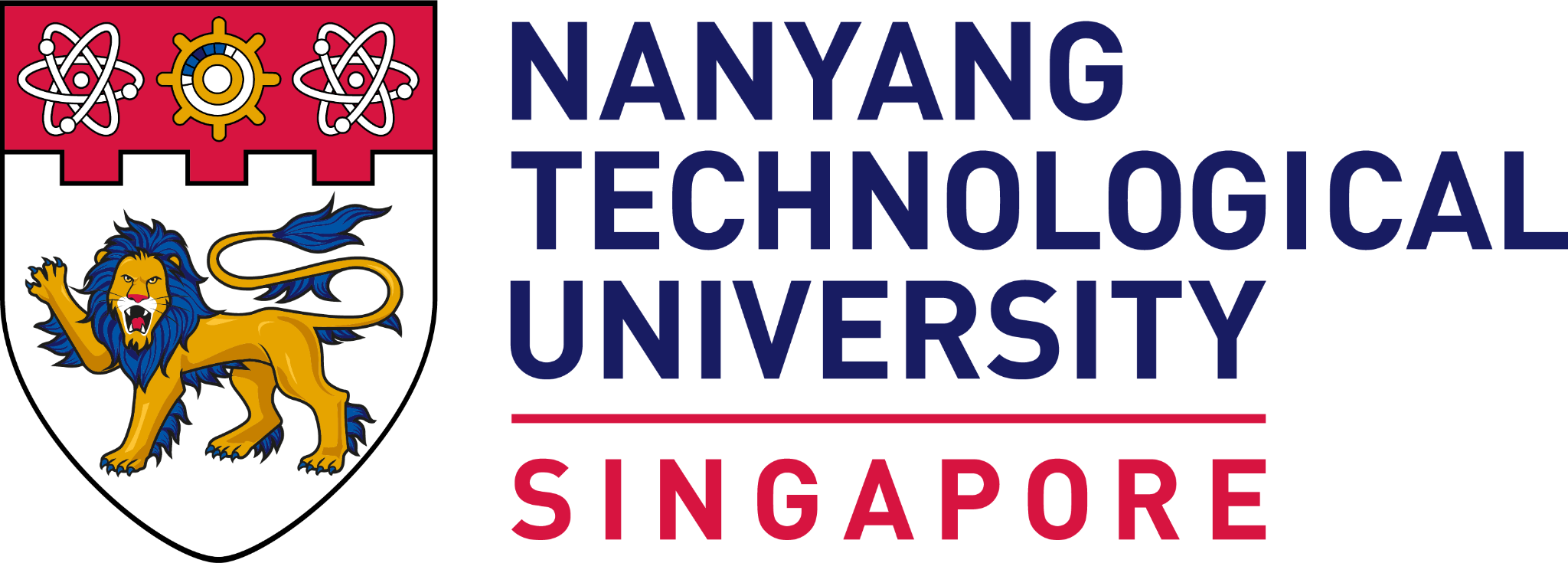
**Nanyang Technological University**

**School of Computing and Data Science**

****

**SC2002 Project Report**

**Hospital Management System**

**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 | Lab Group | Signature /Date |
| Ananthula Harineesh Reddy | SC2002 | SCS6 | 16.11.2024 |
| Dashini Naidu | SC2002 | SCS6 | 16.11.2024 |
| Tan Jin Yong | SC2002 | SCS6 | 16.11.2024 |
| Tay Zhi Xian | SC2002 | SCS6 | 16.11.2024 |
| Valerie Richelle Reivanka | SC2002 | SCS6 | 16.11.2024 |

**1. Design Considerations**

The Hospital Management System (HMS) is a Java command line interface application created for the purpose of arranging or managing appointments between patients and doctors, providing prescriptions for patients, maintaining patients’ medical records, and managing medicine inventory.

**1.1 Design Approach**

The design approach of HMS focuses on simplicity, modularity and scalability around 3 key components: **controllers**, **boundaries**, and **entities.** In the HMS, **controllers** handle the main business logic and act as intermediaries between the interface and core data. Meanwhile, **boundary** classes provide user-specific interfaces, allowing access to relevant features based on the user subclasses. **Entity** classes, on the other hand, encapsulate the core data and logic for different system components, ensuring data integrity and controlling how data is accessed and modified. In this structure, users interact with the system through boundary classes, which relay requests to controllers. Controllers then process these requests, interacting with entities to retrieve or modify data as required before sending results back to the boundaries. This organized flow ensures that the system remains flexible and easy to update, enabling new features to be integrated with minimal disruption.

**1.2 Assumptions**

* All data is kept inside CSV files and must be assigned to an object before use.
* All user IDs are first assigned a default password, which must be changed on the first log in.
* Each Doctor ID is assigned a list of work days, hence no appointments can be made on non-work days.
* All appointments are 30 minutes long, hence each appointment slot will take up 30 minutes.

**1.3 SOLID Design Principles**

Our program follows the SOLID design principle, which is comprised of 5 design principles: Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Injection Principle (DIP). By adhering to the 5 SOLID design principle, our design ensures that each component is maintainable and remains flexible for future modification.

**1.3.1 Single Responsibility Principle (SRP)**

The goal of the Single Responsibility Principle is to ensure that each class is dedicated to a single responsibility and avoid creating a ‘god’ class with multiple jobs. One example in our program is the InitialDataMedicine class. This class's sole purpose is to import and export data from the Medicine\_List.csv, meaning that it will only handle concerns regarding transferring data from and to this specific csv file that contains information regarding the medicines, including medicine name, stock, and low stock alert. This class will not handle data from other csv files. Thus, by focusing solely on Medicine\_List.csv, this class has a clear, well-defined single responsibility.

**1.3.2 Open-Closed Principle (OCP)**  
The Open-Closed Principle ensures that classes are closed for modification, but open for extensions. The goal of this design principle is to add new functionalities or behavior to the class without modifying the classes’ original code. An application of OCP in our program is done through the use of inheritance on the User class, that is a concrete class that provides methods to create a user object. The User class is extended to create multiple subclasses for each types of users, such as the Doctor, Patient, Pharmacists, and Administrator subclass. The initial code of the User class remains unchanged as our program creates different types of users. The use of inheritance in or code allows the subclasses to create additional features beyond the User class’ initial responsibility and extends its functionality.

**1.3.3 Liskov Substitution Principle (LSP)**

The Liskov Substitution Principle is aimed to ensure that the behavior of a subclass is consistent with their corresponding base class. When using the same overridden method, subclasses must ensure to not expect more data or parameters nor should they provide less information than their corresponding base class counterpart. Thus, objects of a base class should be substitutable by objects of the subclasses without affecting the correctness of the program. In our program, the implementation of LSP is visible in the application class App.java, where users are able to log in by entering a unique ID, which serves as the pre-conditions to enter the system. Depending on the user ID entered, the program will identity the corresponding user subclasses and displays the relevant user-interface (UI), which serves as the post-condition. Regardless of which user ID is called, the program will identify which subclass the ID belongs to and identify these subclasses as a replacement for the User class. However, it must be noted that if the unique ID entered is not associated with any subclass, this ID will be deemed as invalid and unrecognized by the system.

**1.3.4 Interface Segregation Principle (ISP)**

The Interface Segregation Principle is a principle which states that classes should not extend from interfaces that contain more abstract methods than what is needed by the class. This key point of this principle is to create more specific interfaces rather than one bloated interface so classes will not be forced to implement unnecessary methods beyond their jobs or responsibilities. Our Hospital Management System implements ISP by utilizing several interfaces, for example, DataExporterAppend which contains appendData() method and the DataExporterWriter interface that contains the writeData() method (to rewrite a csv file). By separating them into two smaller interfaces instead of combining both under a single interface, we allow classes to implement only the behavior that they need. Thus, Classes can realize one of the two interfaces that is relevant to their specific functionality, depending on whether they are required to append data into a csv file or rewrite the file entirely.

**1.3.5 Dependency Injection Principle (DIP)**

The Dependency Injection Principle states that components should depend on abstractions, such as interfaces or abstract classes, rather than concrete and specialized implementations. DIP promotes loose coupling by ensuring that these abstract components are injected into classes to prevent creating a ripple effect to other classes when a minor change is due. In our HMS implementation, DIP can be found on the InitialDatareplenishmentRequest where it implements DataImporter interface and the ListInterface interface. By realizing these interfaces, the InitialDatareplenishmentRequest class took the abstract methods from the two interfaces and created its own implementation of them. If a future modification that requires another type of data to be imported or listed in a different manner, then this change can be easily integrated by creating a new class that realizes the two interfaces without changing any existing code or impacting other classes, hence avoid creating a ripple effect and creates more opportunities for future extensions.

**1.4 Object-Oriented Principles**

To add-ons, the system is also built on fundamental Object-Oriented Programming(OOP) principles such as **Composition, Aggregation, Encapsulation, Abstraction, Inheritance**, and **Polymorphism** to support a clean, maintainable, and scalable design.

**1.4.1 Encapsulation**

In the Hospital Management System (HMS), key classes such as **Users**, **Pharmacists**, **Patients**, **Doctors**, **Administrators**, **Medicines**, and **Appointments** use private fields for sensitive data like personal information, appointment schedules, and prescription details. Public getter and setter methods are created in these private fields to allow other system components to read or modify the data only in a controlled manner, preventing unauthorized access or unintended changes. For instance, the Prescription class uses encapsulation by defining private fields for medicationName, which can be accessed and updated through its respective getter and setter methods, such as getMedicineName(), setMedicineName(). This method ensures data integrity and prevents unauthorized modification of the **Prescription** object’s data.

**1.4.2 Abstraction**

Abstraction in the HMS is demonstrated by creating separate user interface (UI) classes, such as AdministratorUI, DoctorUI, PatientUI and PharmacistUI, which handles user-specific actions and responsibilities while abstracting away the complexity of the system’s business logic. For instance, in the PharmacistUI class, the complex operations like managing inventory and changing prescription status are abstracted into separate classes or control structures. Meanwhile, the Pharmacist user interacts with a simple menu, where the distinct methods displayInventory() and updatePrescriptionStatus() from separate classes are invoked based on user input. This design provides users with easy-to-use display and hides the intricate details of these operations from the user.

**1.4.3 Inheritance**

Inheritance is a key object-oriented principle used in the HMS to promote code reuse and simplify the management of similar classes. The User class serves as a common superclass for subclasses like Doctor, Administrator, Patient and Pharmacist. By inheriting from User, these subclasses automatically gain access to shared attributes and methods, such as displayMenu() and toStrings(). This reduces code duplication, improves maintainability, and ensures that shared functionality is centralized in one place. Furthermore, subclasses can override inherited methods to implement role-specific functionality, such as customizing the displayMenu() method to present different options for Doctors versus Patients. This use of inheritance ensures consistency across user types and simplifies future updates to the system.

**1.4.4 Polymorphism**

Polymorphism is another key object-oriented principle used in the Hospital Management System (HMS) to allow the system to treat objects of different subclasses as objects of the superclass. In the HMS, polymorphism enables the system to interact with different user types (Doctor, Administrator, Patient, Pharmacist) through a common superclass, the User class. Methods like displayMenu() and toStrings() can be called on a User reference, but the behavior of the method will vary depending on the actual object (Doctor, Administrator, Patient, etc.) at runtime and will instead run the specific subclass’ implementation of the method. Using polymorphism keeps the HMS codebase scalable and flexible, allowing extensions of the system to create additional features by creating new user types without changing the overall architecture.

In addition to user classes, polymorphism also simplifies the data import/export functionality. Interfaces like DataImporter, DataExporterAppend define common methods signatures that are implemented differently by specific classes. For example, the InitialDataStaff implements DataImporter, DataExporterAppend, where methods like importData() and appendData() are executed differently depending on the specific implementation for handling Doctor, Administrator and Pharmacist data, ensuring that different types of data can be handled with the same interface and makes the system easier to adapt to future requirements without major structural changes.

**2. UML Class Diagram**

Encapsulation **-** Bluebox

Abstraction **-** Greenbox

Inheritance - Doctor, administrator, pharmacist and patient inherits from Users class

Polymorphism - Purple box

​​ Link to diagram: [Complete UML Class Diagram](SC2002%20UML%20Diagram%20Final.drawio.html)

(If any issues with accessing the file, view SC2002 UML Diagram Final.drawio in the same folder)

**3. Testing**

**3.1 Login System and Password Management**

**3.1.1 First-Time Login and Password Change**

|  |  |
| --- | --- |
| **Interface:** | **Result:**    Patient menu display after successful login and change of password. |
| When User first logs in, they login with a default password and change it. And the new password needs to meet the security requirements. After successful password change they will be logged in. | |

**3.1.2 Login with Incorrect Credentials**

|  |  |
| --- | --- |
|  | When a user logs in with the wrong password, an error message will be printed. And they will be redirected back to the login menu. |

**3.2 Patient Action**

**3.2.1 View Medical Record**

|  |  |
| --- | --- |
| **Interface:**    This is the patient menu | **Result:**    Patient will be able to see their Name, DOB, Gender, Contact Information, Blood Type, and Past Diagnoses and Treatments |

**3.2.2 Update Personal Information**

|  |  |
| --- | --- |
| **Interface:**  Patient have to enter correct email format to change the email successfully | **Result:**    Patient’s email successfully updated |

**3.2.3 View Available Appointment Slots and Scheduled Appointments**

|  |  |
| --- | --- |
| **View Available Appointment Slots** | **View Scheduled Appointments** |
| Patient will be able to choose the doctor that they want to view appointment slot and it will display it available dates and time | Displayed all the pending and upcoming appointments of the patients |

**3.2.4 Schedule and Reschedule an Appointment**

|  |  |
| --- | --- |
| **Schedule an Appointment** | **Reschedule an Appointment** |
| **Interface:**      Patient now can schedule appointments with their desired doctor and after successful booking it will be pending approval from the respective doctor | **Interface:**    Users can select which appointment they want to reschedule. Then send as the interface for schedule appointment they can choose the dates and time |
| **Result:** | **Result:** |

**3.2.5 Cancel an Appointments**

|  |  |
| --- | --- |
| **Interface:** | **Result:** |
| The interface display appointments that are available for cancellation, those that have not passed the scheduled timing. |

**3.2.6 View Past Appointment Outcome Records**

|  |
| --- |
| It will show all the past outcome records of the patients. |

**3.3 Doctor Action**

**3.3.1 View Patient Medical Records**

|  |  |
| --- | --- |
| **Interface:**    Enter 1 to see all the patient medical records. | **Result:**    Will only display medical records of patients who have appointments with you before |

**3.3.2 Update Patient Medical Records**

|  |
| --- |
| Doctors can choose which patient of theirs to update and they can choose to update the diagnoses or treatment and the respective menu will pop up. |

**3.3.3 View Personal Schedule**

|  |
| --- |
| It will display all the available slots of the doctor in the next 14 days |

**3.3.4 Set Availability for Appointments**

|  |
| --- |
| The left side is the interface for doctor update on its availability, start time to end time and working days. The right side is the result after the doctor updates his availability. So the appointment slots will also change based on these details, so on days that he is not working no appointment slot will appear. |

|  |  |
| --- | --- |
| **3.3.5 Accept or Decline Appointment Requests** | **3.3.6 View Upcoming Appointments** |
| It will display all the pending appointments waiting for approval. Then the doctor can choose to accept or decline | Will display a list of accepted appointments |

**3.3.7 Record Appointment Outcome**

|  |
| --- |
| It will display a list of completed appointments, and the doctor can select from the list. And on the right show the interfaces where the doctor can edit the outcome record. |

**3.4 Pharmacist Action**

|  |  |
| --- | --- |
| **3.4.1 View Appointment Outcome Record** | **3.4.2 Update Prescription Status** |
| It will display a list of all the appointment outcome records of every patients, and each appoint outcome record look like the one on the left. | It will display a list of outcome records which have pending medication and each record looks like the one in 3.4.1. Pharmacists can also choose the record they want to update and which medication they want to dispense. |

|  |  |
| --- | --- |
| **3.4.3 View Medication Inventory** | **3.4.4 Submit Replenishment Request** |
| Display all the records of the medication in the inventory. | It will display records of all the medicines that are low in stock and they can choose which they want to replenish and state the quantity. And this request will be sent to the administrator's side for approval. |

**3.5 Administrator Action**

**3.5.1 View and Manage Hospital Staff**

|  |
| --- |
| Top-right shows the interface to manage hospital staff. Administrators can choose to add a doctor for example and the doctor id will be auto-generated based on the past doctor IDs. Administrators can update, remove and view the staff. |

|  |  |
| --- | --- |
| **3.5.2 View Appointment Details** | **3.5.3 View and Management Medication Inventory** |
| Will display all the appointment records | The administrator can update the stock level of the medication inventory. There other features such as adding new medication and updating initial stock are also available. |

**3.5.4 Approve replenishment Requests**

|  |
| --- |
| It will show a list of replenishment requests and the administrator can approve or reject pending replenishment requests from the pharmacists. Also able to approve request or new medication and this will add a new record in the medication inventory |

**4. Reflection**

**4.1 Challenges Encountered and Key Insights**

Throughout the creation process of the Hospital Management System, our group experienced difficulties in managing the workload distribution among the 5 members equally and creating an accurate UML diagram. Our initial UML diagram did not reflects the characteristic required for the Hospital Management System and thus, experienced many stages of revisions even after the code was completed. In the end, through open communication and active discussion, our group was able to complete both the UML diagram and codebase while adhering to the OO design principles to create a maintainable and flexible system.

This project gave us valuable insights into the practical application of design principles. In the early stages of development, we realized that minor changes could create a cascading effect, impacting multiple parts of the system, which highlighted the importance of applying the SOLID design principles and the Entity-Boundary-Control (EBC) pattern for high cohesion and loose coupling. By focusing on these principles, our system is both flexible and maintainable, allowing us to add features and handle role-specific functionalities without excessive rework. In short, this project reinforced the value of thoughtful design, preparing us to build scalable, real-world applications with complex requirements in the future.

**4.2 Further Modifications**

Several enhancements and modifications could further optimize the HMS for better performance, user experience, and scalability. These modifications are designed to address some potential limitations, add new capabilities, and ensure the system remains adaptable to evolving healthcare needs.

**4.2.1. Integration with External Systems**

* **Objective:** Integrate the HMS with external systems, such as laboratory equipment, billing systems, and insurance providers, to streamline hospital operations.
* **Modification:** The system could be extended to automatically receive test results from laboratories, process billing information directly with insurance providers, and generate claims. This integration would reduce manual data entry errors and enhance the overall efficiency of the system.

**4.2.2 AI-Powered Predictive Analytics**

* **Objective:** Leverage data analytics to improve hospital operations and patient care.
* **Modification:** Implement an AI-based predictive analytics module that analyzes historical patient data to predict trends such as patient flow, peak times for appointments, and potential outbreaks of diseases. These insights would enable the hospital administration to make proactive decisions about resource allocation, staffing, and inventory management.

#### **4.2.3. Scalability and Cloud Integration**

#### **Objective:** Make the system scalable to handle larger hospitals or health networks.

#### **Modification:** Migrate the system to a cloud-based platform to ensure scalability, reliability, and reduced hardware dependency. A cloud-based system would also facilitate easy updates, remote access, and disaster recovery. Additionally, it would support the future expansion of the system to multiple hospital locations or networks.

**Advanced Features:** [**Test Cases For Additional Features**](Test%20Cases%20For%20Additional%20Features.docx)

**5. Link to GitHub Repository:** [**Hospital-Management-System**](https://github.com/Dashini16/Hospital-Management-System-HMS-/tree/main)