



**NANYANG
TECHNOLOGICAL
UNIVERSITY**
SINGAPORE

SC/CE/CZ2002: Object-Oriented Design & Programming

Hospital Management System (HMS)

AY 2024/2025 SEMESTER 1 | SCSi Group 6

Github Main Page: <https://github.com/kaibin157/SC2002>






Name	Matriculation Number
Chong Kai Bin	U2321606K
Lee Mei Ting	U2322099H
Heng Zeng Xi	U2323232C
Ng Yi Xiang	U2321562D
Sachdev Garv	U2323262H

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 honoured 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
Chong Kai Bin	CZ2002	SCSI	 12/11/24
Heng Zeng Xi	CZ2002	SCSI	 12/11/24
Lee Mei Ting	CZ2002	SCSI	 12/11/24
Ng Yi Xiang	CZ2002	SCSI	 12/11/24
Sachdev Garv	CZ2002	SCSI	 12/11/24

1. Design Considerations

Hospital Management System (HMS) is a Java application, designed with a modular approach using Object Oriented Programming concepts while focusing on its reusability and extensibility. The HMS system is designed to increase efficiency and streamline healthcare operations to improve patient care and provide a scalable platform for future enhancements and updates.

1.1 Design Approach

HMS was designed in mind to include OOP design principles, such as **SOLID** and **Model-View-Controller (MVC)**. We have used SOLID principles such as **Open-Closed Principle** and **Liskov Substitution Principle** to ensure our code is modular to allow for fast and easy changes. With SOLID in place, we have also implemented MVC which is tied to the **Single Responsibility Principle** to separate each packages' functionalities, consisting of packages such as Models, Views and Controllers. These will be further elaborated on in Section 1.4.

1.2 Assumptions Made

For the Hospital Management System (HMS), we assume that the initial data imported for

staff, patients, and inventory will be the most updated, thus eliminating the need for additional checks during setup. This is also backed by the implementation of real-time updates to the system folders whenever necessary. Additionally, we assumed that users such as Patients, Doctors, Pharmacists, and Administrators are familiar with HMS functionalities and their limitations based on their specific roles.

We assume that all sensitive data, such as patient medical records, are secure and encrypted to follow personal data protection laws and are only accessible to authorised users. This is also done by declaring variables as private in our code, and that any details updated by any individual, are updated in the necessary Excel sheets containing its details. Assumptions are also made that overlapping actions are impossible by taking a Single Program Instance Approach, such that overlapping actions such as identical scheduling appointments by two patients do not occur as only one user is accessing the program at a time. As we are taking a Single Program Instance Approach, we do not need to account for system load and performance either due to overloading of user volumes and other similar issues.

As this system is related to healthcare, we assume that all changes and updates are authorised and that all actions are logged for accountability and error-logging purposes.

1.3 Use of Object-Oriented Concepts

1.3.1 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. In our Project, we have achieved through the User class and the `displayMenu(HMS hms)` method:

- Example: The `displayMenu (HMS)` method in the User class is declared as an abstract method, which must be implemented by any subclass of User. This allows each subclass (such as Doctor, Pharmacist, or Administrator) to provide its own implementation of `displayMenu`, even though they all share the same User superclass.
- Usage: When we call the `displayMenu` on a User object that is actually a Doctor or Pharmacist, Java will use the correct subclass's implementation based on the actual type of the object.

```

public void displayMenu(HMS hms) {
    Scanner scanner = new Scanner(System.in);
    while (true) {
        System.out.println("Doctor Menu:");
        System.out.println("1. View Patient Medical Records");
        System.out.println("2. Update Patient Medical Records");
        System.out.println("3. View Personal Schedule");
        System.out.println("4. Set Availability for Appointments");
        System.out.println("5. Update Availability for Appointments");
        System.out.println("6. Accept or Decline Appointments");
        System.out.println("7. View Upcoming Appointments");
        System.out.println("8. Record Appointment Outcome");
        System.out.println("9. Change Password");
        System.out.println("10. Logout");
    }
}

```

Figure 1.3.1a: Abstract class displayMenu implemented in Doctor.java

1.3.2 Inheritance

Inheritance allows a class to inherit attributes and methods from another class, promoting code reusability. In our project, we have defined a base Class User, which is a superclass, and any specific user type can inherit from it.

- Example: The User class contains common properties like hospitalID, password, and name, along with common methods like getHospitalID, changePassword, and displayMenu. Any subclass of User (e.g., Doctor, Administrator) would inherit these attributes and behaviours.
- Usage: By extending User, the subclasses automatically have access to its properties and methods, eliminating the need to duplicate code.

```

public class Administrator extends User {
    /** The gender of the administrator. */
    private String gender;

    /** The age of the administrator. */
    private String age;

    * Constructs an Administrator with the specified details.[]
    public Administrator(String hospitalID, String password, String name, String gender, String age) {
        super(hospitalID, password, name);
        this.gender = gender;
        this.age = age;
    }
}

```

Figure 1.3.2a: Administrator Class extends to User class and inherits properties of User

1.3.3 Encapsulation

Encapsulation restricts direct access to an object's fields and methods, bundling the data with methods that operate on it. In our project, encapsulation is achieved through private attributes and public getter/setter methods.

- Example: In the Medication class, fields like name (of medication) and lowStockAlert are private. They can only be accessed or modified through public methods (getName, getLowStockAlert), providing controlled access to the data.
- Usage: This keeps the internal state of the Medication object safe from direct modification and only allows changes through defined methods.

```

public class Medication {

    /** The name of the medication. */
    private String name;

    /** The current stock level of the medication. */
    private int stockLevel;

    /** The threshold at which a low stock alert is triggered. */
    private int lowStockAlert;

    /** The status of the medication (e.g., pending, dispensed). */
    private String status;
}

```

Figure 1.3.3a: Variables declared as private in Medication class

1.3.4 Abstraction

Abstraction talks about how the code should only show necessary details to the user to add security to the code. Interfaces or abstract classes can be used to implement abstraction. In our project, abstraction is achieved through declaring an abstract class instead of interface.

- Example: In the User class, we have declared it as an abstract class. It defines the common attributes and methods for all users under User class, such as hospitalID and password.
- Usage: By hiding the non-essential information from the user, it reduces the complexity of the code as it progressively becomes larger. This also allows other classes, such as Doctor, to extend to the abstract class and use its methods instead of writing the same code again, avoiding any duplication of codes.

```

public abstract class User {

    protected String hospitalID;
    protected String password;
    protected String name;
}

```

Figure 1.3.4a: Abstract class User declared in User.java

1.4 Design Principles

1.4.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle is defined for classes to only have 1 responsibility each. The Hospital Management System (HMS) is organised into 4 packages. Each package contains multiple Java files that have different functionalities but all of the files are aligned with a singular responsibility of each package. This ensures each class focuses on a distinct task that is aligned with the singular responsibility stated for each package, providing clear

organisation in the code and improving the modularity and scalability in the program. The name of the packages and their singular responsibilities are elaborated below.

(a) Models

Models are used to define the layout of the program, consisting of entity classes that describe their attributes and their relationships with one another. Some entity classes include Patient, Doctor and Pharmacist. Each class was designed with its key attributes and behaviours in mind. The relationships highlight the interactions and dependencies between different classes to allow seamless management of specific tasks such as patients' appointments and medical prescriptions.

(b) View

Views are classes that are in charge of displaying the interface of the program to users. We have two classes, HMS and Main. HMS displays different user interfaces depending on the role of the users (Patients, Doctors etc.) and allow them to perform specific tasks. Main handles and displays the login page of the program for the user to sign into their respective accounts.

(c) Controllers

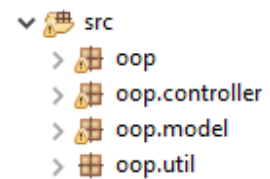
Controllers are classes that act as a linkage between Models and Views. They provide a centralised location to handle specific operations based on what the user has inputted to manage the flow of data and business logic processing. We have 4 controller classes that provide functionality aligned to its domain:

- AppointmentController: Handles operations that are related to managing appointments, such as scheduling and cancelling appointments. It interacts with various Excel files such as Appointment_List.xlsx to persist appointment data.
- AuthenticationController: Handles user authentication when signing into the program. Authentication.xlsx is used to authenticate users using their hospital ID and password. Data in the file and in the memory are updated when new accounts are created or the passwords have been changed.
- InventoryController: Handles the medication inventory, allowing users with specific roles to use its functionalities such as adding or removing medications. It interacts with various Excel files like Medicine_List.xlsx to persist and retrieve medication data.

- **UserController:** Handles user-related operations such as viewing patient records and doctor schedules. It interacts with various Excel files such as `Staff_List.xlsx` to persist and retrieve patient and hospital staff data.

(d) Utilities

These classes are presets that are used in the program to perform common functions that are often reused. For example, the `Constant` class consists of a set of `String` variables that indicate the absolute file paths of the various Excel files used in the program. This discourages the practice of continuously declaring the file path when reading or writing to the Excel files as it can be long, allowing us to simply call the variable's name to perform actions on the specified Excel file.



1.4.2 Open-Closed Principle (OCP)

The Open-Closed Principle is defined as classes being able to include new features without changing any existing code to prevent the coder from going through and making changes to the entire code to adapt to its new features. This means we only need to extend existing classes or implement new interfaces to adjust accordingly without affecting the functionality of the program. An example to achieve this principle is by using abstract classes. One abstract class that has been implemented is `displayMenu(HMS hms)`. This abstract class is implemented by various user roles such as `Doctor` to display the different functionalities each role has. This allows new user roles to utilise and implement this class by including their own functionality without needing to change the existing code.

1.4.3 Liskov Substitution Principle (LSP)

The Liskov Substitution Principle talks about how the objects of a superclass should be correctly substituted with objects of its respective subclasses without causing compilation or runtime errors. This means that subclasses can contain less restrictive rules, but no stricter rules are allowed to be implemented. For example: `Doctor` class is a subclass of `User` class and inherits the methods from `User` class, such as getting the hospital ID and name of `Doctor` objects. This ensures that any instance of `Doctor` class can be substituted whenever a `User` class is expected without causing errors.

1.4.4 Interface Segregation Principle (ISP)

Interface Segregation Principle is about how a class should not be forced to implement methods from other classes when it is not needed. This means interfaces should be segregated to have specific methods for a specific task rather than have one main interface that contains all methods for various tasks. An example will be the AuthManager interface which focuses on authentication-related tasks and contains only authentication-related methods like authenticate(hospitalID, password) and updatePassword(hospitalID, newPassword). These require implementations in HMS class, specifically only used for authenticating users. This avoids redundant methods and keeps the system maintainable.

1.4.5 Dependency Inversion Principle (DIP)

The Dependency Inversion Principle focuses on how higher-level modules should not rely on their lower-level modules, which are also their implementers. Instead, they should rely on abstractions and interfaces whenever possible to reduce dependencies on specific implementations. An example will be our User class consists of methods of various user types for different roles. When we want to retrieve the hospital ID of a Doctor user, rather than depending on Doctor class, we depend on the high level module of the Doctor class, which is the User class. This ensures that high level modules can be reused easily and will not be affected by any changes made by its implementers.

1.5 Further Enhancements

Currently, many hospitals & polyclinics in Singapore have implemented payment of medical bills through the use of apps. This allows patients to pay their bills on their phones before the deadline without spending time queuing at the polyclinics. This makes the process faster and more convenient for them. Based on this concept, we decided to include an additional feature in the interface for the patients to make payment after they have completed their appointments. Patients can select the option to make payment and input their card details. They will then be prompted to choose if they would like their card details to be saved for future transactions. If the patient chooses yes, the card details will be encrypted and saved in Patient.xlsx. Patients would no longer need to re-input their card details during payment, hence completing the transaction in a swift manner.

In future developments, we can add an update logging feature to track user actions, such as changes made to a user's details. This log would capture details like the user, action type, and

timestamp, providing an audit trail for authorised parties for purposes such as audits or investigations.

1.6 Reflections

Throughout this project we have encountered several challenges along the way. One of the tasks needed to use Apache POI to read and write into Excel files. We did not learn how to use the POI at first. But, as a team, we managed to research and understand how to use Apache POI functions by adding the POI library in the folder.

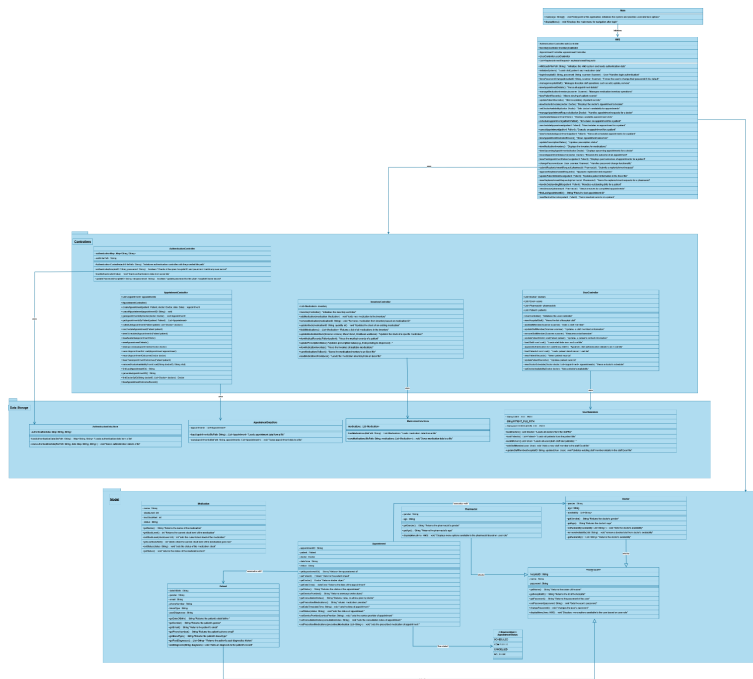
As the project grew, the complexity increased as the number of variables, methods, and functions increased. From this, we realised the importance of a UML class diagram where it helped us to visualise the structure of the project in a clearer view and provided a template for us to keep track of each class which enhances its traceability when debugging. This helped us to save time in events where adjustment was needed as we do not need to look through all the folders and classes individually.

The last issue that arose was that all the functions of the project were initially contained within the HMS class, leading to a cluttered code and making it difficult for us to navigate through. We understood that this was not a good coding practice. Therefore, to resolve this, we used MVC design pattern and Single Responsibility Principle to split the code into multiple models, views, and controllers where each of them handled a specific task. Having this modular approach not only improved the maintainability and readability of the code but also highlighted the importance of good software design and practice.

Overall, we learnt that building an OOP application is not as straightforward as what we have learned during the lecture and tutorials. It contains many small details like pieces of puzzles and it requires consistency ranging from the design to implementation. This is definitely one of the good habits we have adapted and we were able to apply it for other programming modules.

2. UML Class Diagram

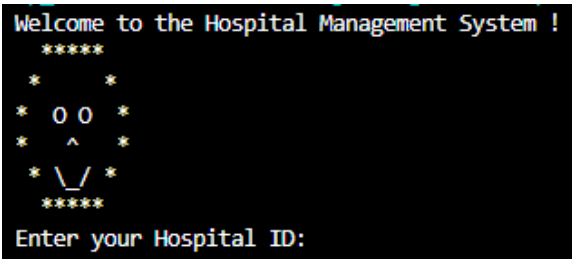
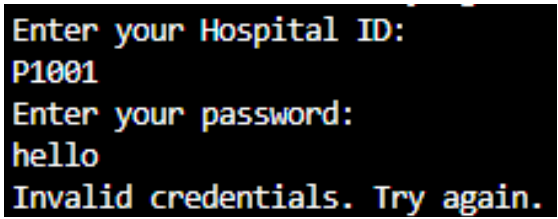
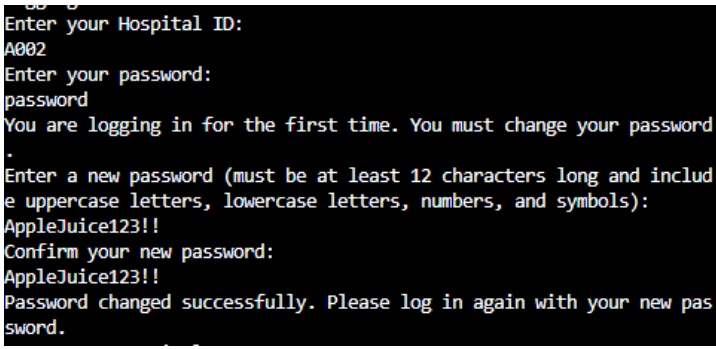
The pictures below show the full UML diagram, which is then sectioned into the view, controller with data storage, and model diagrams. A clearer overview of the diagram can be found in another file that has been submitted.



3. Functional Tests and Results

In this report, we will showcase the important cases instead of displaying all of them. Some of which are as follows:

3.1 Login System and Password Management

Login page and asks user to input password again if it's incorrect	
	
Ask user to change password as a first-time user	
	

Different menu displayed for different users	
Patient Menu: <ol style="list-style-type: none"> 1. View Medical Record 2. Update Personal Information 3. View Available Appointment Slots 4. Schedule an Appointment 5. Reschedule an Appointment 6. Cancel an Appointment 7. View Scheduled Appointments 8. View Past Appointment Outcome Records 9. Pay Outstanding Bills 10. Change Password 11. Logout 	Doctor Menu: <ol style="list-style-type: none"> 1. View Patient Medical Records 2. Update Patient Medical Records 3. View Personal Schedule 4. Set Availability for Appointments 5. Update Availability for Appointments 6. Accept or Decline Appointments 7. View Upcoming Appointments 8. Record Appointment Outcome 9. Change Password 10. Logout
Login successful! Welcome, Mark Lee Pharmacist Menu: <ol style="list-style-type: none"> 1. View Appointment Outcome Record 2. Update Prescription Status 3. View Medication Inventory 4. Submit Replenishment Request 5. View Replenishment Requests 6. Send Invoice 7. Change Password 8. Logout 	Administrator Menu: <ol style="list-style-type: none"> 1. View and Manage Hospital Staff 2. View Appointments Details 3. Manage Medication Inventory 4. Approve Replenishment Requests 5. Change Password 6. Logout

3.2 Patient & Doctor Actions

Doctor creates appointment slots, patient books a slot with doctor	
Enter available time slot for appointments (format: yyyy-MM-dd HH:mm): 2025-01-20 12:30 Availability set for: 2025-01-20 12:30 Do you want to add another time slot? (yes/no):	Available slots for all doctors: 1 Dr Bob - 2025-01-20 12:30 2 Dr Bob - 2024-11-26 13:30 3 Dr Bob - 2024-11-30 15:20 4 Dr Bob - 2025-02-10 14:15 Choose a slot by entering the number: 1 Appointment scheduled successfully with Dr. Dr Bob at 2025-01-20 12:30
Doctor accepting and declining appointments, reflects on patient's side	
Appointment ID: APT7 Patient: Charlie White Date/Time: 2024-12-20 10:30 Do you want to accept this appointment? (yes/no) yes Appointment confirmed. Appointment ID: APT8 Patient: Charlie White Date/Time: 2025-01-20 12:30 Do you want to accept this appointment? (yes/no) no Appointment declined.	Your scheduled appointments: Appointment ID: APT7 Doctor: Dr. Dr Bob Date/Time: 2024-12-20 10:30 Status: confirmed Appointment ID: APT8 Doctor: Dr. Dr Bob Date/Time: 2025-01-20 12:30 Status: cancelled
Doctor updates appointment outcome, reflects on patient's side	

Confirmed Appointments for Dr. Bob: Appointment ID: APT7 Date/Time: 2024-12-20 10:30 Do you want to record the outcome for this appointment? (yes/no) yes Enter the type of service provided (e.g., consultation, X-ray): Consultation Enter any prescribed medications (comma-separated if multiple): Panadol Enter consultation notes: Rest more Appointment outcome recorded successfully.	Past Appointment Outcomes for Charlie White: Doctor: Dr Bob Date: 2024-12-20 10:30 Service Provided: Consultation Prescribed Medications: Panadol Consultation Notes: Rest more Status: completed
---	---

3.3 Pharmacist and Administrator Actions

Pharmacist requesting replenishment for Panadol, Administrator approving it	
Medication Inventory: 1. Medication: Panadol Stock Level: 10 Low Stock Alert: 20 Enter the number of the medication to request replenishment: 1 Enter the amount to request for replenishment: 100 Replenishment request submitted successfully and is pending approval	Pending Replenishment Requests: 1. Pharmacist ID: P001 Medication: Panadol Requested Amount: 100 Status: pending Enter the number of the request to approve: 1 Replenishment request approved successfully.
Pharmacist sees status on replenishment request changes to approved, stock level updated	
Medication: Panadol Requested Amount: 100 Status: approved	Medication Inventory: Medication: Paracetamol Stock Level: 2019 Low Stock Alert: 20 Medication: Ibuprofen Stock Level: 50 Low Stock Alert: 10 Medication: Amoxicillin Stock Level: 75 Low Stock Alert: 15 Medication: Panadol Stock Level: 110 Low Stock Alert: 20 Medication: Morphine Stock Level: 20 Low Stock Alert: 30

3.4 Additional Feature: Payment after appointment outcome

Pharmacist sends appointment invoice to patient, patient pays													
Completed Appointments: 1. Appointment ID: APT7 Patient: Charlie White Date/Time: 2024-12-20 10:30 Enter the amount: \$100 Invoice sent successfully.	Outstanding Bills for Charlie White: 1. Appointment ID: APT7 Doctor: Dr Bob Date/Time: 2024-12-20 10:30 Total: \$100.0 Enter the number of the appointment you want to pay for: 1 Proceed to payment for Appointment ID: APT7 Enter your card number (16 digits): 2543679260293618 Enter card expiry date (MM/YY): 1/24 Invalid expiry date. Please enter in MM/YY format. Enter card expiry date (MM/YY): 12/30 Enter card CVC (3 digits): 693 Payment successful. Thank you! Do you want to save your card details for future payments? (yes/no): yes Card details saved successfully.												
Card details saved in Patient_List.xlsx in encrypted format													
<table><thead><tr><th>Patient ID</th><th>Name</th><th>Card Details</th></tr></thead><tbody><tr><td>P1001</td><td>Alice Brown</td><td>lXHI0x+PHHqiojVq3R3Fow==:F8KlJo7y7JkPSnEdGIXH0UfZ0/COANxEI4zqyknQ9UA=</td></tr><tr><td>P1002</td><td>Bob Stone</td><td></td></tr><tr><td>P1003</td><td>Charlie White</td><td>JT3yNOU/2exhqjGRJ/dgvA==:Ko3zp+5UmKDM7yo+HnEgx3SWYevCdmlQBnBtuUqLmKQ=</td></tr></tbody></table>		Patient ID	Name	Card Details	P1001	Alice Brown	lXHI0x+PHHqiojVq3R3Fow==:F8KlJo7y7JkPSnEdGIXH0UfZ0/COANxEI4zqyknQ9UA=	P1002	Bob Stone		P1003	Charlie White	JT3yNOU/2exhqjGRJ/dgvA==:Ko3zp+5UmKDM7yo+HnEgx3SWYevCdmlQBnBtuUqLmKQ=
Patient ID	Name	Card Details											
P1001	Alice Brown	lXHI0x+PHHqiojVq3R3Fow==:F8KlJo7y7JkPSnEdGIXH0UfZ0/COANxEI4zqyknQ9UA=											
P1002	Bob Stone												
P1003	Charlie White	JT3yNOU/2exhqjGRJ/dgvA==:Ko3zp+5UmKDM7yo+HnEgx3SWYevCdmlQBnBtuUqLmKQ=											