similar to the Single Responsibility Principle). Classes such as `RemoveStaff` concentrate exclusively on staff removal, while `ViewPastAppointments` simply display past appointments. This simplicity enables a clear understanding of the functionality of each class.

Reusable Tools are present, like the `Database` class, which is utilised for typical file tasks such as reading, updating, and modifying records. Instead of repeating these tasks across different classes, this single class manages them all, streamlining the code for increased efficiency. Moreover, our code is easily understandable from a client's perspective. For example, the `ViewMedicalRecords` class clearly presents patient records. They are simple, containing only a few lines of code that perform precisely as their name implies, making them readable and adjustable for anyone.

## Principles Used

### 1. Single Responsibility Principle (SRP):

Each class in our system adheres to the **Single Responsibility Principle (SRP)** by focusing on a single, well-defined responsibility. For instance, `ManageStaff` handles all operations related to staff management, such as `AddNewStaff` and `RemoveStaff`, without mixing unrelated functionality. Similarly, `UpdateInventory` focuses solely on updating inventory records, while `ScheduleAppointment` is exclusively responsible for scheduling appointments. Separating responsibilities makes the system easier to maintain and modify, ensuring that changes to one aspect of the system do not affect unrelated components.

`ManageStaff`: This class handles all operations related to staff management, such as adding, updating, and removing staff. It works closely with other staff-related classes, such as `AddNewStaff`, `RemoveStaff`, and `UpdateStaff`, to further break down responsibilities.

`ScheduleAppointment`, `RescheduleAppointment`, and `CancelAppointments`: These classes are responsible for appointment management. Each focuses on a different aspect of appointment handling: scheduling, rescheduling, and cancellation, respectively. By breaking this functionality across multiple classes, changes to one part (e.g., how appointments are rescheduled) do not affect the others.

### 2. Open-Closed Principle (OCP):

The system is designed to allow extensions without modifying existing code. The **Open-Closed Principle (OCP)** ensures that a class is open for extension but closed for modification. This means a system's core functionality should not require modification when new features are added.

The `ViewAppointments` interface provides a standardised way to handle various appointment-viewing operations. It is implemented by specific classes such as `ViewAvailableAppointments`, `ViewPastAppointments`, and `ViewScheduledAppointments`, each focusing on a unique aspect of appointment data. This design allows for easy extension; for instance, if new viewing functionality is required (e.g., viewing virtual appointments), a new class can simply implement `ViewAppointments` without altering the existing implementations.

Similarly, our system employs eight interfaces, each tailored for specific viewing operations.

Extension in Inventory Management: Classes like AdminStockApprove extend CheckLowStock and UpdateInventory, showing how additional inventory-related operations can be added without altering the core inventory functionality.

### **3. Liskov Substitution Principle (LSP):**

The **Liskov Substitution Principle (LSP)** states that objects of a subclass should be able to replace objects of the parent class without altering the correctness of the program. In your system, the nurse-related classes such as, NurseUpdatePatient, and NurseViewPatient extend their corresponding doctor-related classes, adhering to the LSP by ensuring consistent behaviour and substitutability.

In all these cases, nurse-specific classes (NurseViewPatient, NurseUpdatePatient, etc.) can be substituted in place of their corresponding doctor-specific parent classes (CiewPatient, UpdatePatient, etc.) without any runtime errors or changes in expected behaviour.

Polymorphic Behaviour:

The system takes advantage of polymorphism. High-level modules can rely on the parent class (ViewPatient, UpdatePatient) while dynamically invoking nurse-specific implementations (NurseViewPatient, NurseUpdatePatient), ensuring correct and consistent behaviour.

### **4. Interface Segregation Principle (ISP):**

The **Interface Segregation Principle (ISP)** ensures that interfaces are small, role-specific, and tailored to the needs of the implementing classes. This prevents classes from being forced to implement methods they do not use, promoting modularity and reducing unnecessary dependencies.

Our project effectively demonstrates ISP by using eight interfaces designed for specific roles and functionalities. Each interface focuses on either viewing or updating operations for different entities such as Patients, Appointments, Inventory, and Staff:

Each interface serves a specific purpose, ensuring that implementing classes are not forced to handle responsibilities outside their scope.

NurseMenu and DoctorMenu focus on operations relevant to their roles, like NurseViewPatient or NurseViewDoctor, avoiding unrelated methods such as those for managing appointments or inventory.

Inventory Management is divided into ViewInventory and UpdateInventory, ensuring that viewing tasks (e.g., checking stock levels) and update tasks (e.g., approving stock changes) are handled independently by specialized classes.

## **5. Dependency Inversion Principle (DIP):**

Database Class: The Database class provides a single point of access for all operations across different entities such as Staff, Patient, Inventory, and Appointments. High-level modules like ManageStaff, ScheduleAppointment, and UpdateInventory interact with the Database class to perform data operations without knowing the specifics of how the data is stored or managed.

For example:

- ManageStaff interacts with the Database to add or remove staff.
- ScheduleAppointment relies on a Database to create new appointment records.
- UpdateInventory uses a Database to modify inventory levels.

This design ensures that changes in the underlying data storage logic (e.g., switching from an SQL-based system to a NoSQL system) would only require changes in the Database class, leaving the high-level modules unaffected.

## **Loose Coupling:**

The system demonstrates loose coupling by using the Database class as a centralised abstraction for managing entities like Patient, Staff, and Inventory. High-level components such as UpdatePatient and AdminStockApprove interact with Database through its methods, without relying on the internal implementation details of data storage or retrieval. This ensures that any changes to the Database logic, such as switching to a new database system, will not impact these high-level components, promoting flexibility and ease of maintenance.

## **Abstraction:**

We utilised abstraction to highlight essential features and simplify the interaction with various components by concealing implementation details. This method enables developers to utilize advanced features without comprehending the complex classes, leading to a more manageable, expandable, and user-friendly system. For example, UpdateAppointments and ViewAppointments interfaces simplified the essential features for managing appointments. By specifying functions like updateAppointments or viewAppointments within these interfaces, we guaranteed that different implementations (such as ScheduleAppointment, CancelAppointments, and ViewPastAppointments) adhered to a uniform agreement. This enabled us to incorporate additional features, like rearranging appointments, without changing the current code base. Another instance is the Database class, which covers file operations like reading, writing, and updating records. Developers dealing with patient or appointment data can utilise these methods without being concerned about the specific details of the file handling implementation.

**Encapsulation:**

Throughout our whole code, we followed the principle of encapsulation by keeping the attributes of every class private. This guarantees that the internal state of an object remains concealed from outside entry and allows us to control how the data is accessed and modified by only allowing changes to the attributes through methods like get and set. This ensures the safety and reliability of our data and improves the maintainability of our system. For example, any modifications to the internal structure of a class would not affect the external components. This leads to a clean and structured design.

**Inheritance:**

The extra role of the nurse can access several tasks similar to some of the options provided to the Doctor role. Hence, we leveraged the Object-Oriented feature of inheritance to share functionalities between the two roles. Through this, we are able to put common functions in the base class(e.g., Classes for Doctor) and include role-specific methods in the derived classes(e.g., Classes for Nurse). This allowed us to create a reusable and scalable system.

**Polymorphism:**

We used this key OOP feature to provide dynamic role and user choice-specific interactions. We used polymorphism to display the appropriate submenu based on the user's role during login, such as Doctor, Patient, Nurse, Admin, or Pharmacist. This design gave us a simple code for the main menu, allowing for flexibility and ease of adding new submenus specific to different roles. A single interface Menu with function displayMenu() caters to all the roles, eliminating the need for complex conditional logic.

**Extra features:****1. Hashing password upon login for enhanced Security:**

We have also implemented password hashing to increase the security and protect user passwords during the login process. Instead of storing the user's password in plain text, we first pass it through a custom hash function to generate a unique hash for each password and then store it in the allusers.txt file. When the user logs in, the system hashes the inputted password and compares it with the stored hash. By implementing this feature, we ensure that the user password details remain secure even if the allusers.txt database is compromised.

**2. Addition of Nurse as a New Role:**

Another significant extension of the system is the addition of a nurse role to the already existing roles. This provides a new level of interaction within the HMS and highlights the ease of adding new roles and features to our code. The nurse role has three classes:

- NurseViewPatient() gives nurses the option to view patients according to their patient ID or they can just view all patients and get an overview of their medical information.

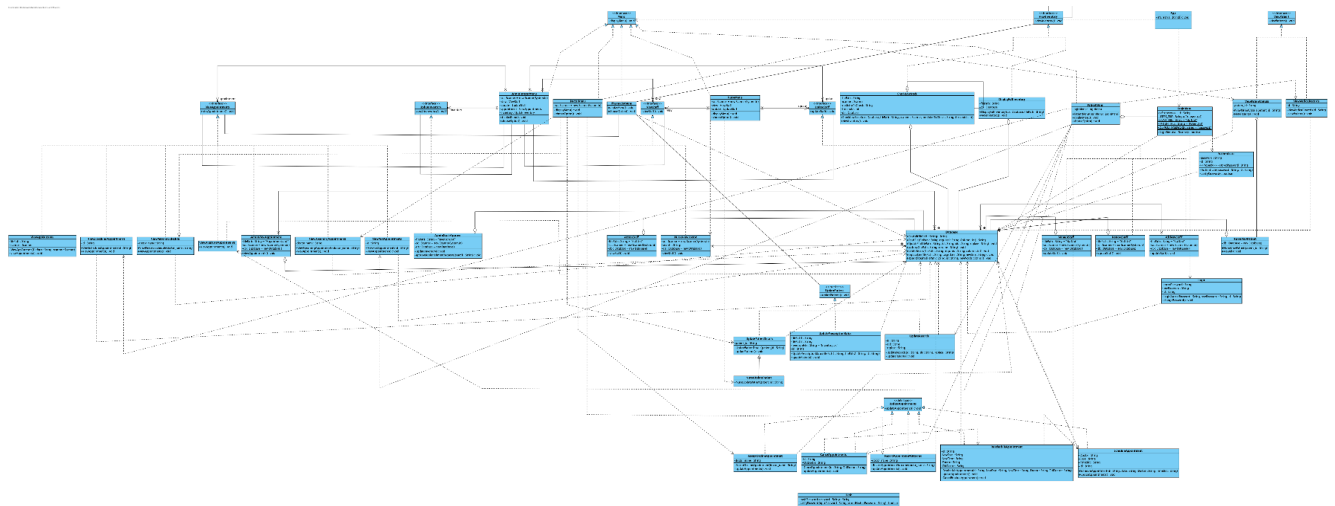
- NurseUpdatePatient() allows nurses to modify patient details, such as status, medications provided, and treatment plan.
- NurseViewDoctor(): Nurses can also view details of doctors from the Staff.txt file, which contains a record of all staff members.

Since our design approach follows SOLID principles with a focus on modularity, we can integrate future additions without any changes to the existing code. The ability to easily add new roles and functionalities also exemplifies the use of interface segregation and dependency injection, making our code testable and easily maintainable.

## Detailed UML Class Diagram:

We understood the relationship between the different Classes in our system and then made arrows connecting them in our UML Class diagram to visualise our code easily. We used “is-a,” “uses-a,” and “has-a” relationships between classes to create an initial structure, followed by our current class diagram. This enabled us to have a well-organized architecture for our system,, allowing for efficient scalability and maintainability for the future.

Even though our UML diagram looks very complicated, it is estimated that every class will have around two arrows for loose coupling. Therefore, for our 47 classes, there should be around 95-100 arrows, which is around the number of arrows we have. Thus, our code is reusable, extensible, and maintainable.



(Uploaded Separately for better clarity)



## **Testing:**

### **1) Patient Menu**

- Test 1 [View Medical Record] - Passed
- Test 2 [Update Personal Information] - Passed
- Test 3 [View Available Appointment Slots] - Passed
- Test 4 [Schedule an Appointment] - Passed
- Test 5 [Reschedule an Appointment] - Passed
- Test 6 [Cancel an Appointment] - Passed
- Test 7 [View Scheduled Appointments] - Passed
- Test 8 [View Past Appointment Outcome Records] - Passed

### **2) Doctor Menu**

- Test 1 [View Patient Medical Records] - Passed with Assumption
- Test 2 [Update Patient Medical Records] - Passed
- Test 3 [View Personal Schedule] - Passed
- Test 4 [Accept/Decline Appointment Requests] - Passed
- Test 5 [View Upcoming Appointments] - Passed
- Test 6 [Record Appointment Outcome] - Passed

### **3) Pharmacist Menu**

- Test 1 [View Appointment Outcome Record] - Passed
- Test 2 [Update Prescription Status] - Passed
- Test 3 [View Medication Inventory] - Passed
- Test 4 [Submit Replenishment Requests] - Passed

### **4) Administrator Menu**

- Test 1 [View and Manage Hospital Staff] - Passed
- Test 2 [View Appointments Details] - Passed
- Test 3 [View and Manage Medication Inventory] - Passed
- Test 4 [Approve Replenishment Requests] - Passed

### **5) Nurse Menu**

- Test 1 [View Patient Details] - Passed
- Test 2 [View Doctor Details] - Passed
- Test 3 [Update Patient Status] - Passed

## 5) Login System

- Test 1 [First-Time Login and Password Change] - Passed
- Test 2 [Login with Incorrect Credentials] - Passed

## Assumptions Made:

- 1) The hospital ID is P\_ for patients, A\_ for administrators, M\_ for pharmacists, N\_ for Nurses, and D\_ for doctors- This ensures consistency throughout the code.  
Example of Patient ID- P123
- 2) We assume that the users interacting with our system fully know its purpose, how it works, and its various features and functions. Considering this assumption, our output minimises explanations.
- 3) We use Available, Upcoming, Requested, and Completed to check appointment status.
- 4) We store details in four TXT files (Patient, Appointments, Staff, Inventory), each with a fixed format. A “|” symbol separates each column of the TXT file.
- 5) All users behave professionally and don't misuse or try to break the system in any way.
- 6) Login credentials for staff and patients are securely hashed through our hash class.
- 7) In Test Case- 1 of the doctor menu, we assume that any doctor can view any patient's medical records.

## Challenges:

- 1) Naming Issues: When naming some interfaces, half of us wrote a different name, and others wrote something else. It took a lot of time to combine the names and ensure everyone was on the same page. For example, two members wrote ViewPatient() as the interface name, while others wrote ViewPatients().
- 2) Duplicate Code: In the beginning, many of us wrote duplicate code. Improving the design became a significant task, and we incorporated all the sound design principles, like SOLID.
- 3) Structure: Coding was the easy part. Our main challenge with the project was achieving the output while ensuring good design. Coming up with the basic UML structure took the most time.
- 4) Compilation: Sometimes, when we changed our code, the .class files didn't auto-recompile, so even though our code was correct, the output was wrong. This led to us wasting time.

## **Room for improvements:**

- 1) UI improvements: With more future users, we could provide many more explanations throughout the runtime, making our system more user-friendly and navigable.
- 2) Comprehensive Testing: We could Expand test cases to cover edge scenarios and integrate automated testing frameworks to ensure reliability.

## **Personal Growth and Learning:**

- 1) Helped us prepare for the final exam
- 2) Ensured constant communication throughout the project completion
- 3) The project prompted greater understanding and use of advanced OOP concepts and design patterns.
- 4) Inspired continuous development and improvement

## **Conclusion:**

In conclusion, our Hospital Management System effectively incorporates key functionalities such as data management, efficient process handling, and seamless integration across different modules. The system also incorporates Object-oriented design principles to improve operational efficiency and ensure a user-friendly interface.