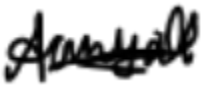
Declaration of Original Work for SC2002/CE2002/CZ2002 Assignment

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 (SC2002/CE2002 CZ2002)	Lab Group	Signature /Date
Aanya Monnappa	SC2002	SCMA, 3	19/11/2024 
Jervis Tham Kai Kit	SC2002	SCMA, 3	19/11/2024 
Loo Yi Xuan Maeko	SC2002	SCMA, 3	19/11/2024 
Seet Jia, Viona	SC2002	SCMA, 3	19/11/2024 
Yeo Ke Jun	SC2002	SCMA, 3	19/11/2024 

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.

UML Class Diagram

(Attached as a separate file)

Additional Features/Functionalities Implemented in System

Password Hashing

To improve HMS security, we implemented password hashing with PBKDF2. This method uses a unique, random salt for each password, ensuring that even identical passwords generate different hashes. By securely storing hashed passwords instead of plaintext, PBKDF2 protects sensitive hospital data from unauthorised access and brute-force attacks.

Before hashing:	After PA001 changes their password from default, the new password is hashed as: hashed password,salt
ROLE,ID,PW,SALT,NAME PATIENT,PA001,password,NULL,ALice Brown PATIENT,PA002,password,NULL,Bob Stone	ROLE,ID,PW,SALT,NAME PATIENT,PA001,bVNXsCMsiQrfeIE/QFzKXwZsgbyRmVctR96YZ2A0MiA=,QC93lCkhRvZAnvUzpe9zUQ==,Alice Brown PATIENT,PA002,password,NULL,Bob Stone

DOCTOR'S AVAILABILITY Database

A doctor's default availability is set from 8am to 5pm for the entire week, indicated by 'Y' (Yes) under each time slot in the DOCTOR'S AVAILABILITY section. If a doctor is unavailable during a particular time, 'Y' will be replaced by 'N' (No) in the CSV file. This availability is visible to patients, enabling them to schedule appointments based on the doctor's available time slots.

forgetUserID option

If users forget their user ID, our HMS offers an option to retrieve it by entering their full name. The system checks the entered name against existing records and returns the corresponding user ID if a match is found.

clearConsole()

Clearing the console after each session protects patient confidentiality by preventing sensitive data exposure. It also enhances user experience by providing a clean, seamless interface for professional and focused interactions with the HMS.

Startup class

Improves user experience. A dynamic welcome message attracts the user's attention and gives a prelude to the interactive features of our HMS, effectively communicating system readiness.

Password Requirements

To enhance security in HMS, password validation ensures that user passwords meet the specific requirements:

1. At least 8 characters long
2. Includes at least one uppercase letter
3. Includes at least one lowercase letter
4. Includes at least one digit
5. Includes at least one special character (e.g., @, #, !)

This is implemented using regex (regular expressions) as a tool for defining patterns that the passwords must match.

```
public static boolean isValidPassword(String var0) {
    String var1 = "^(?=.*[a-zA-Z])(?=.*\\d)(?=.*[@#$%^&+=!]).{8,}$";
    return var0.matches(var1);
}
```

Implementation of regex in HMS

Design Considerations

Use of Object-Oriented Programming Concepts

Encapsulation

Within each user class (e.g. Patient), the variables (e.g. ID, name) are defined as private to promote encapsulation. This prevents the variables from being accessed outside the class, protecting the internal state of the object from unintended interference from outside the class.

Abstraction

The Patient class serves as a high-level module, abstracting the complexity of lower-level modules like Appointment, AppointmentManager, and DoctorSelection. It interacts with these modules to schedule appointments and assign doctors, hiding implementation details behind a simple interface. This design allows users and other components to engage with the Patient class without needing to manage the underlying logic.

Composition

Our system uses composition to model real-world relationships between objects like Patient, Doctor, and Appointment. For example, a Patient can have multiple Appointment objects, representing a one-to-many relationship. The Patient object holds a collection of Appointment objects, each detailing a specific visit, enabling the system to track multiple visits over time.

Considerations

Error Handling and Validation

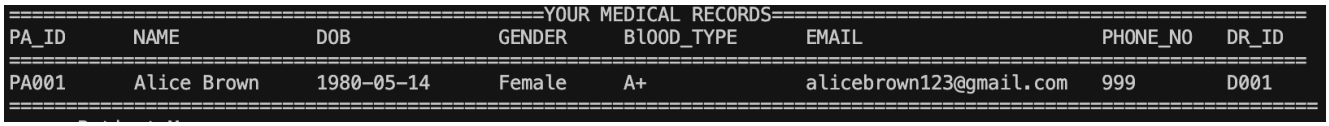
Error handling and data validation are crucial in a Hospital Management System (HMS) to ensure data integrity and system reliability. The timings are formatted as HHMM, and if an improper format is keyed in, HMS will print out an error message asking for another input using a while loop. In addition, HMS uses e.printStackTrace() for exception handling, as it prints the full stack trace of the exception to the console, allowing us to identify the location and reason for the error.

Data Management

Since the HMS stores various databases containing sensitive information, our code should allow the system to efficiently store and manage data. We have decided to store the data using CSV files. An advantage is that CSV files are relatively simple and readable. Furthermore, since the datasets for this project are relatively small, CSV is lightweight with very little system resource overhead.

User Experience

Although this application is limited to a command-line interface (CLI), our group prioritised the user experience by designing clear, role-specific menus. For example, when users choose to view their medical records, the CLI will display the following:



CLI display of medical records

In addition, to improve user experience and ensure flexibility, the system is designed to handle input case insensitivity. This allows the user to provide inputs in lowercase without needing to worry about case-related errors or inconsistencies, thus enhancing the robustness and usability of the application.

Interaction Between Classes

In HMS, interactions between the classes are crucial to ensure that various components of the system work together seamlessly to fulfil its functions. By structuring the HMS around interconnected classes, the system becomes more cohesive, reliable, and capable of handling the diverse, real-time needs of healthcare management.

Appointment.java

Doctor, Patient, Administrator and Pharmacist can all access the Appointment class to different extents. Patients can schedule, reschedule, cancel appointments and view their upcoming appointments. As for doctors, they are able to view their upcoming appointments scheduled by the patients, and can mark it as completed after the appointment is completed. Administrators can view scheduled appointments as well, but they are unable to edit the appointment details.

MedicalRecManager

Medical Record contains a patient's complete medical history, including personal and medical information. Patient class can view their medical records but can only update personal information. Meanwhile, Doctor class can update the patients' medical records by adding new diagnoses, prescriptions and treatment plans.

OutcomeRecord

The AppointmentOutcomeRecord is tied to a single appointment, storing details like appointment date, service type, prescribed medications (name and status), and consultation notes. Patients can view but not edit their past records. Doctors update the outcome after each appointment, including prescribed medications, which pharmacists access to fulfill prescriptions and update their statuses.

Design Patterns

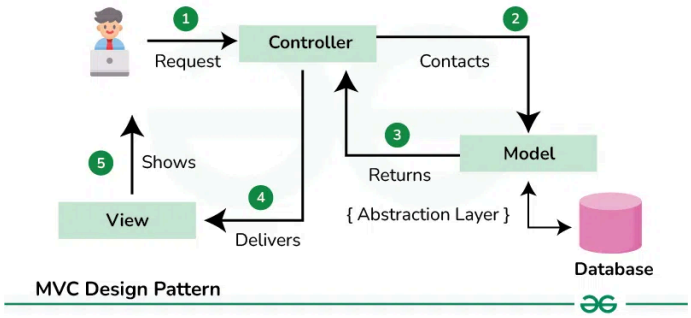


Figure 1: Model-View-Controller (MVC) Design Pattern Diagram (GeeksforGeeks, n.d.)

Model-View-Controller (MVC) Design Pattern

The MVC (Model-View-Controller) design pattern breaks an application into three parts: the Model (which handles data and rules), the View (which is what users see), and the Controller (which connects the two and handles the implementation) (GeeksforGeeks, n.d.). This separation allows for easier updates and maintenance. In our project, each role has its own MVC structure, organised in separate folders. The controller handles user input, and updates the Model accordingly with the updated data. It houses the main “bulk” of the code, as it includes method implementation, as well as requesting, updating of databases through the model. It interacts with other class controllers

Model

The model manages the data for different roles (Doctor, Patient, etc.) and handles information requests from the Controller or View. In our code, it is represented by classes like Doctor.java, Patient.java, and Pharmacist.java.

Controller

The controller handles user input, and updates the Model accordingly with the updated data. It houses the main “bulk” of the code, as it includes method implementation, as well as requesting, updating of databases through the model. It interacts with other class controllers to carry out its actions, and works with the View classes to take in user input. In our code, it is represented as DoctorController.java, AppointmentManager.java, PatientController.java etc.

View

The view classes in our project are slightly different, in a sense that it contains less responsibility. It mainly handles the output of the different user menus, after logging in. It interacts with the controllers to pass on the user input as a parameter, for the controller to then carry out the selected action. In our code, it is represented as DoctorMenu.java, PatientMenu.java etc., and it extends from the abstract UserMenu class.

Advantages of MVC Design Pattern

One advantage of the MVC design pattern is its scalability. Methods can be easily extended in the controller classes without needing to modify the Model or View.

Another benefit is cleaner code, as the three components (Model, View, Controller) are separated. Changes to the View, for instance, only require edits to the View classes, reducing errors and eliminating the need to rewrite data retrieval code.

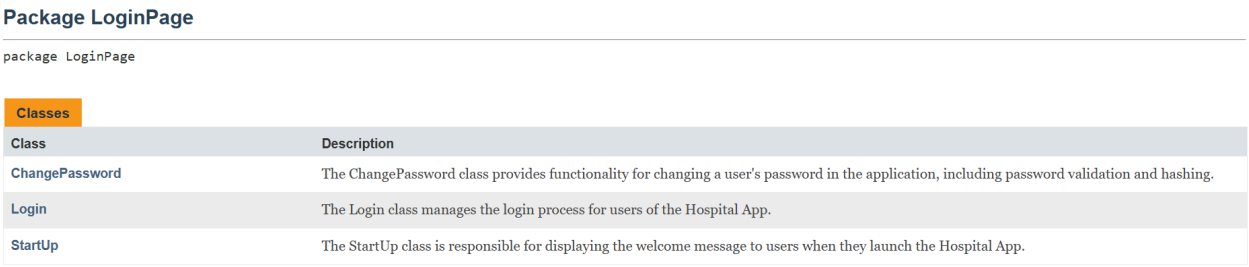
Finally, the MVC pattern simplifies testing and debugging. Each component can be tested independently, such as testing methods in the Controller class without relying on the Model or View.

SOLID Principles

Single-Responsibility Principle:

The Single-Responsibility Principle dictates that effectively, every class should have only one responsibility.

In our project, we adhered to this by ensuring every class has a distinct, focused role. For example, in our LoginPage package, it contains 3 classes (ChangePassword.java, Login.java, StartUp.java). These 3 classes provide functionality to change a user's password, manage login processes, and displaying the welcome message to users respectively, each with its own unique functionality.



Package LoginPage

package LoginPage

Classes

Class	Description
ChangePassword	The ChangePassword class provides functionality for changing a user's password in the application, including password validation and hashing.
Login	The Login class manages the login process for users of the Hospital App.
StartUp	The StartUp class is responsible for displaying the welcome message to users when they launch the Hospital App.

Figure 2: Application of Single-Responsibiltiy Principle in LoginPage

More examples of how we have applied the Single-Responsibility Principle across various classes in our project are available in the **JavaDocs** documentation. Each class is documented with detailed descriptions of its specific role and responsibility, showcasing how SRP is implemented to maintain a clean and modular code structure.

Open-Closed Principle:

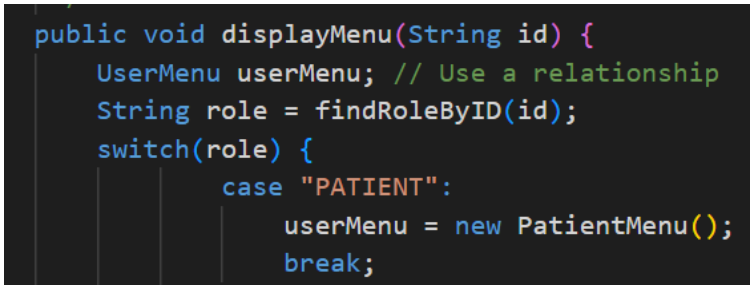
The Open-Closed Principle states that classes should be open for extension and closed to modification.

The Open-Closed Principle can be effectively applied when new types of users are added to the system. For instance, when introducing a new user type, such as a Pharmacist or an Administrator, we extend the abstract UserMenu class without modifying the class, ensuring system stability.

Liskov Substitution Principle:

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

Using polymorphism in our code, for example, methods such as menuActions() in the base abstract UserMenu.java class will be overwritten in the various menu classes to receive input for different actions. This means that, we are able to pass an object of the PatientMenu.java subclass to any method that expects an object of UserMenu.java class, without any errors.



Example of Liskov Substitution Principle in our code

Interface Segregation Principle:

The Interface Segregation Principle ensures clients only depend on the methods they use. In our project, the User.java interface provides a contract for user roles (e.g., patients, doctors) to implement. Each role defines its own methods, preventing unnecessary dependencies and making it easier to extend functionality without affecting other parts of the code.

Dependency Inversion Principle:

The Dependency Inversion Principle states that classes should depend on abstractions, not concrete classes. In our project, the User.java interface defines common attributes for roles like Doctor.java and Pharmacist.java, with specific role classes implementing their own behaviours. This ensures flexibility and adherence to abstraction. As for our menu classes, the UserMenu.java class serves as an abstract base class for user menus in the hospital management application. It provides common functionalities for displaying menus, performing actions, and managing user-related tasks.

Trade-Offs

While implementing the HMS, several trade-offs had to be considered. Firstly, we only allowed the patient to select one primary doctor, which becomes their permanent assigned doctor. This decision prioritises providing personalised care by fostering a consistent doctor-patient relationship. However, it introduces a trade-off by limiting patient choice, as the doctor cannot be changed later on. Despite this, the focus of the design was to ensure continuity of care and a deeper understanding of the patient's medical history, reflecting the system's emphasis on personalised treatment. Next, users are given the option to retrieve their user ID by entering their full name, prioritising ease of use. While this minimises friction, it sacrifices security, as it relies solely on the full name instead of stronger authentication methods. Our group's rationale is that this trade-off favours convenience for the users.

Reflections

Difficulties we encountered:

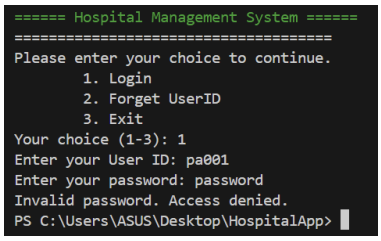
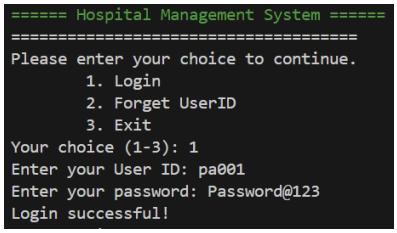
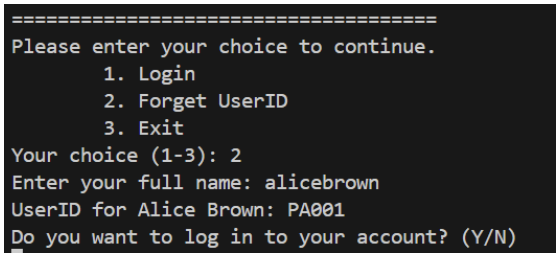
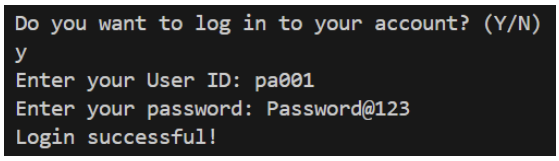
One difficulty we encountered was storing data, as we had to decide how many databases to use. HMS handles diverse types of data, such as patient records, appointment schedules and staff management. One major decision we had to make was whether to store these in separate databases or a single centralised one. Separate databases for each domain could offer clear boundaries and security advantages. However, a single database may simplify management and make it easier to join data across different modules. Ultimately, we decided to use separate databases for each domain.

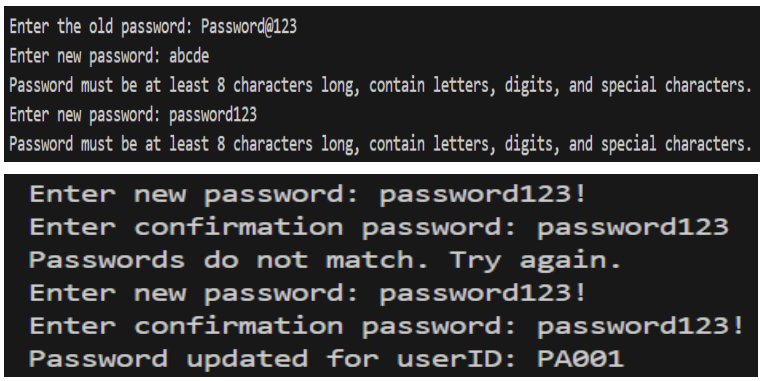
Our team faced challenges in collaborating on the code due to the interdependence of classes in the HMS. Many classes are linked through shared resources, making it hard to isolate tasks. For example, changes to the appointment scheduling feature could impact how patient information is retrieved or updated. As a result, team members had to work simultaneously to ensure compatibility, slowing down individual progress.

Knowledge gained

Through this project, we gained a deeper understanding of OOP principles like encapsulation, inheritance, polymorphism, and abstraction. These principles helped in structuring the system and managing complex relationships between patients, doctors, and appointments. Furthermore, we gained insights into designing systems that are user-friendly for non-technical users, focusing on minimising friction and enhancing accessibility.

Test cases

	Test Case	Results
1	Logging in with new password after changing it	 
2	Retrieving userID from full name (forget user ID)	
3	Logging in directly after retrieving userID	
4	Clearing the console after logging out	Passed.

5	User prompted to re-enter the password if the new password does not meet the requirement	

GitHub Link

https://github.com/lyxm58290/SC2002-HMS-

References

GeeksforGeeks. (n.d.). MVC Design Pattern. GeeksforGeeks. Retrieved November 13, 2024, from https://www.geeksforgeeks.org/mvc-design-pattern/