

# **Group Project - Hospital Management System**

SC2002 - Object Oriented Design & Programming

Lab Group: SCS7 | Group Number: 2

Name	Matriculation Number
Hui Cheok Shun, Jordan	U2321272L
Jerwin Lee Chu Hao	U2321487B
Nguyen Pham Minh Quan	U2320069H
See Tow Tze Jiet	U2322598B
Dallas Ng Zhi Hao	U2323084C

## **Declaration of Original Work**

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 beawarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature/Date
Nguyen Pham Minh Quan	SC2002	SCS7/Group 2	(2024) 17/11/2024
Jerwin Lee Chu Hao	SC2002	SCS7/Group 2	Jein 16·11-24
Hui Cheok Shun, Jordan	SC2002	SCS7/Group 2	Zola 16/11 forg
See Tow Tze Jiet	SC2002	SCS7/Group 2	John 16/11/2024
Dallas Ng Zhi Hao	SC2002	SCS7/Group 2	T2 17/11

# 1. Design Considerations

The Hospital Management System (HMS) is a Java-based Command Line Interface (CLI) designed to streamline hospital operations such as patient management, appointment scheduling, staff coordination. The system seeks to optimize resource management and simplify administrative tasks. It offers role-specific functionalities while ensuring data privacy. The system is built with flexibility in mind, allowing addition of new roles or features in the future.

# 1.1 Design Approach

Our HMS utitises the Entity-Control-Boundary (ECB) architecture design pattern. The reason behind the use of ECB pattern is due to the use-case driven nature of the HMS, with various different use cases for each user role: patient, doctor, pharmacist and adminstrator. Our packages can be divided in 4 main functions:

Our Model classes act as entity classes define data as attributes of each object in the programme such as appointments, patients, doctors, etc. The corresponding CSV files act as a lasting local database implementation of our HMS, allowing the programme to be terminated without losing the data and any changes made during run time.

Our Users and System classes are our controllers, handling program logic and store temporary data for quick retrieval and processing on those data. It automatically load data when necessary and save data once updated to save on computing power and memory space.

Our Menu classes are our boundary classes that enables encapsulation of data and methods, only allowing access to certain methods without revealing the details and logic behind them, thus making the user's experience more seamless.

Finally, the Enum classes represent fixed values that are used throughout the programme..

# 1.2 Assumptions

Our HMS assumes that:

- 1. Changes in data will be saved once made and when exiting programme
- 2. Data is saved in CSV files in the data folder.
- 3. Only one user will be using the system at any one time

4. User will need to change password after the first login (ie. when their password is still the default 'password').

## 1.3 Limitations

The current implementation of the system involves several trade-offs that affect its limitations:

- 1. **Space vs. Time Complexity**: The system uses HashMaps instead of lists to improve data fetching efficiency, reducing time complexity. However, this increases space complexity, consuming more memory compared to simpler data structures like arrays or lists.
- 2. **Scalability vs. Simplicity**: The use of CSV files for data storage makes the system simple and lightweight, but limits scalability. Unlike more robust database solutions like SQL or MongoDB, CSV files struggle with handling large datasets and complex queries, affecting the system's ability to scale efficiently.
- 3. **Security vs. Convenience**: The system stores data in CSV files without encryption to keep the project simpler and focus on core features. While this avoids complexity, it compromises security, particularly for sensitive information, as encryption is not implemented for passwords and other critical data.

# 1.4 Applied Design Principles

#### 1.4.1 Single Responsibility Principle (SRP)

SRP states a class should have only one reason to change, and it should focus on a single functionality or responsibility, allowing the code to be easier to maintain, modify, test, and reuse.

We applied this principle throughout our project by organising the structure and defining clear responsibilities for each class, where each directory and file has a single responsibility. For example, the menu file only manages user interactions related to the menu, and does not handle data management. We also implemented this principle with classes, taking AppointmentOutcomeSystem for example. This class is responsible solely for managing appointment outcomes, such as adding and dispensing medications. It does not deal with the display of outcomes or managing appointments directly, ensuring single responsibility.

### 1.4.2 Open/Closed Principle (OCP)

OCP states that classes should be open for extension but closed for modification, ensuring new functionality can be added without altering existing code, reducing issues that may arise. This

was done by making use of models without functionality, allowing us to add new functions without changing existing code.

The User class was designed as a base class with common attributes ,to act as a model where specific users such as Doctor and Patient can extend from user, without modifying the base User class.

#### 1.4.3 Liskov Substitution Principle (LSP)

LSP states that subclasses must be substitutable for their base classes without altering the correctness of the program, ensuring derived classes enhance functionality of the parent, instead of causing issues by asking for more information. This was achieved by using inheritance in user-related classes such as Doctor, Patient, Administrator, and Pharmacist, inheriting from the User class. Each subclass inherits all attributes and implements the methods defined in the User class, such as *functionCall()*, ensuring it behaves as expected. This shows that the subclass is able to properly substitute the User class properly. For example, when a Doctor object is to be passed into a method requiring a User type, the system will still function as intended as the behaviours of the Doctor object is valid and expected of a User object.

### 1.4.4 Interface Segregation Principle (ISP)

ISP states that a class should not be forced to implement interfaces it does not use. This keeps interfaces specific and avoids unnecessary dependencies that can cause complications. We implemented a MenuInterface, which has subclasses DoctorMenu, PatientMenu, AdministratorMenu and PharmacistMenu. This allows each of these classes to call their own instance of *displayOptions()* to show the different menu options. This is a simple design and we do not overcomplicate the process.

### 1.4.5 Dependency Inversion Principle

This principle states high-level modules must not depend on low-level modules, and they should instead depend on abstraction. This decouples components, making the system more flexible and easier to extend. For example, the HMS class does not depend on every single menu classes and user classes directly. Rather, to use these classes, HMS is dependent on the abstract parent class Menu and the base model class User instead, making the HMS flexible in handling any type of users without having an extensive code base with unnecessary functions.

## 1.5 OOP Design Principles

#### 1.5.1 Abstraction

The system leverages abstraction to streamline interactions with complex components by emphasizing key attributes essential to its functionality. Each class, such as the UserManagementSystem, offers methods that reveal only the required features while concealing the underlying implementation. For example, the class includes methods like *updatePassword() setMedication()* to allow users to change their passwords that met the system's requirements, abstracting the intricate details of the list implementation such as the use of the private *passwordValidation()* method. This design allows users to work with high-level methods without needing to understand the internal data structures or implementation specifics.

#### 1.5.2 Encapsulation

All Model classes such as User, MedicalRecord and Appointment implement encapsulation by keeping all attributes private and exposing them only through public getter and setter methods. This design prevents unauthorized changes and ensures data is properly validated before updates. For instance, the Appointment class uses the *setAppointmentStatus()* method to control status modifications, ensuring state changes are regulated. Sensitive information, such as patient data, is securely stored and accessed solely through dedicated methods.

### 1.5.3 Polymorphism

The system leverages polymorphism by utilizing a base Menu class to define shared behaviors for subclasses like DoctorMenu and PharmacistMenu. These subclasses override specific methods, such as the *displayOptions()* function, to implement role-specific functionality dynamically at runtime, demonstrating dynamic polymorphism.

#### 1.5.4 Inheritance

The User class functions as a base class for specialized roles like Administrator, Doctor, Pharmacist, and Patient, providing a consistent structure for shared attributes such as userId, password, name while allowing each subclass to define unique role-specific behaviors such as Doctor's *updateMedicalRecord()* or *setAvailability()* methods

# 1.6 Additional Features Improvements

### 1.6.1 Password Complexity

To improve the security of users' account, we enforce password requirements for more complex passwords. These requirements include:

- 1. Length should be between 6 and 20.
- 2. Must include at least one lowercase letter.
- 3. Must include at least one uppercase letter.
- 4. Must include at least one digit.
- 5. Must not be the default 'password'.

### 1.6.3 Future Improvements

Finally, there are additional features such as billing system and Notification system, that we would like to implement but due to the time constraint of the project, we have yet done so. Specifically, we would like to implement:

- A billing system that print out a nicely formatted invoice that contains the patient name, appointment date, appointment outcome record and the required payment as a sum of service fee (depending on service type) and medical fees (depending on medication dispensed).
- 2. A notification system that saves notifications such as automated low stock alert or new scheduled appointment and notify the relevant users as soon as they log in and allow them to check whenever they want.

Beside these changes, we would like to improve the database implementation and security of the system by employing encryption for our sensitive data such as password and conduct frequent backup to prevent substantial data loss caused by programme errors or cyberattacks. Given the importance of hospital data, it is one of our top priority to improve our database implementation.

### 1.7 Reflection

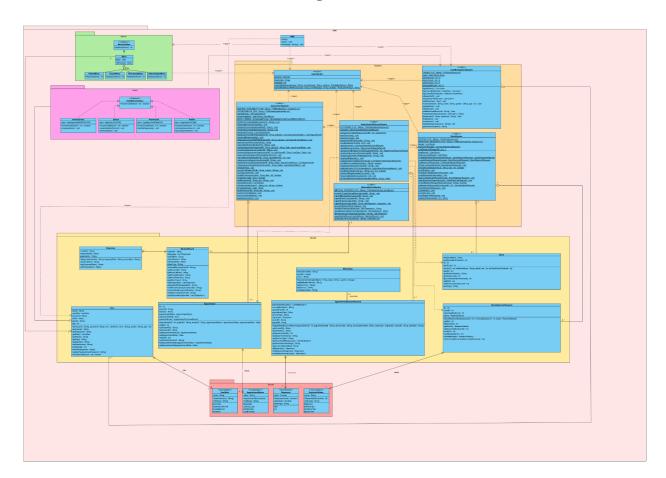
From this course, we have learnt the importance of applying Object-Oriented Design in Systems. This ensures that the system is navigable, easily upgradable and keeps users' data safe. Initially, we came up with a very simply designed diagram that requires complicated communications

between each class, making the system convoluted and too tightly coupled, making it hard to update or extend any class.

We found that by adopting the SOLID Design principles, we were able to make our project much more modular and functional as evident by our class diagram. As a result, we had less trouble making improvements and extending the code while keep the main code working. Moreover, referring to the design patterns such as singletons, ECB, etc. help us in better formulate our design.

Furthermore, we have gained useful experience on developing projects as a team. Work delegation for such an integrated project had proven to be difficult and we struggled to complete the task at hand without messing up others' part. While we have yet to found the best strategy, we managed to complete it by divide and conquer while all maintaining the SOLID principles. This helps all the part fit together easily and minimal changes were required to have the system work as intended.

# 2. Detailed UML Class Diagram



# 3. Testing

# 3.1 Welcome Page:

3.1.1 Login & Logout

```
Welcome to the Hospital Management System!

Please choose an option:
1. Log in
2. Register as a new patient
3. Exit
1

User ID: D1

Password: 123

Welcome, doctor Dr.John.

Doctor Menu:
1. View Patient Medical Records
2. Update Patient Medical Records
3. View Personal Schedule
4. Manage Availability for Appointments
5. Accept or Decline Appointment
5. Accept or Decline Appointment
6. View Upcoming Appointment
7. Record Appointment
8. Logout
Choose an option:
```

Upon Login, the user is brought to their respective menu pages. In this case, doctor is brought to the doctor menu.



Log out brings the UI back to the welcome page. Option 3 will exit the System, ending the program.

3.1.2 Register as new Patient



Create new Patient, Patient Menu, Change password upon next login.

#### 3.2 Patient:

#### 3.2.1 View Medical Record

```
Medical record not found for User ID: PT8 | Prompting user to create a new medical record... | Blood Type | 0+ | Creating a new medical record... | Diagnoses: | No diagnoses recorded.
```

If patient does not have an existing medical record, they will be prompted to create one. Once Created, the medical record can be viewed at any time.

3.2.2 Update Personal Information

```
--- Updating Medical Record for User ID: PT8 ---
Leave the input blank to retain the current value.
Current Date of Birth: 2000-02-02
Enter new Date of Birth (YYYY-MM-DD):
2000-01-01
Current Phone Number: 91234567
Enter new Phone Number:
91234234
Current Email Address: jeff@ntu.com
Enter new Email Address: jeff@ccds.com
Current Blood Type: 0+
Enter new Blood Type (e.g., A+, B-, 0+):
0-
Medical record saved or updated successfully.
Medical record updated successfully for User ID: PT8
```

Patients may update their medical record.

- Date of Birth
- Phone number
- Email address
- Blood Type

If the input is invalid, the user will be prompted to re-enter the correct input.

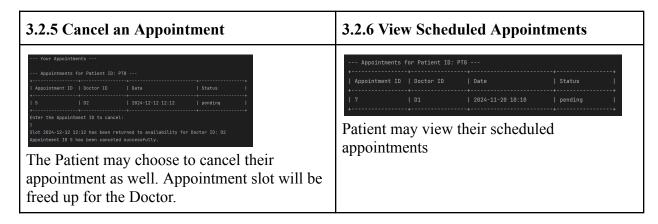
3.2.3 View Available Appointment Slots & Scheduling an Appointment

Patients may view available appointments to book.

To book an appointment, Patient must choose the Doctor they wish to book an appointment with and the timeslot they want.

3.2.4 Reschedule an Appointment

The Patient is able to reschedule their appointment by selecting the appointment they wish to reschedule, and then selecting a different available slot. The previous appointment slot will then be freed up for the Doctor.



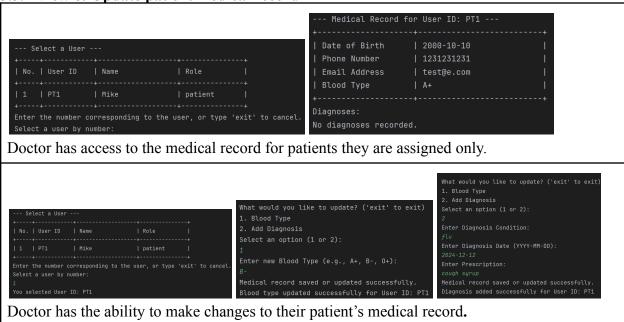
**3.2.7 View Past Appointment Outcome Records** 



Patients can view their past appointment records, which include AppointmentID, Date, Service Type, Medication (if any), Consultation notes, and medicine disbursement status.

### 3.3 Doctor

3.3.1 View & Update patient medical record



#### 3.3.2 Setting an available timeslot

```
--- Doctor Menu ---

1. View Availability

2. Add Availability

3. Remove Availability

4. Exit

Select an option:

2

Enter an available slot (YYYY-MM-DD HH:mm):

2024-12-12 12:00

Availability added for Doctor ID: D1
```

Doctor is able to create an empty appointment

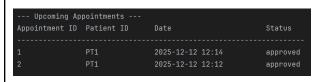
- will be filled upon accepting an appointment request
- Empty appointment will be reflected in their personal schedule

3.3.3 Receive and accept appointment request



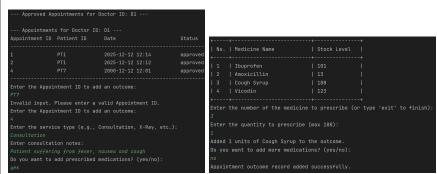
Doctor can receive an appointment request issued by patient. If choose to accept request, their upcoming appointments and personal schedule will be updates

3.3.4 View Upcoming Appointment



Doctors can view their upcoming appointments

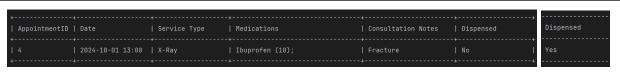
3.3.5 Record Appointment outcome



Doctors can issue an appointment outcome report once the appointment has been completed.

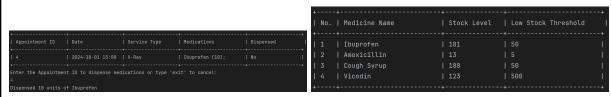
# 3.4 Pharmacist

3.4.1 View & Update Appointment Oucome Record



Pharmacists view Appointment Outcome Record to see the status of Medication dispensement. Pharmacists can then update the Medication Dispensement status once medication.

3.4.2 View Medical Inventory



Pharmacists have access to the medical inventory, where they can check the stock of various medications.

Administrators can also view the medical inventory.

3.4.3 Submit Replenishment Request

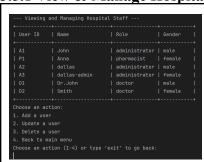


Pharmacists can submit a Replenishment Request for medication that are below the low stock threshold.

Once submitted and approved by the administrator, the medication stock level will be updated.

### 3.5 Administrator

3.5.1 View & Manage Hospital Staff



Administrators can view hospital staff information



Administrators can manage staff information, by adding, removing or updating staff.



Administrators can update staff information, namely staff name, gender, and age.

3.5.2 Approve Replenishment Requests



Stock Replenishment Requests will be passed to the Administrator, who can either approve or reject the request.

4. GitHubRepository: SC2002-HMS