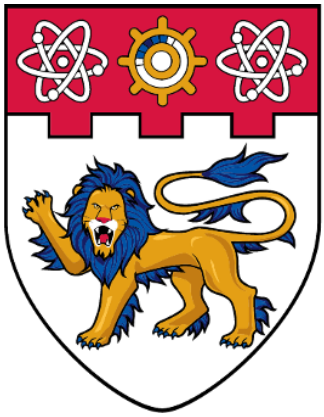


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

SC2002 OBJECT ORIENTED DESIGN & PROGRAMMING

Hospital Management System (HMS)

Report

AY24/25 Sem 1 | SCS1, Group 1

NAME	MATRICULATION NUMBER
Guo Yichen	U2320626C
Li Yikai	U2320525D
Li Zhenxi	U2322971B
Cai Xubin	U2320538H
Liang Jianpeng	U2321126K

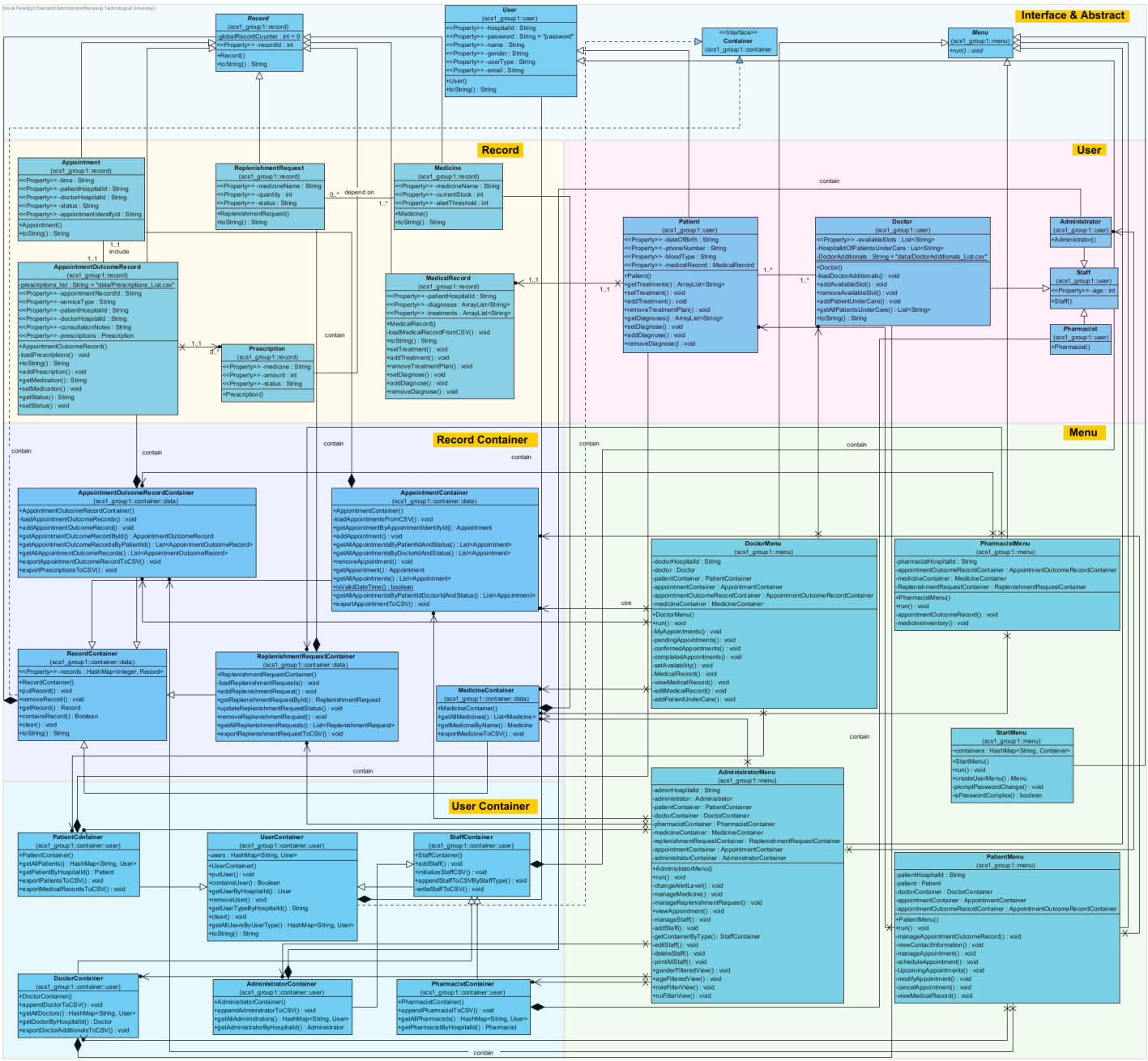
GitHub Main Page:https://github.com/SunnyRaymond/SC2002-HMS

Javadoc Page: https://sc2002-hms-scs1-group1-javadoc.netlify.app/

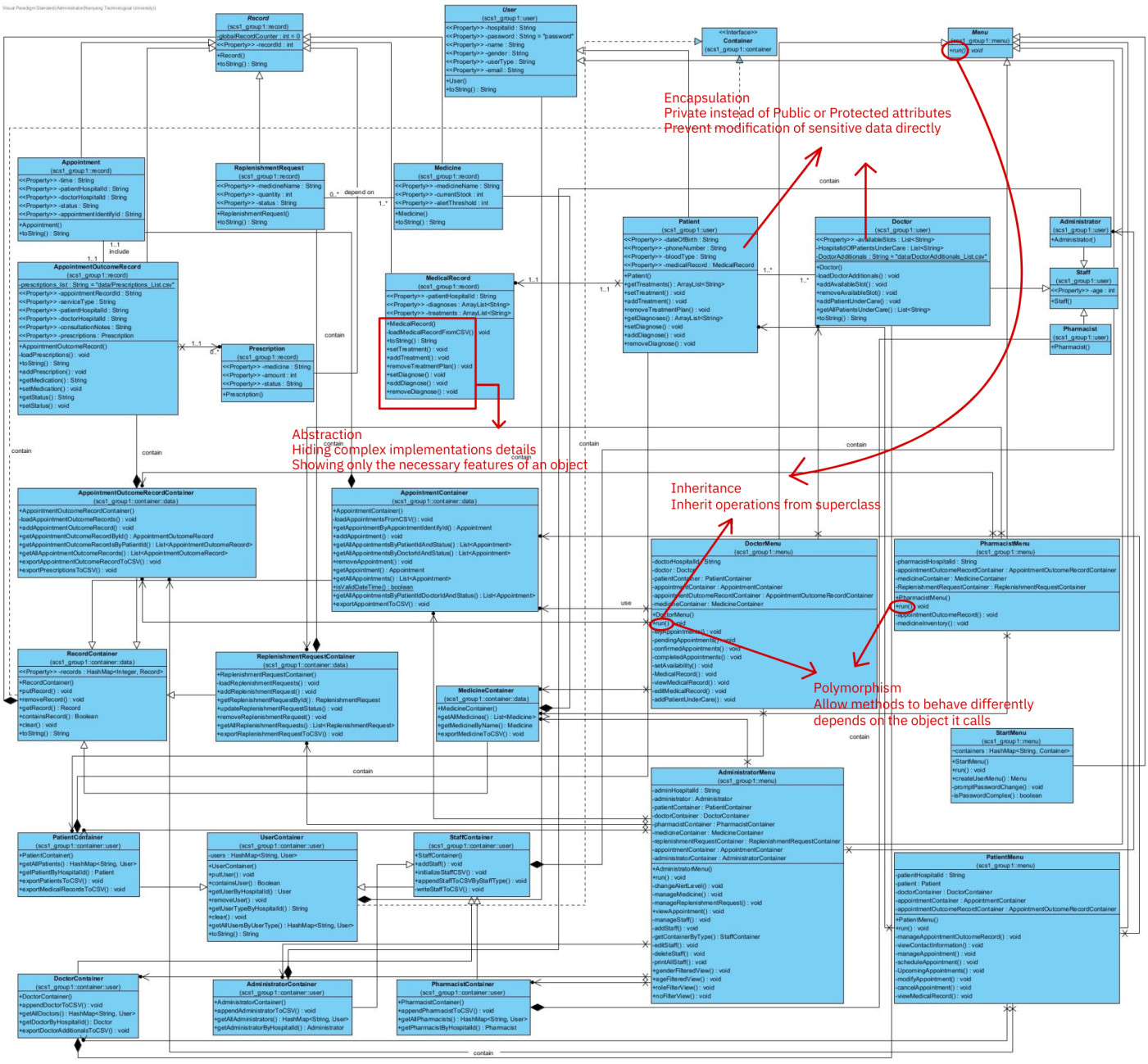
Date of Submission: 2024.11.17

1. UML CLASS DIAGRAM

1.1. DIAGRAM



1.2. ANNOTATED DIAGRAM (OOP)



2. DESIGN CONSIDERATIONS

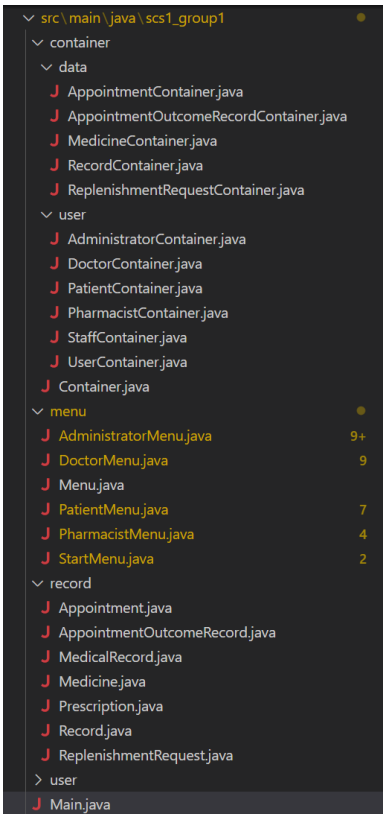
2.1. AIM

Hospital Management System(HMS) is a Java Command Line Interface(CLI) aimed to automate the management of hospital operations, including patient management, appointment scheduling, staff management, and billing. Our system hopes to facilitate efficient management of hospital resources, enhance patient care, and streamline administrative processes.

The system aims to provide role-specific functions and ensure data privacy complying to PDPA. The system is also extensible to adding new roles or functions in the future.

2.2. APPROACH

The structure we use is a modified version of typical software engineering. We have 3 levels: Data, Container and Menu. Data and its corresponding CSV file act as "Entity" and database to store our data. Data storage capability enabled by this structure allows the programme to be terminated without losing the data changes made during runtime.



Containers act as "Controllers" which not only has many control and implementation functions, but also acts as a temporary database to store the data during runtime and store the data back into a CSV file after terminating. Container together with the CSV file ensures maintainability of the programme by having a temporary and permanent data storage.

The implementation of containers also increases extensibility of our programme. Each user type(role) has its own containers. When a new role is needed, we can simply create a new class of container for that role to extend the project without affecting existing classes.

Menu acts as "Boundary", an interface that allows users to perform actions such as accessing appointment outcome records without the users needing to know where it is stored.

2.3. ASSUMPTIONS

Assumptions are:

1. Changes in data will be saved upon exiting programme
2. Data is saved in CSV file
3. Only one user will be using the system at any one time
4. User will only change password in the system after initial login
5. Subsequent password change will only be through changing the CSV file
6. Patient data will be kept and not be deleted
7. Admin will allocate password to newly added staff

8. New patient data will be added from excel

2.4. TRADE OFF

In any System/Software/Programming project, there will be inevitable trade-offs. A common trade off is time-complexity for space complexity or vice versa. Our group project we made several trade offs in data fetching and many other aspects:

Firstly, we trade off space complexity for time complexity by using hashmap instead of lists. During data fetch, we use Hashmap to fetch data more easily. For instance, we used the key: "Patient" to fetch value: patientContainer, key: "Doctor" to fetch value: doctorContainer. In the containers, we also used hashmap to store data temporarily, hash-mapping hospitalID to corresponding user, like hash-mapping "P1005" to patient Alice. This approach allows for efficient data fetching with lower time complexity, however at the expense of space complexity where it consumes more memory than simple arrays or lists.

Secondly, we trade off scalability with simplicity. Our group benefited from the usage of CSV for data persistence from its nature of lightweightedness which is an advantage in simplicity. However, it also meant limited scalability as a large database is difficult to be contained in CSV as compared to traditional SQL or MongoDB which allows for querying and transactional consistency.

Lastly, our group traded off security for convenience. Our group chose to save the data in a CSV file without encryption to prevent further complicating the project, allowing us to have more time to focus on other features/functions which we deem more important for this particular project. Despite not having encryption, our group is not discounting the importance of having encryption for passwords and sensitive data, however, due to the scope of this project, we will omit it.

2.5. SOLID DESIGN PRINCIPLES

2.5.1. SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility Principle (SRP) ensures that each class has one responsibility, making the code easier to modify, test, and reuse.

1. Each container class, for example: "PatientContainer", "StaffContainer" and "AppointmentContainer", is responsible only for managing and storing a specific type of object. They handle data loading, querying, and storing only.
2. Classes like "Appointment", "MedicalRecord", and "Prescription" focus on encapsulating data(attributes) and operations related to those specific attributes without operations.
3. Classes such as "PatientMenu" or "DoctorMenu" are dedicated to user interaction and flow control, keeping UI logic separate from core functionality.

2.5.2. OPEN-CLOSE PRINCIPLE

The Open/Closed Principle (OCP) ensures that the system can be extended without modifying existing code, that is, open for extension but closed to modification.

1. Inheritance and Extension: “StaffContainer” serves as a base class for “DoctorContainer”, “PharmacistContainer”, and “AdministratorContainer”. This allows new staff types to be added without changing the base container logic. Similarly, "UserContainer" is open to extension to add different roles such as “PatientContainer” or other roles in the future if necessary.
2. Polymorphism: For example, the “addStaff” method in “StaffContainer” uses polymorphism to handle different staff types (Doctor, Pharmacist, Administrator) which are the subclasses of “StaffContainer”.

2.5.3. LISKOV SUBSTITUTION PRINCIPLE

The Liskov Substitution Principle (LSP) ensures that derived classes can be substituted for their base classes without altering the correctness of the program. The subclass must be able to provide more function without asking for more information.

1. User and Staff Hierarchy: Subclasses such as “Doctor”, “Pharmacist”, and “Administrator” extend “Staff” and staff extends “User”, ensuring they can be treated as “User” objects. Methods like “getUserByHospitalId” or “putUser” in “UserContainer” can work with any subclass of “User”.
2. For example, the "createUserMenu" method dynamically returns specific menu types (“DoctorMenu”, “PatientMenu”, etc.) based on the logged-in user's type. These menus implement different behaviors while being interchangeable at the higher level.

2.5.4. INTERFACE SEGREGATION

The Interface Segregation Principle (ISP) encourages using multiple, specific interfaces instead of a large, general interface. The main purpose is to avoid having a “God-Interface” that is able to accomplish everything.

1. Instead of designing a single interface for all users, specific responsibilities are distributed. For example: The “getAvailableSlots” and “addAvailableSlot” methods are specific to the “Doctor” class and not available to other roles..
2. Methods like “addPrescription” and “setConsultationNotes” are specific to “AppointmentOutcomeRecord” and not mixed into unrelated classes.
3. By separating container logic (“PatientContainer”, “StaffContainer”) and record logic (“MedicalRecord”, “Appointment”), no class is burdened with unrelated methods and we can achieve role-specific functions .

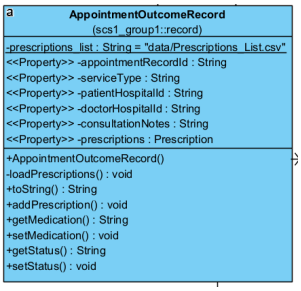
2.5.5. DEPENDENCY INJECTION PRINCIPLE

The Dependency Injection Principle (DIP) states that high level modules should not depend on low level ones. Classes should not depend on other concrete classes, but on higher level interfaces instead.

1. All menus such as DoctorMenu, PatientMenu are dependent on Menu which is an abstract class.
2. Each container relies on the abstraction provided by the "RecordContainer" or "UserContainer" class. For example, adding new methods to handle more types of records or users does not affect the base container design.

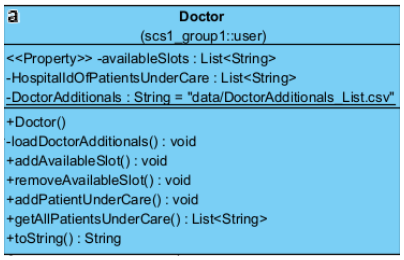
2.6. OO DESIGN PRINCIPLES

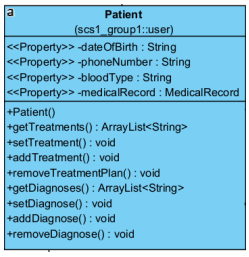
2.6.1. ABSTRACTION

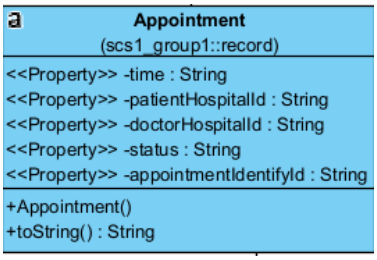


The system employs abstraction to simplify interactions with complex components, focusing on key attributes relevant to the system. Each entity, such as the class shown in the right, "AppointmentOutComeRecord", has methods that expose only necessary functionality while hiding implementation details. For instance, the class provides methods like "getMedication()" and "setMedication()" for medication management, abstracting the underlying detailed implementation of a list. This approach ensures users interact with surface level methods without worrying about deeper level internal data structures and implementation details.

2.6.2. ENCAPSULATION



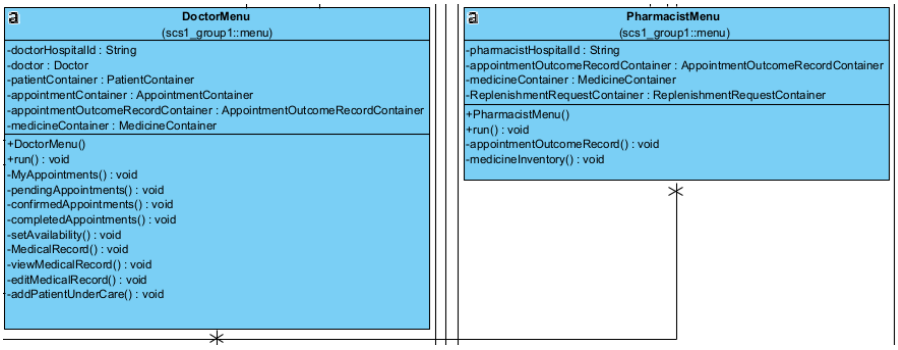




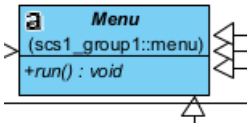
Encapsulation is a key puzzle piece of the system's Object Oriented Design, promoting data integrity and security. All attributes in classes like "Doctor", "Patient", and "Appointment" are private and accessible only through public "getter" and "setter" methods. This prevents unintended modifications and ensures proper validation when data is updated. For example: the "Appointment" class allows only through the "setStatus()" method to modify its status, enabling changes of controlled state. Sensitive information like patient's data is stored securely and only accessible through the methods designed for getting and setting this private data. Hence encapsulation ensures that each class manages its data responsibly, reducing errors and increasing maintainability.

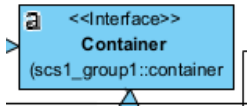
2.6.3. POLYMORPHISM

Polymorphism is implemented extensively to enhance flexibility and scalability:

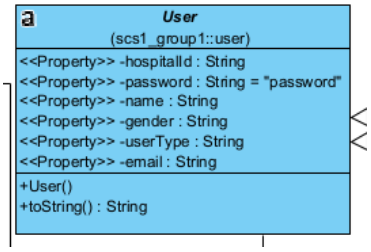


1. The system uses a base "Menu" class for shared behavior among “DoctorMenu” and “PharmacistMenu”. Specific behaviors are overridden in subclasses, such as role-specific methods in the “run()” function of their respective menu classes. This implementation is a display of polymorphism(Dynamic Polymorphism).
2. The “Container” interface serves as a base for all container types (e.g., “AppointmenContainert”, “MedicineContainer”, and “ReplenishmentRequestContainer”, enabling seamless handling of records through polymorphic containers.





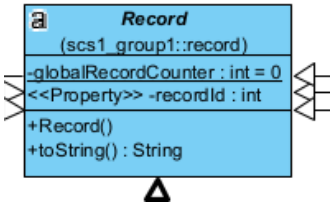
2.6.4. INHERITANCE



Inheritance is extensively used to promote code reuse and extensibility:

The “User” class serves as the base class for specific roles like “Doctor”, “Pharmacist”, and “Patient”. This shared structure ensures consistency in user attributes while allowing subclasses to implement role-specific behavior.

Similarly, the “Record” class provides a unified framework for all record types, such as “Appointment”, “Prescription”, and “MedicalRecord”, enabling them to share common attributes like “recordId” and behaviors like CSV parsing.



2.7. Design Highlights (Additional Functions)

2.7.1. Password Complexity

This function enforces secure password standards for users of the hospital management system. The following rules apply:

Here are some requirements regarding to the length and character composition:

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.

Adding function of checking password complexity brings us advantages in four main aspects:

1. When a mix of characters and a moderate length is required, the system protects against brute force attacks and weak password choices.
2. Many healthcare regulations, such as HIPAA, emphasize the importance of safeguarding patient information. A robust password policy is a step toward compliance.
3. Patients and staff would be more confident given the fact that their data is being protected by a strong password.
4. By restricting the password length to a maximum of 20 characters, the system prevents unnecessarily complex passwords from deterring user compliance.

2.7.2. Appointment Record Filter

This function allows users to filter appointment records based on two key criteria:

1. Filtered by Time: Enables users to view appointments on specific time, to the precision of Day, in format of YYYY-MM-DD, such as 2024-11-18.
2. Filtered by Doctor: Allows users to view appointments associated with a specific doctor.

Addition function of Appointment Record Filter enhances patient experience in four dimensions:

1. By having easy access to past appointments, patients can better prepare for discussions with their doctors according to their appointment history, leading to more effective consultations
2. This function enables patients to keep track of their healthcare data in a better way, fostering a sense of involvement in their treatment and care
3. Patients can quickly identify any discrepancies, such as incorrect dates or doctor assignments, and notify the administration for corrections

3. FUTURE ENHANCEMENTS

The current system saves data in CSV files for data persistence. This approach could be improved to support more robust and scalable storage. For example:

1. Our group can implement a backup system to periodically save CSV files or database snapshots to ensure data is not lost during unexpected interruptions, which is especially important for hospital systems which runs for a prolonged period of time each day
2. The current system stores sensitive data like passwords, patient details in plain text within CSV files. To enhance security, passwords could be encrypted through implementing hashing algorithms to store encrypted passwords, ensuring that they are not directly accessible even if the CSV files are compromised.

4. REFLECTIONS

This assignment has been a meaningful learning experience for everyone in our group, highlighting the significance of object oriented design principles through the development of Hospital Management System which simulates real-world applications. The project has brought about many benefits, however, our group faced several challenges.

4.1. CHALLENGES FACED

Firstly, our initial code versions lacked a well planned structure, which led to cascading issues. A change in a part of the code could cause bugs and problems in other parts of our code, requiring debugging and modifying multiple other parts.

Secondly, our group faced the issue of time management. We needed to strike a balance between integrating additional features, ensuring compactibility of the features with the purpose of the system and leaving sufficient time for testing of the system. It was not an easy task given the limited amount of time and the hope for a “perfect system” which can support functions outside the scope of what is necessary.

Finally, Our group only found out the need for data saving after termination of the program near after we have completed coding all the functions. This required us to make significant changes to the code nearing the deadline.

4.2. LESSONS LEARNED

As cliche as it might sound, through challenges, our group picked up valuable lessons, which benefitted us immensely for future projects/courses.

Firstly, we learnt about the importance of planning. As the Chinese idiom goes: 磨刀不误砍柴工 which means sharpening the axe will not hold up the work of chopping wood – more preparation will speed up the working process. Our group learnt to spend more time on designing the architecture upfront. By doing so, we could have saved us significant time during implementation and debugging stages.

Secondly, our group picked up the skill of time management and work delegation through the challenge of time management. This skill has helped us to better stick to deadlines while not compromising the quality of work. Work delegation also prevents over representation of ideas from a single person when developing the project which could curb creativity and efficiency.

Thirdly, our group learnt to pay attention to details. Through the painful mistake of only finding out the requirement of saving data at the last minute, our group learnt to pay attention to details which will be extremely useful in our future projects and in the workforce. By paying attention to details, we can be more user centric when developing our programme, considering the mandatory functions from the user's point of view.

4.3. CONCLUSION

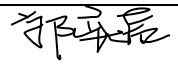
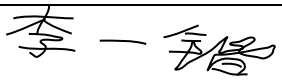
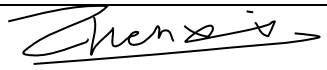
This project has deepened our appreciation for the importance of good Object Oriented Design and Object Oriented Design Principles. Through creating a modular system with high scalability and extensibility, we not only met the project requirements but also established a strong foundation for future improvements. The challenges we faced and overcame have reinforced the value of thorough planning, time management, work delegation and attention to details.With these values, our group believes we will be able to handle future projects with ease.

Declaration of Original Work for SC/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
Guo Yichen	SC2002	SCS1	
Li Yikai	SC2002	SCS1	
Li Zhenxi	SC2002	SCS1	
Cai Xubin	SC2002	SCS1	
Liang Jianpeng	SC2002	SCS1