





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
Yeo, Ruizhi Carwyn	SC2002	SCSJ	 21/11/24
Khoo Ze Kang	SC2002	SCSJ	 21/11/24
Lam Yan Yee	SC2002	SCSJ	 21/11/24
Malcolm Fong Cheng Hong	SC2002	SCSJ	 21/11/24

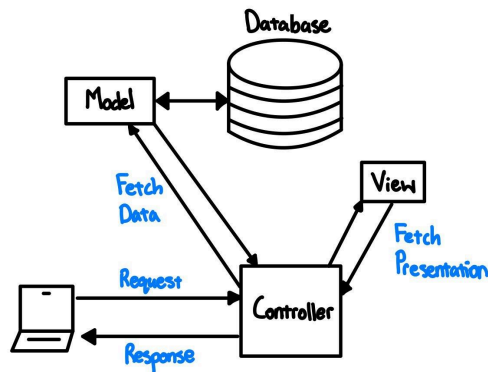
Important notes:

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.

Design Considerations

Approach Taken

We used the Model-View-Controller (MVC) architecture.



This diagram illustrates the Model-View-Controller (MVC) architecture. Model interacts with the database to fetch and manage data. View is responsible for the presentation layer, fetching presentation details from the controller. Controller handles user requests, processes data through the model, and sends responses back to the view for display. Data Flow starts from User requests go to the controller, which fetches data from the model, then sends it to the view for rendering, and finally responds to the user. This architecture promotes separation of concerns, making the codebase easier to maintain and scale.

Other Approaches Considered

Singleton Pattern:

Initially, we considered using the Singleton pattern for managing system-wide resources (e.g., logging or configuration). However, the pattern was avoided due to its potential negative impact on testing and its overuse in certain areas. The system instead relied on dependency injection to manage resources, which is more testable and adheres better to the SOLID principles.

Observer Pattern:

The Observer pattern was also considered for the handling of real-time updates between the view and the model. While it would be helpful in some use cases (such as when a patient's appointment is updated in real-time), the simplicity of the CLI interface did not warrant its implementation in the initial version. It could be a useful addition if the system later requires real-time functionality.

TRADE-OFFS

In any system or software project, trade-offs are inevitable. Our group project involved several:

1. Time vs. Space Complexity

We prioritised faster data retrieval by using hashmaps instead of lists, and mapping keys like "P001" to patient data. While this reduced time complexity, it increased memory usage compared to simpler data structures.

2. Simplicity vs. Scalability

We used lightweight CSV files for data storage, which simplified implementation but limited scalability compared to databases like SQL or MongoDB, which handle larger datasets more effectively.

3. Convenience vs. Security

To focus on core features, we stored data in unencrypted CSV files, trading security for development simplicity. While encryption is important, it was deemed beyond the scope of this project.

These trade-offs were made to balance efficiency, simplicity, and project constraints.

SOLID Design Principles

Single Responsibility Principle (SRP)

In our project, SRP is demonstrated by the SchedulerBoundary class, which is solely responsible for managing the scheduling of appointments for patients.

By focusing exclusively on this task, the scheduling logic remains self-contained and unaffected by other responsibilities, such as managing doctor records or patient data. This separation allows updates or changes to the scheduling process without impacting other parts of the system.

Clearly isolating this responsibility enhances flexibility, reduces dependencies, and improves maintainability, ensuring that changes in one area are less likely to cause issues elsewhere.

Open-Closed Principle (OCP)

In our project, OCP is demonstrated by the “ViewAppointmentOutcomeBoundary” class, which is responsible for displaying appointment outcome records. It is designed to be extendable, allowing new features to be added without changing its core logic. For instance, if we need to display additional details like MRIScan results, we can extend the class without altering its existing functionality.

By following OCP, we keep the system flexible, enabling new features to be introduced without disrupting existing behaviour. This design supports system evolution while ensuring that current functionality remains unaffected, isolating changes to avoid side effects.

The class’s structure promotes loose coupling, separating the business logic for appointment outcomes from how new features are integrated. This encourages easy updates and scalability, improving maintainability and reducing bugs, ensuring a stable and adaptable system.

Liskov Substitution Principle (LSP)

In our project, LSP is demonstrated by the “User”, “Administrator”, and “Doctor” classes. The User class defines common properties like UID, fullName, email, etc. While the “Administrator” and “Doctor” classes inherit these attributes and add their specific properties, such as dateOfCreation for the “Administrator” and dateJoined for the “Doctor”. These additions don’t affect the inherited methods from the User class.

For example, we can pass either an Administrator or Doctor object to a method expecting a User object without issues:

```
User admin = new Administrator(fullName, username, email, phoneNo, passwordHash,
DoB, gender, "Admins", dateOfCreation);
```

```
User doctor = new Doctor(fullName, username, email, phoneNo, passwordHash, DoB,
gender, dateJoin );
```

This demonstrates that both “Administrator” and “Doctor” objects can be treated as “User” objects, and the getFullName() method works for both without modification.

By adhering to LSP, our system allows “Administrator” and “Doctor” objects to replace “User” objects without errors, ensuring stability and flexibility in the codebase. New roles or user types can be introduced without breaking existing functionality.

Interface Segregation Principle (ISP)

In our project, ISP is demonstrated by breaking down functionality into smaller, focused interfaces like “Identification”, “Authenticable”, “Contactable”, “PersonalDetails”, and “RoleAssignable”. This avoids creating a large interface with unrelated methods, ensuring that classes only implement the interfaces they need.

For instance, the “User” class implements the “Authenticable” interface for authentication methods, “Contactable” for email and phone methods, and “PersonalDetails” for personal information such as name and birthdate. Each interface focuses on a specific responsibility, allowing the User class to implement only what is necessary for its role.

Similarly, an “ExternalVendor” class might implement only the “Contactable” interface, as it only requires contact information methods, without needing to handle authentication or role assignments. This approach reduces unnecessary dependencies, making the system more flexible and maintainable. By adhering to ISP, we ensure that classes evolve independently, with less risk of unused methods affecting functionality, leading to a cleaner, more adaptable system design.

Dependency Inversion Principle (DIP)

In our project, the DIP is demonstrated by the design of the “DoctorBoundary” and “MainBoundary” classes. These classes focus on high-level application logic and depend on the “Boundary” abstraction rather than low-level details like user input handling or data display. This separation allows the business logic to remain independent of how data is presented or interacted with, promoting flexibility.

For example, the “DoctorBoundary” class manages doctor-related tasks but delegates the responsibility for displaying information and handling user input to abstract methods in the “Boundary” class. This ensures that changes in the way data is presented or user interaction is handled do not affect the core logic within these classes.

By adhering to DIP, we achieve loose coupling between components, making the system more flexible and maintainable. This design allows for updates to low-level behaviours (like input or display methods) without impacting the high-level logic, reducing the risk of bugs and supporting future scalability.

Abstraction

The boundary classes, such as AdministratorBoundary and LoginBoundary, interact only with the necessary information through their respective methods. By handling only the relevant data and abstracting the implementation details, these classes keep the system simple, secure, and easier to maintain.

Encapsulation

Encapsulation ensures an object's private data is protected by restricting direct access. In the User class, sensitive attributes like name, role, and password are set to private, making them accessible only through public getters and setters, preventing direct modification.

Inheritance

Inheritance allows the subclasses Pharmacist and Doctor to inherit methods from the User superclass, such as setFullName() and getEmail(). This enables both subclasses to reuse and access common functionality defined in the User class, promoting code reusability and reducing redundancy.

Polymorphism

Polymorphism allows the 'Pharmacist' and 'Doctor' subclasses to inherit methods from the 'User' superclass, like 'setFullName()' and 'getEmail()'. This means both subclasses can use these methods, but they can also override them if needed to implement specific behaviour, providing flexibility while maintaining a common interface.

Assumptions made:

1. Initial data is read from the files, nowhere else
2. A maximum of 999 entries in each CSV File contain data
3. Users can use the same email to create multiple accounts
4. Patients know their specific blood type
5. Patient must have had a diagnosis, to have information about their treatment plans.
6. The hospital ID users use to log in to the system and their hospital ID in the repository differ. (username for logging in, hospital ID for referencing.)
7. Doctors can only View and Update patient records, after at least one complete appointment outcome record
8. Doctors can only accept one appointment at a time
9. The first appointment scheduled will show up as the first one for the doctor to accept
10. Doctors cannot edit their diagnosis/treatment plan/prescription after making one, they can only add new diagnoses/treatment plans/prescriptions
11. Doctors will not create a duplicate appointment availability
12. Administrators cannot add, delete, or update another administrator

13. Users will only change passwords once, which is the first time they log in
14. Additional feature billing system does not directly handle, payment. It only records and handles the process after the payment is completed at a hypothetical counter. (Assumes payment is done outside of the system)
15. Difference between Personal Schedule and View Upcoming Appointments (In Doctor Menu):

Personal Schedule: It is like a calendar for the doctor which marks both his past and present appointments, and its function is to check for his timetable availability and history.

View Upcoming Appointments: Its function is more to help the doctor take note of where is the appointment, what time, who and what he should prepare specifically.

Additional Features:

1. Change Password Requirements, when the user changes their password it has to fulfill our security requirement (8 characters, consisting of uppercase and lowercase letters, numbers and 1 special character)
2. Billing/Payment System, which records and handles completed payments.
3. New Allergies attribute added for Patient
4. Patients can view their Date of Admission (Autogenerated when they register an account) and add their allergies
5. The patient can edit their birthday, name, and allergy fields
6. Doctors can view the Date on which they joined (Autogenerated when they register an account)
7. Pharmacists can view their Date of Employment (Autogenerated when they register an account)
8. Expiry Date attribute added for Medicine
9. Patients can view and acknowledge their rejected appointment requests.

UML Class Diagram

Because the file is too big, Please refer to the UML Class Diagram attached to this document or in Github at the following link:

<https://github.com/carwynyeo/SC2002/tree/main/UMLClassDiagram>

Github Link : <https://github.com/carwynyeo/SC2002/tree/main>

Testing

We extracted a few test cases and screenshotted how they work as per below:

Login and Registration System

Test Case 1: Registering an Account

You would like to register as: 1. Patient 2. Doctor 3. Pharmacist 4. Administrator 5. Back to Main Menu Enter your choice: 1 Enter full name: Patient Enter email: patient@gmail.com	Enter phone number: 91239123 Enter date of birth (yyyy-MM-dd): 2024-11-21 Enter gender (M/F): F Enter allergies (if any): peanuts Enter BloodType: O+ Enter desired username: Patient Patient registered successfully with username: Patient Patient registered successfully!, your default password is: password
--	--

Test Case 2: First-Time Login and Password Change

You would like to login as: 1. Patient 2. Doctor 3. Pharmacist 4. Administrator 5. Back to Main Menu Enter your choice: 1 Please enter your username: Patient	Please enter your password: password PATIENTS Patient logged in successfully. It appears that you are using the default password. Please change your password for security reasons. Enter new password: Testtest1! Confirm new password: Testtest1! Password updated successfully for Patient Password changed successfully!
---	--

Test Case 3: Log in with Incorrect Credentials

You would like to login as:
1. Patient
2. Doctor
3. Pharmacist
4. Administrator
5. Back to Main Menu
Enter your choice: 2
Please enter your username:
Doctor
Please enter your password:
Testtest2!
Login failed: Invalid username or password.

Patient Actions

Test Case 4: Schedule an Appointment

```
--- Available Appointment Slots ---
Doctor ID      : D002
Doctor         : Doctor
Day            : THURSDAY
Date           : 2024-11-21
Time           : 11:11
Location       : Level 2 - Heart Clinic

Enter the Doctor ID you want to schedule an appointment with:
Enter Doctor ID (e.g., D001, D002): D002
Validated Doctor ID: D002
Enter the date for the appointment (yyyy-MM-dd):
2024-11-21
Enter the time for the appointment (HH:mm):
11:11

--- Schedule Appointment for Patient ID: P002 ---
Schedule Appointment Menu:
1. Schedule Availability for Appointments
2. Back to Patient Dashboard
Enter your choice: 1
Enter your availability. Type 'no' when finished.
Are you sure you want to schedule appointment? (yes/no)
yes

--- Scheduled Availability Summary ---
Doctor ID      : D002
Doctor         : Doctor
Day            : THURSDAY
Date           : 2024-11-21
Time           : 11:11
Location       : Level 2 - Heart Clinic
```

Test Case 5/6: View Medical Record and View Past Appointment Outcome Record

```
--- Patient Medical Records for Patient ID: P002 ---
+-----+
| Medical Record |
+-----+
| Doctor ID      : D002 |
| Patient ID     : P002 |
| Patient Name   : Patient |
| Patient DOB    : 2024-11-21T00:00 |
| Patient Gender : F |
| Blood Type     : O+ |
| Phone Number   : 91239123 |
| Email          : patient@gmail.com |
+-----+
| Diagnoses: |
+-----+
| Diagnosis ID   : DIAG-006 |
| Description    : fever |
| Prescription Date : 2024-11-21T17:57:59.993719 |
| Medications: |
| Medicine ID    : M000 |
| Quantity       : 5 |
| Dosage         : 2 |
| Period (days) : 5 |
| Status        : Pending |
+-----+
| Treatment Plan: |
| Treatment Date  : 2024-11-21T18:00:15.499036 |
| Treatment Description: medicine |
+-----+

--- Past Appointment Outcomes ---
Appointment Outcome ID: AO-004
Patient ID: P002
Doctor ID: D002
Appointment Time: 2024-11-21 11:11
Type of Service: General consultation
Consultation Notes: fever
Appointment Outcome Status: COMPLETED
Prescription Details:
Medicine ID: M000
Quantity: 5
Dosage: 2
Period (days): 5
Prescription Status: Pending
```

Doctor Actions

Test Case 7/8: View Personal Schedule and View Upcoming Appointments

```
--- Appointments for Dr. Doctor (ID: D002) ---
Appointment Record:
- Appointment ID: A-007
- Date & Time: 2024-11-22 11:11
- Location: Level 2 - Heart Clinic
- Status: CONFIRMED
- Patient ID: P002

Appointment Record:
- Appointment ID: A-006
- Date & Time: 2024-11-21 11:11
- Location: Level 2 - Heart Clinic
- Status: COMPLETED
- Patient ID: P002

--- Upcoming Appointments for: Doctor (UID: D002) ---
Day: FRIDAY, Time: 2024-11-22 11:11, Location: Level 2 - Heart Clinic, Patient ID: P002
Patient Information:
UID: P002
Full Name: Patient
Username: Patient
Email: patient@gmail.com
Phone No: 91239123
Date of Birth: 2024-11-21T00:00
Gender: F
Role: Patients
Allergies: peanuts
Date of Admission: 2024-11-21T17:25:35.867953
```

Pharmacist Actions

Test Case 9/10: View Appointment Outcome Record and Medicine Inventory

```
=====
Appointment Outcome for Record ID: AO-006
Patient ID: P002
Doctor ID: D002
Diagnosis ID: DIAG-007
Appointment Time: 2024-11-22T11:11
Type of Service: General Consultation
Consultation Notes: Chicken Pox
Appointment Outcome Status: COMPLETED
Prescription Details:
Medicine ID: M001
Quantity: 7
Dosage: 1
Period (days): 7
Prescription Status: Pending
=====
Full Inventory - Monitoring stock levels:
Medicine ID: M000
Name: Paracetomal
Stock Level: 61
Low Stock Level: 20
Expiry Date: 2026-12-12T12:00
Medicine ID: M001
Name: CHICKEN
Stock Level: 95
Low Stock Level: 20
Expiry Date: 2026-12-12T12:00
=====
Expired Medicines:
No medicines are expired.
=====
Medicines Below Low Stock Level:
All medicines are above the low stock level.
=====
```

Administrator Actions

Test Case 11: View and Manage Hospital User

```
Listing all staff of type: DOCTORS
=====
===== Doctor Details =====
=====
Personnel Details:
=====
UID          : D002
Full Name    : Doctor
Username     : Doctor
Email        : doctor@gmail.com
Phone No     : 81238123
Password     : Testtest1!
DOB          : 2024-11-21T00:00
Gender       : M
Role         : Doctors
=====
Date Joined   : 2024-11-21T17:26:29.711551
=====

Listing all staff of type: PHARMACISTS
=====
===== Pharmacist Details =====
=====
Personnel Details:
=====
UID          : PH001
Full Name    : Pharmacist
Username     : Pharmacist
Email        : pharmacist@gmail.com
Phone No     : 92349234
Password     : Testtest1!
DOB          : 2024-11-21T00:00
Gender       : F
Role         : PHARMACISTS
=====
Date of Employment : 2024-11-21T17:29:08.310856
=====
```

Test Case 12/13: View and Manage Medication Inventory and View and Manage Billing Information

```
Listing all medicines:
UID: M000
Name: Paracetomal
Manufacturer: NTU
Expiry Date: 2026-12-12T12:00
Inventory Stock: 61
Low Stock Level: 20
Replenish Status: REQUESTED
Replenishment Request Date: 2024-11-21T19:08:30.305141
Approved Date: 2026-12-12T12:00

UID: M001
Name: CHICKEN
Manufacturer: NTU
Expiry Date: 2026-12-12T12:00
Inventory Stock: 95
Low Stock Level: 20
Replenish Status: NULL
Replenishment Request Date: 2026-12-12T12:00
Approved Date: 2026-12-12T12:00

=====
Payment Records for Patient ID: P002
Payment Amount: 150.0
Payment Status: ACTIVE
Created At: 2024-11-21T19:14:07.882638
Updated At: 2024-11-21T19:14:07.882647
=====

=====
Payment Records for Patient ID: P002
Payment Amount: 150.0
Payment Status: ARCHIVED
Created At: 2024-11-21T19:14:07.882638
Updated At: 2024-11-21T19:20:10.721286
=====
```

Reflection

Difficulties Encountered: Challenges and Solutions

1. Implementing Role-Specific Functionalities

Challenge: Each user role (Patient, Doctor, Pharmacist, and Administrator) required distinct functionalities such as managing appointments, inventory, or staff. These functionalities had to be implemented to maintain the simplicity and usability of the system.

Solution: We leveraged the Model-View-Controller (MVC) architecture by designing specialised controllers and views for each role. For example, the `'PharmacistBoundary'` managed pharmacist-specific functionalities such as submitting replenishment requests, while the `'AdministratorBoundary'` handled staff and inventory. Role-specific models encapsulated data attributes relevant to each user type. Controllers were modularised to handle only the logic relevant to their role, ensuring a clean separation of concerns.

2. Data Persistence Without External Storage

Challenge:

In the absence of databases or JSON/XML files, ensuring that data persisted across application restarts was a significant challenge. This became particularly difficult as the size and complexity of the data grew. Without a traditional database, we needed an alternative solution that allowed for scalable data persistence without relying on external storage.

Solution:

To simulate persistent storage, we leveraged CSV files as a method of reading, loading, and saving data. Instead of using external databases, we implemented in-memory data repositories within the Model layer, using collections like `HashMap` and `ArrayList` to hold data during runtime. At the end of each session, we serialized the in-memory data into CSV files, storing them in a structured format. Upon application startup, the system reloads the data from these CSV files, effectively mimicking persistence by restoring the state of the application. This method allowed us to simulate the persistence of data without relying on complex databases, ensuring that the system could handle and test a reasonable volume of data. While this approach was not as robust or scalable as a full database solution, it provided a practical way to persist data for the project's needs.

3. Input Validation and Error Propagation

Challenge: Validating user input in a CLI environment and providing actionable feedback without cluttering the interface proved to be complex.

Solution: Implemented centralised validation logic in the Controller layer to streamline the process. Controllers intercepted erroneous inputs and returned clear error messages to the View layer. For example, initially, invalid appointment dates were detected in ``AppointmentController`` and prompted with specific instructions to re-enter correct values.

Finally, we decided on creating a ``Validator`` class that is designed specifically for validating user inputs. So it handled user inputs all across the program. We made the methods in the class static and public so that they could be reused wherever in the code we needed user input.

Knowledge Learnt

1. Effective Data Management Without Databases

Designing pseudo-repositories in the Model layer taught the team:

- How to use data structures like ``HashMap`` for efficient data retrieval.
- The significance of handling edge cases, such as duplicate entries or concurrent updates, in non-persistent storage.
- Practical workarounds for mimicking persistence without compromising system reliability.

2. Robust Error Handling Techniques

The project showcased how to integrate error handling seamlessly:

- Centralised validation logic minimised code duplication across Controllers.
- Standardised error messages improved the user experience and reduced confusion.
- The use of exception handling in Controllers ensured predictable system behaviour even during unexpected errors.

3. Scalability and Modularity

Building the system with modular components demonstrated:

- How well-designed modules allow independent development and testing.
- The value of reusable components like a shared ``Validator`` for validating user inputs.

- How modularity aligns with open-closed principles, enabling future extensions without altering existing code.

Further Improvement Suggestions

1. Enhance User Interface

Upgrading the CLI interface could significantly improve usability:

- Add search and filtering options for data-heavy views like patient lists.
- Incorporate visual enhancements like colour coding for alerts or warnings.
- Add a user-specific latest update feature that notifies users when they log in.
- Design a text-based dashboard summarising key statistics for each user role.
- explicit manual refresh.

2. Introduce Automated Testing

Testing is crucial for long-term reliability. Introducing:

- Unit tests for Models and Controllers to validate core functionality.
- Integration tests to ensure seamless interactions between MVC layers.
- A mock environment for simulating user interactions and edge cases.

3. Implement Advanced Logging

A robust logging mechanism would:

- Provide traceable records of system actions, useful for debugging and audits.
- Help identify system bottlenecks, especially in processing-intensive operations.
- Offer insights into user behaviour, which could inform future enhancements.

Conclusion: Wrapping It All Up

This project offered hands-on experience in designing a system within the MVC framework, addressing challenges like role-specific functionalities and input validation. It emphasised the value of modularity, scalability, collaboration, and Object-Oriented Design Principles in improving software quality. Ultimately, it highlighted how technical skills, combined with adaptability and teamwork, create impactful solutions.