# SC2002 OOP Report

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/CE2002 CZ2002)	Lab Group	Signature /Date
Chiu Yeow Keng	SC2002	SCEB	13/11/2024
Chung Wai Kit Jared	SC2002	SCEB	13/11 /2024
Chea Yuan Sheng	SC2002	SCEB	13/11/2024
Ghate Harshal Shrikant	SC2002	SCEB	13/11/2024

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.**

GITHUB:

Section A: Design Considerations

During the development of this project, we tried to develop the application closely based on what was taught in the lectures of this module. As such, the project was designed with these considerations:

- The use of OO Concepts
- Ensuring software qualities are met (Reusability, Extensibility, Maintainability, Low Coupling and High Cohesion)
- SOLID framework

In the section below, we will go through each consideration, and how it was applied in the context of the project.

The project follows OO concepts during the creation of this project. With this, the project was easier to update, debug and extend as we further develop the project with functionalities.

1. Encapsulation
    a. Entity classes manage their own attributes. To access these attributes, the entity class provides their own get/set methods, which exposes only what is necessary.
    b. To ensure encapsulation in entity classes, the classes have methods that delegate responsibility of logic to Managers through implementing methods like this:

    ```
    public void viewAvailableAppointmentSlots(Doctor doctor) {
        appointmentManager.viewAvailableSlots(doctor);
    }
    ```

        i. This design will help in hiding the internal logic from the entity class itself.
2. Inheritance and Polymorphism

    ```
    public class Patient extends User implements IUser
    ```

    a. By using inheritance, subclasses are able to inherit shared attributes and are also able to extend them further (e.g. User, Staff and its subclasses)
    b. Methods are also capable of interacting with any user subtype through the IUser interface, which supports polymorphism.

Following these qualities are crucial as it allows the project to be more efficient in updates, especially when there are additional functionalities that could be added into this project. By adhering to this, it also simplifies the process of debugging, due to the distinct responsibilities that were made for each class.

1. Reusability
   a. Through the implementations of interfaces and abstract classes, the codes are able to be used across different classes in the project.
   b. Since the codes were able to be used across different classes, it promotes reusability as methods are not required to be rewritten again for each implementation, instead, reused - thus reducing duplications.

2. Extensibility
   a. The abstract classes like User, allows new User types to be added to the application without changing existing codes.
   b. Managers were also introduced into this project, enabling extensibility by allowing new features to be added without affecting the original code logics.

3. Maintainability
   a. Each class has their own distinct roles, with entity classes like Patient, Doctor to handle data attributes, while Manager classes (PatientManager, DoctorManager) are able to handle processing of logic.
   b. Naming conventions are followed consistently throughout the entire project, making it easy to identify and understand the purpose.

4. Low Coupling
   a. The project has high dependency on interface classes instead of concrete classes, which in return, reduces dependencies between classes.
   b. With the project being low coupling, it will minimize the impact of changes in one, to another.
   c. Managers classes are also capable of operating independently, reducing cross-class dependencies.

5. High Cohesion
   a. Each class has a distinct set of responsibilities, ensuring each class has a cohesive purpose.

b. Similar to the previous explanations, Manager classes are designed to handle tasks which are specific to themselves, e.g. AppointmentManager handles appointments only, without extending the logic to other domains not related to it.

The application is also developed with adherence to the SOLID principles.

1. SRP
   a. The design approach ensures that each class has only one single reason to change.
      i. Each User class (Pharmacist, Doctor, Administrator, Patient) handles only their own attributes, maintaining a clear responsibility.
      ii. Each User class has a Manager class (PharmacistManager, DoctorManager, AdministratorManager, PatientManager, etc.) are responsible for their respective tasks.
      iii. Other medical tasks also have their own Manager class (MedicineManager, AppointmentManager, PrescriptionManager, etc.)
   b. For example, if we want to change a logic regarding appointments, we can modify the codes on AppointmentManager, without impacting other classes such as Patient and Doctor directly.
   c. For specific medical managers, they encapsulate their own medical tasks, preventing crossovers in responsibilities.

2. OCP
   a. User class is implemented as an abstract class, preventing modification of shared attributes (id, pwd, name, gender), which is used throughout the different subclasses.
      i. By making the User an abstract class, it prevents attributes from unintended modifications. Subclasses which utilize the User class will be able to implement their own features, without altering the User structure.
      ii. In the User class, an abstract method - displayMenu() is also initialized, allowing the subclasses to implement their own unique Menus.
   b. Another abstract class, Staff, is also implemented with their unique set of attributes (role, age), which are not needed for the Patient class.
      i. By implementation, if a new Staff role were to be introduced, it can inherit

from the Staff class enabling it to require the same attributes and abstract methods, without changing any logic.

    ii.   Through this implementation, a subclass which inherits Staff will have attributes - id, pw, name, gender, role, age - which will be consistent across the subclasses.

  c. The introduction of interfaces for classes will also introduce flexibility in implementations, allowing preservation of current logic.

3. LSP

  a. Subclasses like Doctor, Patient, etc. implements IUser.

  b. Any IUser implementation can substitute another, supporting flexibility.

4. ISP

  a. The interface classes introduced in this project only include methods that are required by other classes.

    i.   Avoids imposing unrelated methods on classes, prompting a modular design overall.

    ii.   Through this, it allows a clear distinction of what each interface can or can not do.

    iii.   One example would be in IMedicineManager, addMedicine() method is being initialized, as it is being called in CSVImportManager. This keeps the behavior constant and predictable, without needing to know the specific implementations.

5. DIP

  a. Higher level modules rely on interfaces rather than concrete classes. For example, the subclasses of Users interact with methods declared in IUser, instead of the specific user types.

  b. Overall, it reduces coupling and improves flexibility in adding or modifying components without requiring to modify multiple files when one is changed

Section B: UML Diagram

For the design of our UML Class Diagram, we have split our classes into 3 different categories, namely, Control, Entity and Boundary. Since we have designed the application to have a single

responsibility for each class, it was obvious and clear what class belongs to what category. We will briefly explain what each category is meant for, and for a clearer overview and image of the UML Class Diagram, we will attach it in a separate document.

For simplicity and clarity, we have removed the interface classes from the Class Diagram, and we will assume that all Managers are directly referenced. Please view it in the folder.

The boundary class handles the interaction between the user and the system, which in our project, would be the Main class. The Main class serves as an entry point for the system, enabling users to interact via menus and inputs. It serves as a delegation of user requests to the respective control classes based on the user input.

In our Control Class, it contains all of the Managers. They are the classes which are responsible for logic and implementation, used by other classes. For example, PatientManager would deal

with patient-related operations, which interacts with the Entity, Patient. By doing so, it ensures that boundary classes also do not interact directly with entity classes, which reduces coupling and improves flexibility.

For the entity classes, they hold the responsibility of data, without user interaction or logic implemented. Through this, they are able to encapsulate data of the system, only accessible through get and set methods. Throughout the entire application, no boundary class directly interacts with entity classes.

Section C: Feature Implementation

To reduce the odds of human error throughout the application, we introduced a simple yet efficient method of user input - via the use of indexes.

Looking from the perspective of the users, users may find it difficult to remember certain information (e.g. as a Patient, if the patient wants to create an appointment but do not know/remember any details of a Doctor, how could they proceed?).

So instead of asking users for an input, we display all the information that the application has, e.g. when Patient wants to create a new appointment, the details of Doctor and their respective TimeSlot will be printed out, allowing the Patient to have easy reference to what they need.

From there, the Patient is only required to input the index indicated together with the output to proceed with the workflow. This implementation is consistent throughout the entire application that requires an input of information that is already in the application. (Not new information e.g. addMedicine or addStaff)

Section D: Testing

A short summary of test cases will be shown in this section. For a more detailed test case, refer to the document in the submitted folder.

	Test Case	Expected Outcome

Patients Functionalities		
1	Test Case 1: View Medical Records	Records for the Patient that is logged in (P1001) to be shown. Information provided should be in a clean slate, given that no information has been modified.
2	Test Case 2: Update Personal Information	Inputs provided should be updated and reflected in ViewMedicalRecords.
3	Test Case 3: Invalid phone number	An error will be prompted, asking for a numeric input.
4	Test Case 4: View Available Appointment Slots	Available slots will be shown (project will have added in TWO appointment slot for D001)
5	Test Case 5: Schedule an Appointment	Inputs provided should allow Patient to schedule an appointment with the Doctor
6	Test Case 6: Reschedule an Appointment	Inputs provided should allow Patient to reschedule an appointment by choosing a different TimeSlot
7	Test Case 7: Cancel an Appointment	Inputs provided should allow Patient to cancel the appointment
8	Test Case 8: View Schedule Appointments	Output should show Patient's scheduled appointments. If its empty, it should be reflected.
9	Test Case 9: View Past Appointment Outcome Records	Patient should only be able to see Appointments that are in the past (before today's date) or Status = "Completed"
Doctors Functionality		
10	Test Case 10: View Patient Medical Records	Initial state should be empty, as the Doctor has not accepted any appointments made by the Patient. Upon accepting a Patient's appointment (Test Case 14), it should reflect the Patient's Medical Record here.

11	Test Case 11: Update Patient Medical Records	Initial state should be empty, as the Doctor has not accepted any appointments made by the Patient. Upon accepting a Patient's appointment (Test Case 14), it should show an index to select patient to update their Medical Record
12	Test Case 12: View Personal Schedule	Initial State should be empty, but application is pre-programmed with TWO appointment slots. It should only reflect available schedules.
13	Test Case 13: Set Availability for Appointments	Inputs provided should create a new TimeSlot for the Doctor's schedule. It should be reflected in Test Case 12.
14	Test Case 14: Accept or Decline Appointment Requests	Inputs provided should change a Patient's appointment status to either Accepted or Declined.
15	Test Case 15: View Upcoming Appointments	Initial state should be empty, upon accepting a Patient appointment in Test Case 14, it should be reflected here.
16	Test Case 16: Record Appointment Outcome	Initial state should be empty, as Doctor has not accepted any appointments made by Patient. Upon accepting Patient's appointment (Test Case 14), it should show an index to select patient to update their Appointment Outcome. Results will be reflected in Test Case 10.
Pharmacists Functionality		
17	Test Case 17: View Appointment Outcome Record	Initial state should be empty, as no appointment outcome have been made yet. Upon Test Case 16, it should be reflected here as well.
18	Test Case 18: Update Prescription Status	Inputs provided should change a Prescription status from "Pending" to "Dispensed". Medicine quantity should be updated accordingly.

19	Test Case 19:View Medication inventory	Output of Medicine Inventory should be shown to the caller. Information provided should consist of: Name of Medicine, Quantity, Alert Level
20	Test Case 20: Submit Replenishment Request	Inputs provided should create a Replenishment Request, IF alert level > stock quantity. This will be reflected in Test Case 23
Administrators Functionality		
21	Test Case 21: View and Manage Hospital Staff	A new menu will be provided to the caller, allowing the caller to CRUD Hospital Staff.
22	Test Case 22: View Appointment Details	Initial state should be empty, as no Patient has scheduled any appointments.
23	Test Case 23: View and Manage Medication Inventory	A new menu will be provided to caller, allowing the caller to CRUD Medicines
24	Test Case 24: Approve Replenishment Requests	Initial state should be empty, as no replenishment request have been made. Upon Test Case 20, it should be reflect here, and approval will change status from "Pending" to "Approved".
Login System and Password Management		
25	Test Case 25: First-Time Login and Password Change	Upon logging in for the first time, User will be prompted for a password change.
26	Test Case 26: Login with Incorrect Credentials	Upon invalid credentials given, User will be prompted for Username and Password

Section E: Reflection

When the first project requirement came out, the requirements looked deceivingly simple. We were able to quickly draft up our first UML diagram with separate classes and identify what methods are required for each class and started with implementation fairly quickly. At this point, we were not exposed to the SOLID framework nor the essential software qualities.

As time went by, we were required to make certain changes, either due to errors or refactoring of code, we met a huge hurdle in the modification of code. Due to the amount of dependency each class had with each other, a ripple effect of modifications were required every time something changes. In addition, the initial application only had around 13 classes, which had attributes and all the methods and logic of their respective class.

This implementation did not adhere to the Single Responsibility Principle, which was eventually taught in the lectures. Therefore, there was a huge pain everytime we had to modify a code. After being exposed to the information of the SOLID framework through the lecture class, it was obvious that the initial structure of the application was nowhere near adhering to it.

As a result, a huge overhaul was required, with an addition of 16 new classes, of which, 6 of them were interface classes and 10 Manager classes. In those classes, we tried to adhere as much as possible to the SOLID framework, resulting in a clearer, more flexible, scalable and maintainable system. Since the application was developed in different phases, we also had to ensure that there was no duplication in logic throughout the entire project.

To ensure that the application worked smoothly, we went through testing, to ensure the application worked as expected, with proper exception handling and clear boundaries between the different User roles. As such, the application was able to be improved upon based on feedback, which further improved the reliability and predictability of the application.

To allow this application to be deployable in a real life scenario, additional features can be considered, such as Register Patient, introducing a proper Graphical User Interface (GUI) and deploying the application together with a database for persistent data storage. These features

would benefit all stakeholders of the application. The Register Patient feature could allow addition of new Patients, without requiring the access of Patient_List data, the GUI will provide an easier workflow visually, improving user experience. And lastly, with the implementation of a database, modifications can be altered directly to the database with a query, instead of losing the information upon a fresh start of the application. (Alternatively, we can overwrite existing information each time an update is made on the data.)

In conclusion, through this project, it highlighted the importance of software engineering practices and applying it from the beginning of the development phase. The experience allowed us to apply what was taught in the module, both in programming and designing, resulting in an application that is maintainable and extensible for the foreseeable future.