NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

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 (CE2002 or CZ2002)**	**Lab Group**	**Signature /Date**
Lim Jing Rong	CZ2002	SCSF	 09/11/2024
Loo Ping Wee	CZ2002	SCSF	 09/11/2024
Ng Jing En	CZ2002	SCSF	 09/11/2024
Ong Wee Kiat, Ryan	CZ2002	SCSF	 09/11/2024
Ong Yao Sheng	CZ2002	SCSF	 09/11/2024

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.

Chapter 1 Design Considerations & Tradeoffs

For our Hospital Management System(HMS) we chose the Model-View-Controller pattern, which divides responsibilities into three distinct components— Model, View, and Controller, combined with the Repository pattern. We chose this approach for the following reasons:
Repository Pattern: The Repository pattern is used within the Model layer, where repositories (e.g., UserRepository, AppointmentRepository) abstract data access, provides a centralized way to manage data. It enhances code maintainability, testability, and flexibility, making it easier to work with various data sources.
Model-View-Controller(MVC):The MVC design pattern breaks up our application into three parts- Model (which handles the data), View (which is what the user sees) & Controller(which connects the two). This achieves modularity in our project, making it easier to work on each part separately.

1.1 Object Oriented Principles (OOP)

Encapsulation: Our project maintains encapsulation by ensuring that each layer handles specific task. Models manage data, controllers handle logic, and views control user interface interactions. This separation minimizes interdependencies and allows each layer to evolve independently, making the system better organized and maintainable.

Inheritance: The use of inheritance in the **User** subclasses (e.g., Doctor, Patient) allows shared behaviour and attributes across different types of users.

Polymorphism: Interfaces like **IUserRepository** and **IAppointmentRepository** in the Controller layer allow for polymorphic behaviour, where different implementations can be used interchangeably, supporting flexibility and extensibility.

Abstraction: By using interfaces, the project abstracts the implementation details of data handling. For example, **IUserRepository** abstracts the user data handling, allowing different underlying data stores or techniques to be used without impacting the rest of the system. This level of abstraction supports flexible and maintainable code.

1.2 SOLID design principles

The SOLID design principles is used extensively in our project in order to make our program more understandable and maintainable. It also helps us manage, extend and refactor the code as we collaborated.

1.2.1 Single Responsibility Principle (SRP)

We utilised this principle greatly within our project as seen with the multiple packages within our project. This can be seen in our models package. This package contains Entity classes and their relationships with each other. Examples include **Staff**, **Patient** and **User**.

1.2.2 Open Closed Principle (OCP)

To ensure extensibility of HMS, we utilised the concept of OCP by creating interfaces to enhance loose coupling between classes. We ensure that our interfaces are Open to Extension and Closed

to Modification. This can be seen when we use interfaces, we allow new implementations without modifying existing code.

1.2.3 Liskov Substitution Principle (LSP)

We ensured that our subclass must do all the things super class do and that subclass must not introduce new requirements that are not present in the super class. This can be seen in **AdminRepository** where it extends from **UserRepository** but it does not introduce new requirements to the methods that are already implemented in **UserRepository**.

1.2.4 Interface Segregation Principle (ISP)

The interfaces in HMS are made such that the classes that implements these interfaces uses all the methods defined in the interfaces.
Examples are **IMedicineRepository** and **IPatientRepository** where they are specific CRUD methods that solely focuses on Medicine and patient related CRUD operations respectively.

1.2.5 Dependency Inversion Principle (DIP)

HMS was designed to follow the DIP as high level modules depends on abstractions/interfaces rather than concrete implementations. This is achieved by using interfaces in HMS.
As seen in **AdminController**, it is dependant on **IAdminRepository** which is an interface that is implemented by **AdminRepository**. The high level model in this case is **AdminController** and the low level module in this case is **AdminRepository**. **AdminController** is focusing on the business logic while **AdminRepository** is performing the actual implementation of tasks.

1.3 Extensibility and Maintainability

The MVC pattern inherently supports extensibility and maintainability by separating concerns:
Extensibility: The design makes it straightforward to expand functionalities. For example, adding a new type of user role or a new data entity (like a medical record) would only require implementing new classes or interfaces rather than modifying existing code. This approach minimizes the need for extensive rewrites and reduces the risk of introducing bugs in stable modules.
Maintainability: The project is structured to be highly maintainable. Each class has a single, well-defined purpose, which makes future updates or bug fixes simpler. For example, using separate data loaders (e.g., PatientLoader) keeps repository classes focused on data management rather than data loading, making it easy to modify without impacting other system components.

1.4 Design Trade-offs

Interfaces for Flexibility: Implementing multiple interfaces for repositories and controllers allows for significant flexibility but adds complexity by requiring additional setup and multiple implementing classes. The trade-off we took was balancing the short-term complexity against the long-term benefits of having loosely coupled components, which improves adaptability and ease of testing.
Separate Data Loaders: Introducing classes like **PatientLoader** and **MedicationLoader** was a choice made to decouple data loading from the core logic. While this approach added more classes to manage, it improved modularity, allowing each loader to be modified, separate from other parts of the system. This trade-off aimed to reduce fragility, as loaders can now evolve independently even if data sources change.

1.5 Alternative Patterns Considered

Service Layer Pattern: We considered adding a service layer between controllers and repositories to centralize business logic and make the project structure even more modular. However, the current project's requirements didn't have complex enough interactions between models to justify an extra layer. Keeping the business logic in controllers made the design simpler and more straightforward.
Singleton Pattern: We also considered using the Singleton pattern in the project for certain parts, such as controllers or repositories, where shared access might have been better. However, we chose not to implement this pattern because avoiding Singleton helps to keep dependencies explicit and adaptable, making each controller and repository's instantiation visible and manageable.

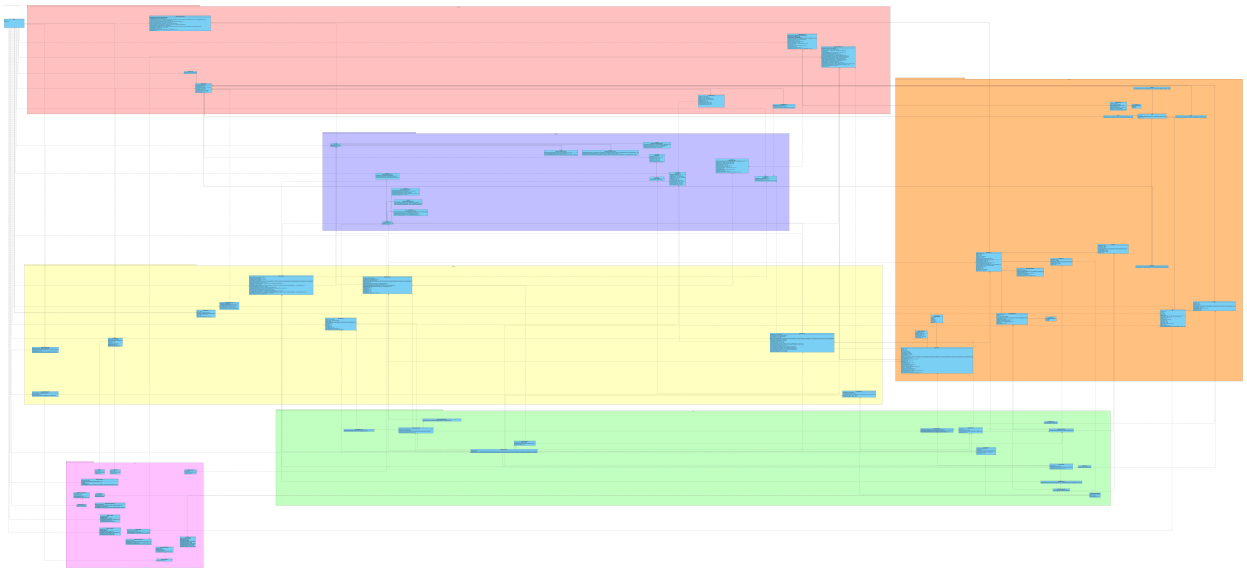
Chapter 2 Additional Features/ Functionalities

Nurse Class: We created a fifth **Nurse** module to focus on patient management, specifically validating, checking and registering new patients.
An example is the **NurseController** which enables nurses to carry out functions such as checking whether the patient exists, whether the patient is valid, and registering new patients. This enhances our project workflow by ensuring all patients are valid before being registered in the system. The controller interacts with the **NurseRepository**, which securely handles data storage, adding new patients to the system only if they don't exist. This class is created to further mimic patient management capabilities in real-world healthcare workflows.

Save to CSV: We implemented the ability to save all the data back to the CSV files making it easier to handle data portability, reporting, and backups. Each exporter class is responsible for saving CSV files for all the data, such as **appointments, medical records, medicines, patients**, and **staff**.
By using a modular approach, these exporter classes handle data retrieval and formatting independently, ensuring that data is consistently updated after each use for each record type. This separation mimics real world scenarios where the database needs to be updated in real time so that they can then be easily shared or analysed outside the application. This feature is especially useful for generating reports, meeting compliance requirements, all of which are important aspects in a real-world operating healthcare system.

Fixed Salt Password Hashing: We decided to implement the **Password Hashing** feature instead of just plain-text password, to significantly enhance security within the project by protecting sensitive user credentials. The **PasswordHasher** class works by converting the user passwords into hashed representations with a fixed salt before they are saved in the system. This feature adds a critical layer of defence against unauthorized data access, simulating the role of user data security in hospital settings.

Chapter 3 UML Class Diagram



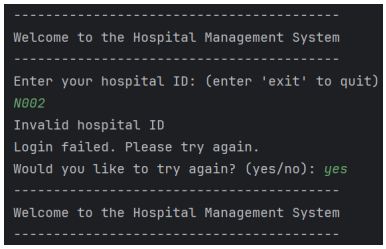
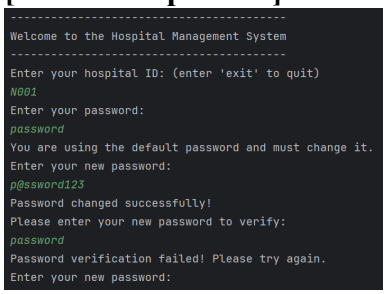
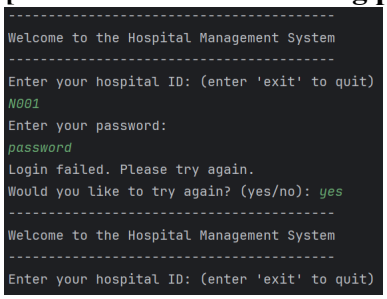
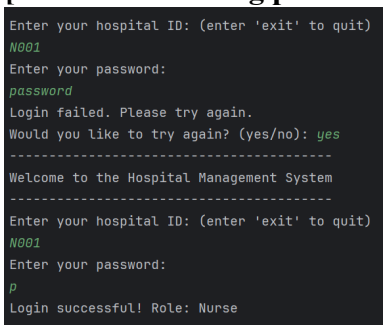
The .vpp and .jpg file for our HMS implementation are attached along with this report.

3.1 Explanation

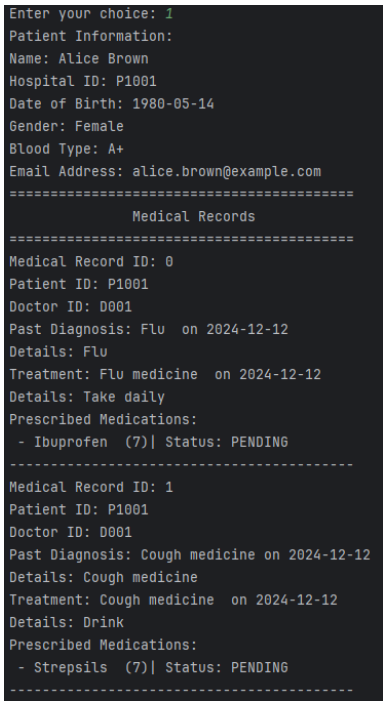
Our project is split into 6 packages, namely Controller, Interfaces, Model (includes enum), Repository, Utils and View to support the MVC principle by organizing code into distinct, purpose-driven packages, each contributing to a clear, modular and maintainable design. This aligns with the OOP practice of **encapsulation** through clear boundaries and modularity, **abstraction** by separating interface definitions and utility functions, **inheritance** by sharing common attributes and methods across related classes, and **polymorphism** by allowing interfaces to define interchangeable implementations.

Chapter 4 Test Cases & Results

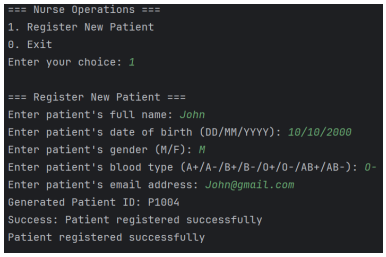
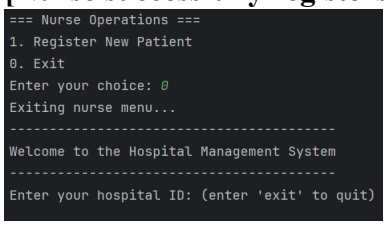
For test cases, we chose to include one from every segment. Additional test cases are attached at the end.

Test Case	Expected Outcome	Result
1. Authentication	User has to login with a valid hospital ID. **[Exception Handling]** If user enters an invalid hospital ID, an error message is displayed, and the user is prompted to try again. The default password that every user has is "password," and the user is then prompted to change their password. **[Exception Handling]** During password verification, if the user enters a password that is different from the one that they have set, an error message is displayed, and the user is prompted to re-enter their new password	 **[Invalid hospital ID]**  **[User enters non-matching password]**  **[User enters wrong password]**  **[User successfully logins]**

INFORMATION ACCESS

Test Case	Expected Outcome	Result
2. View Medical Record	All the Patient's information and medical records are being printed out.	 **[Patient's Medical Record]**

NURSE

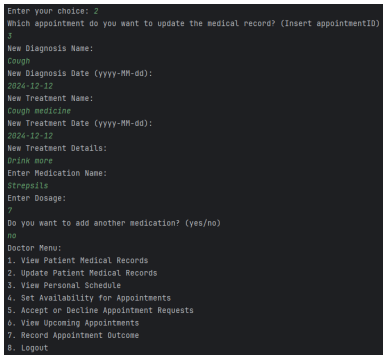
Test Case	Expected Outcome	Result
3. Login as Nurse to register new Patient	Nurse can view the Nurse Menu and choose to "Register New Patient" or "Exit". **[Backtracking]** Entering "0" allows the Nurse to go back to the Hospital Management System Menu. **[Register New Patient]** Nurse is prompted to enter the following details:	 **[Nurse successfully registers Patient]**  **[Returned to the Hospital Management System Menu after entering "0"]**

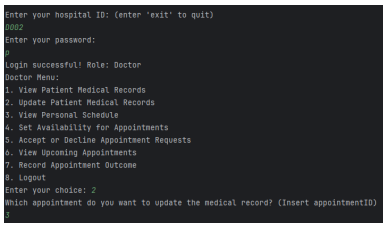
	• Name (String), Date of Birth (DateTime), Gender (String), Blood Type (String), and email address (String) • Error checking is done at every step for invalid inputs.	

APPOINTMENT MANAGEMENT

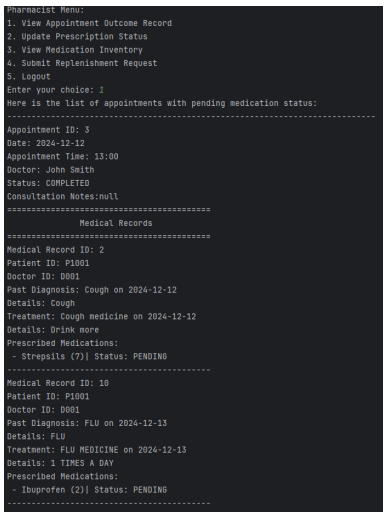
Test Case	Expected Outcome	Result
4. Viewing Scheduled Appointments as Patient	All the appointments with status of PENDING and COMPLETED are being printed together with the appointment details that include: • AppointmentID • Date • Appointment Time • Doctor • Status • Consultation Notes **[Backtracking]** Patient is redirected to Patient Menu after scheduled appointments are displayed	 **[Scheduled Appointments displayed]**

MEDICAL RECORD MANAGEMENT

Test Case	Expected Outcome	Result
5. Update Patient Medical Record as Doctor	Doctor is prompted to enter the appointment ID of the medical record that they would like to update and the relevant details: • New Diagnosis Name, Date • New Treatment name, Date, Details	 **[Updating Patient Medical Record]**

Test Case	Expected Outcome	Result
4. Viewing Scheduled Appointments as Patient	• New Medication Name, Dosage – Doctor is prompted to enter yes/no to add another medication **Error Handling** If doctor enters an appointment ID that does not belong to them, an error message is printed and Doctor is redirected to Doctor Menu	 **[Patient not under Doctor's Care]**

PRESCRIPTION MANAGEMENT

Test Case	Expected Outcome	Result
6. View Appointment Outcome Record	Displays the list of appointments containing the medical records with pending medication status **[Backtracking]** Redirects Pharmacist back to the Pharmacist Menu	 **[List of Appointments and Medical Record with pending medication status]**

Additional test cases here: Github Link

Chapter 5 Reflection

5.1 Difficulties Encountered

Firstly, using GitHub for team collaboration presented the challenge of managing code conflicts as we worked simultaneously. To address this, we adopted good development practices, including incremental delivery. By focusing on priority functions and delivering them in batches rather than waiting for a complete working product, we improved traceability and minimised conflicts.
Secondly, as our system grew, we struggled to maintain a clear separation of concerns. We realised that more classes were needed than originally planned so as to ensure adherence to the SOLID Principle. To address this, we adopted the MVC architecture and created additional packages, such as a repository package for CRUD operations and an utilities package for common functions. This approach improved the modularity and cohesion of our code, making it more extensible and easier to manage.
Lastly, handling of test cases posed a challenge. While writing an extensive amount of codes, we could not ensure every possible edge case was covered, which made the later stage during the testing process difficult and left room for potential errors. We overcame this by including exception handlings to catch unforeseen issues and implementing a more systematic approach to cover edge cases comprehensively, thereby improving the overall reliability and robustness of the code.

5.2 Knowledge Gained from This Course

Throughout this course, we have been exposed to the concepts of OOP. By having the opportunity to develop an application in this assignment, it allowed us to have first-hand experience with these core concepts which helped deepen our understanding. This ensures that we know how to apply them effectively in real-world scenarios. We learned that a well-structured architecture like MVC, not only improves collaboration but also enhances usability, emphasising the crucial role of leveraging diverse packages effectively. Throughout the development process, we also gained a deeper appreciation for the importance of testing and documentation. Writing comprehensive tests and maintaining clear documentation allowed us to verify the functionality of our code and provided a reference for future development or maintenance. All in all, this course imparts essential skills in designing a modular, secure and efficient system that mirrors real-world software design practices.

5.3 Possible Further Improvements

User-Specific Salt for Passwords: Although hashing with a fixed salt is effective, using unique salts per user would further improve security.
Service Layer: This service layer would encapsulate business logic, acting as an intermediary that processes data before it reaches the controller or repository. By adding this layer, the application can maintain a cleaner separation of concerns.
Optimized Data Storage: We can further optimize data storage by splitting the fields and data into separate CSV files. This modularization ensures each file is focused on a specific type of information, such as patient details, prescriptions, or appointments. It enhances data organization and protects against data loss, as the integrity of the remaining files is preserved if one file becomes corrupted or deleted