**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 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  (SC2002/CE2002CZ2002) | Lab Group | Signature /Date |
| Ng Hoe Ping | SC2002 | SCSG | 6/11/2024 |
| Edmund Yeo Zi Long | SC2002 | SCSG | 6/11/2024 |
| Angelina Delia Sutiarto | SC2002 | SCSG | 6/11/2024 |
| Spyridon Giakoumatos | SC2002 | SCSG | 6/11/2024 |
| Wu Yiqing | SC2002 | SCSG | 6/11/2024 |

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.

# **1. Design Considerations**

## **1.1 Approach Taken**

To enhance the user-friendliness of the data, promote flexibility, and ensure data security, we have taken some approaches such as coupling and cohesion, delegation, encapsulation, and error handling.

### **1.1.1 Coupling and Cohesion**

To facilitate system modifications, we aim for low coupling and high cohesion among classes. Low coupling minimizes dependencies, allowing each class to perform distinct tasks independently, making it easier to modify and maintain the system. For example, to ensure low coupling, instead of the Patient class directly interacting with the database, it relies on PatientAppointmentDB for data operations, ensuring a clear separation of responsibilities. In addition, each class maintains a cohesive role. For instance, PatientAppointmentDB manages appointment-related database tasks while the Patient class handles patient activities and interface interactions. This focused, logically grouped structure improves code organization, maintainability, and modularity.

### **1.1.2 Delegation**

To promote flexibility, and reduce complexity, we prioritized delegation over inheritance. Deep inheritance can introduce unnecessary complexity by forcing subclasses to inherit unwanted functions. Delegation, on the other hand, ensures that each class only has the functions it needs, enabling changes without affecting others. For example, rather than inheriting from an Appointment class, the Patient class delegates scheduling tasks to the PatientAppointment class, which specializes in appointment handling. This way, any updates to appointment functions only require changes to the PatientAppointment class, leaving the Patient class unaffected.

### **1.1.3 Encapsulation**

To safeguard data, we apply information hiding by exposing only essential parts of a class. In our system, classes like Patient and Doctor handle user interactions, such as scheduling appointments, while dedicated classes like PatientAppointment and DoctorAppointment manage underlying implementations. For instance, when a user schedules a new appointment via the Patient class, the actual data handling occurs in the PatientAppointment class. This separation keeps the Patient class focused on user interactions, hides scheduling details, and simplifies the system.

### **1.1.4 Error Handling**

To ensure a smooth user experience and handle errors, we used try-catch blocks for input validation. InputMismatchException and IndexOutOfBoundsException catched invalid inputs and prompted users to re-enter choices, while IOException managed issues when reading or writing XLSX files.

## 

## **1.2 Principles Used**

We have designed the classes in adherence to SOLID principles to maintain scalability, flexibility of code, and to allow for modifications with minimal impact to other methods/classes. This ensures readability and maintainability by making the code modular and cohesive.

### **1.2.1 Single Responsibility Principle (SRP)**

Each role-based class (Doctor, Pharmacist, Patient, Administrator) is responsible and handles functionalities only for its own role. This means that each class can only do things related to its class and will not have more than a single reason for modification. Furthermore, in accordance with this principle, other supporting classes such as Medical Record are made separately, to maintain modularity in the code and ensure low coupling with high cohesion.

### **1.2.2 Open-Closed Principle (OCP)**

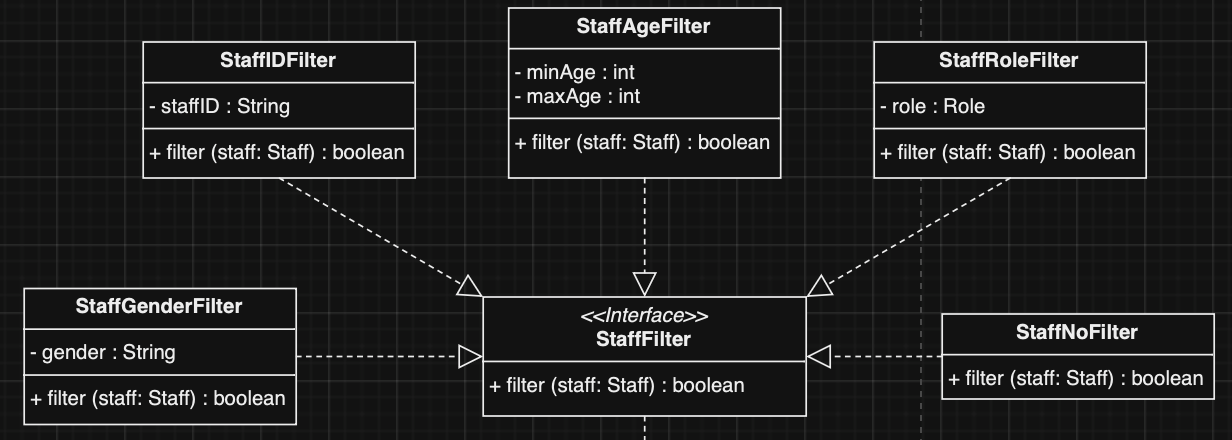
OCP describes the need for extension without modification. As our role-based classes are all subclasses of the superclass, User, it offers a flexible foundation for future role-based classes that may be needed, such as the “logout” function. As such, we are able to allow for extension to multiple other role-based classes without having the need to modify any other classes, ensuring scalability. In addition, role-specific functions can be simply extended to the specific role class without having to modify other classes.

### 

### **1.2.3 Liskov Substitution Principle (LSP)**

In our design, the User superclass provides a common shared functionality, such as “logout”, which is inherited by all subclasses. This means that any instance of subclass or the superclass can be used interchangeably. This principle is used to ensure the core functionality of the user is maintained throughout all the subclasses.

### **1.2.4 Interface Segregation Principle (ISP)**

Following the ISP which mentions that many client specific interfaces are better than one general purpose interface, we have designed the “StaffFilter” class to be flexible and modular. As shown in the snippet of our UML diagram below, this approach allows clients to only interact with the interface they require. This modular design allows new filter functionalities to be easily added by implementing new classes derived from the “StaffFilter” class, which also promotes readability. This also allows the classes to not inherit more than necessary other than the functions it was created to do.

### 

### 

### 

### 

### **1.2.5 Dependency Injection Principle (DIP)**

In essence, our project adheres to this principle by ensuring that individual role-base classes do not interact with the database directly. For instance, the “DoctorAppointment” class, which is a high-module class, depends on low level modules such as “DoctorSchedulesDates” and “PatientScheduledAppointment” to handle the core logic process, which abstracts the data handling detail from the class. It also leaves data access, such as reading and writing of the XLSXfile, to “DoctorAppointmentDB”, ensuring that the logic remains flexible by having an intermediary class.

## **1.3 Additional Features Added**

|  |  |  |
| --- | --- | --- |
| No. | Description | Illustration |
| 1 | Register new patient |  |
| 2 | Informative Prompts | Pharmacist’s prompt:  Doctor’s prompt:  Admin’s prompt: |
| 3 | Exit process anytime in any function |  |

## **1.4 Assumptions Made**

|  |  |
| --- | --- |
| No. | Assumption Made |
| 1 | Existing user will always remember their hospital ID and password |
| 2 | Doctor will always accept/reject before the start date of the appointment. |
| 3 | Doctor will always update the Appointment Outcome Record for the patient after every appointment. |
| 4 | Doctors will always set availability, so that patients are able to book an appointment |
| 5 | All data files are stored in the XLSX format, with correct data types for its columns |
| 6 | Phone number, Email and blood type entered will always be valid |

# **2. UML Class Diagram**

# 

# **3. Test Cases**

|  |  |
| --- | --- |
| Steps taken for Test Case | Result |
| Test Case: 2,1   1. Login as patient P1001 2. Update personal information    1. Phone: 12345678    2. Email: abcd@gmail.com 3. View Medical Record |  |
| Test Case: 12,3,4,7,5,6   1. Login as Doctor D001 2. Set availability    1. Set date 10/11/2024    2. Set time 09:00 to 10:00 3. Set availability    1. Set date 11/11/2024    2. Set time 12:00 to 13:00 4. Login as Patient P1001 5. View available appointment slots 6. Schedule an appointment    1. Choose Appointment 10 (10/11/2024 09:00 to 10:00) 7. View scheduled appointment 8. Reschedule an appointment    1. Reschedule Appointment 10 (10/11/2024 09:00 to 10:00)    2. Schedule Appointment 11 (10/11/2024 09:00 to 10:00) 9. Login as Patient P1002 10. View available appointment slots 11. Login as Patient P1001 12. Cancel Appointment 11 | Doctor Setting Appointment POV:    Patient (P1001) Viewing and Setting Appointment POV:    Patient (P1001) View scheduled appointment    Patient (P1001) rescheduled appointment    Patient (P1002) view appointment slots    Patient (P1001) cancel appointment |
| Test Case: 13,11,14   1. Login as Patient P1001    1. Schedule appointment 10 (10/11/2024 09:00 to 10:00)    2. Schedule Appointment 11 (10/11/2024 09:00 to 10:00) 2. Login as Doctor D001    1. Accept appointment 10    2. Decline appointment 11 3. View Personal Schedule 4. Login as Patient P1001 5. View schedule appointment | Doctor (D001) Accept/Decline Appointment    Patient (P1001) View schedule appointment |
| Test Case: 15,8   1. Login as Doctor D001 2. Record Appointment Outcome    1. Choose Appointment 10    2. Input services: “XRAY”    3. Input medicine: “AMOXICILLIN”    4. Input notes: Rest more 3. Login as P1001 4. View Past Appointment Outcome Record | Doctor (D001) Record Appointment Outcome    Patient (P1001) View Past Appointment Outcome Record |
| Test Case: 10,9   1. Login as Doctor D001 2. Update Patient Medical Record    1. Choose Patient P1001    2. Choose “Add New Diagnosis”    3. Input Description: “Scoliosis”    4. Input Treatment: “XRAY”    5. Input notes: “Regular Checkups” 3. View Patient Medical Record    1. Choose Patient P1001 | Doctor (D001) Update Patient Medical Record    Doctor (D001) View Patient Medical Record |
| Test Case: 16, 17   1. Login as Pharmacist P001 2. View Past Appointment Outcome Record 3. Choose Patient (P1001) 4. Update Prescription Status 5. Choose Patient (P1001) 6. Input Appointment ID: “8” 7. View Appointment Outcome Record 8. Choose Patient (P1001) 9. View Medication Inventory | Pharmacist (P001) View Past Appointment Outcome Record    Pharmacist (P001) Update Prescription Status    Pharmacist (P001) View Past Appointment Outcome Record    Pharmacist (P001) View Medication Inventory |
| Test Case: 18, 19, 23   1. Login as Pharmacist P001 2. View Medication Inventory 3. Submit Replenishment Request 4. Select “AMOXICILLIN” 5. Input added Stock Level: “20” 6. Press Enter to continue 7. Login as Administrator A001 8. Approve Replenishment Requests 9. Select “Pending” 10. Input Request ID: “4” 11. Select “Approve” 12. Select “No” 13. Approve Replenishment Requests 14. Select “Approved” 15. View and Manage Medication Inventory | Pharmacist (P001) View Medication Inventory    Pharmacist (P001) Submit Replenishment Request    Administrator (A001) Approve Replenishment Requests    Administrator (A001) Approve Replenishment Requests    Administrator (A001) View and Manage Medication Inventory |
| Test Case: 20   1. Login as Administrator A001 2. View Hospital Staff 3. Select the appropriate filters      1. Verify by syncing the data with the no filter list 2. Manage Hospital Staff and Select “Add Hospital Staff”    1. Input Name: “James Tan”    2. Select “Administrator”    3. Select “Male”    4. Input Age: “32”    5. Input Phone Number: “98765432”    6. Input Email: “[jamestan@email.com](mailto:jamestan@email.com)” 3. Verify in View Hospital Staff 4. Manage Hospital Staff and Select “Update Hospital Staff”    1. Input StaffID: “A002”    2. Select “Name”    3. Input New Name: “Jerald Low” 5. Verify in View Hospital Staff 6. Manage Hospital Staff and Select “Remove Hospital Staff”    1. Input StaffID: “A002” 7. Verify in View Hospital Staff | Administrator (A001) View Hospital Staff  No Filter: By Administrator: By Doctor: By Pharmacist:    By Male: By Female: By Age (25-35):    Administrator (A001) Manage Hospital Staff    Add: Update: Remove: |
| Test Case: 21   1. Login as Administrator A001 2. View Appointment Details | Administrator (A001) View Appointment Details |
| Test Case: 22   1. Login as Administrator A001 2. View and Manage Medication Inventory 3. Select “Add Medication”    1. Input Medicine Name: “PENICILLIN”    2. Input Initial Stock: “50”    3. Input Low Stock Level Alert: “10” 4. Repeat Step 2 to check for the newly added medicine 5. Select “Remove Medication”    1. Input Medicine Name: “PENICILLIN” 6. Repeat Step 2 to check for the newly updated list 7. Select “Update Stock Level”    1. Input Medicine Name: “IBUPROFEN”    2. Input Restock Amount: “10” 8. Repeat Step 2 to check for the updated initial stock level 9. Select “Update Low Stock Level Alert”    1. Input Medicine Name: “IBUPROFEN”    2. Input New Low Stock Level Alert: “20” 10. Repeat Step 2 to check for the updated low stock level alert | Administrator (A001) View and Manage Medication Inventory    Add: Remove:    Update Stock Level: Update Low Stock Level Alert: |
| Test Case: 25, 26   1. Select Login    1. Input Hospital ID: “A002”    2. Input Password: “pass” 2. Re-enter login details    1. Input Hospital ID: “A002”    2. Input Password: “password”    3. Input New Password: “admin123” | Main Menu Section (Login)    Incorrect Credentials: First-Time Login & Password Change: |

# **4. Reflection**

## **4.1 Lack of knowledge**

In this project, our team encountered multiple instances of gaps in our technical knowledge, which led us having to find an alternative or having to spend several hours learning the technical skills required. Key areas where we lacked expertise include:

* Learning GitHub branching, pull requests, and conflict resolution was challenging. Tasks like creating sub-branches and reversing commits slowed our workflow.
* Database management was an unventured territory. We retained the provided XLSX format for data readability but discovered midway that Java lacks built-in XLSX support. Converting to a Maven project to include Apache POI for file handling was difficult, as most members were new to dependency management and Maven. To make matters worse, there were unclear instructions from the professor nearer to the submission date, which led us having to convert the file to csv format and use serializer to read/write data. There were also mixed instructions given; in NTULearn, the professor said that maven could be used for compiling external dependencies, but when emailed, she replied “We require students to manage dependencies manually rather than relying on modules”. This led us to doing double work for everything as we had to reformat away from maven.
* Creating and updating the UML class diagram was challenging due to limited knowledge and evolving project requirements. This caused delays, as understanding relationships and specifications between classes took time.

## 

## **4.2 Trouble splitting task equally**

It was often hard to allocate tasks as our tasks were often vague due to unclear objectives. For instance, the “PatientScheduleAppointment” class needed access by both the “DoctorAppointment” and “PatientAppointment” classes, causing conflicts over shared class implementation. Dependencies between classes made parallel work difficult, slowing our overall progress.

## **4.3 Following the SOLID principle**

Following SOLID principles, especially ISP and DIP, was challenging as we were inexperienced in writing specific, modular code. Early code structural issues led to multiple refactorings, which improved our understanding of DIP but required numerous restructuring to avoid redundant code.

## **4.4 Future Improvements**

We recognise the need for detailed pre-planning, especially in regards to UML and adhering to the principles. It not only allowed us to deepen our knowledge of OOP, but also helped us realize the importance of scalability and maintainability in the system. In addition, it emphasizes on how important communication between coders is for shortening the coding process.