



SC2002: OBJECT ORIENTED DESIGN & PROGRAMMING

HOSPITAL MANAGEMENT SYSTEM

Project Design and Functionality Report

AY24/25 Semester 1

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 honoured 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	Matriculation Number	Signature/ Date
Rachmiel Andre Teo Ren Xiang	SC2002	SCS7 - 1	U2322410E	 13/11/2024
Luke Eng Peng Kee			U2321688J	 13/11/2024
Yeo Boon Ling, Faith			U2310449G	 13/11/2024
Lee Ding Lin			U2320179C	 13/11/2024
Khoo Qian Yee			U2320573A	 13/11/2024

Table of Content

1. Design Considerations.....	3
1.1 Approach.....	3
1.2 Assumptions.....	3
1.3.1 SOLID Design Principles.....	3
a. S - Single Responsibility Principle.....	3
b. O - Open Closed Principle.....	4
c. L - Liskov Substitution Principle.....	4
d. I - Interface Segregation Principle.....	5
e. D - Dependency Inversion Principle.....	5
1.3.2 Design Patterns.....	6
a. Multi-Instance Finite State Machine (FSM).....	6
b. Model-View-Controller (MVC).....	6
1.4 Object-Oriented Programming Principles.....	7
1.4.1 Abstraction.....	7
1.4.2 Encapsulation.....	7
1.4.3 Polymorphism.....	8
1.4.4 Inheritance.....	8
2. Proposed New Features.....	8
3. UML Class Diagram.....	9
4. Test Case Demonstration.....	10
5. Reflections.....	11

1. Design Considerations

1.1 Approach

Hospital Management System (HMS) is a Java console application, built using Object-Oriented Principles, for access to hospital data by stakeholders like doctors, pharmacists and patients. The system consists of various functions: patient management, staff management, appointment scheduling, and billing. It is designed to be scalable and follow SOLID design principles.

1.2 Assumptions

- No concurrent usage (i.e. single running instance) of the application.
- No database (e.g. SQL); thus, pre-loaded data from CSV files are assumed to be sanitary (i.e. validated, non-breaking (e.g. no two appointments on the same date)).
- Doctors can create a record upon appointment confirmation (even if patients no-show).

1.3 Design Principles

1.3.1 SOLID Design Principles

SOLID design principles are five fundamental principles of object-oriented programming that make software designs more comprehensible; the following outlines how we applied them.

a. S - Single Responsibility Principle

Single Responsibility Principle (SRP) ensures that each class only has one responsibility. The class will be named appropriately to its function, improving better abstraction and readability. This also improves maintainability, as new functions can be added by instantiating another class.

We implemented SRP by splitting every package based on their functionality. For example, each class in the `appointments` package serves a sole purpose: `Appointment` stores the basic details, `AppointmentOutcomeRecord` stores the outcome, and `Prescription` handles the medication orders and its request status. Such modularization of our codebase allows for separation of concerns, allowing for ease of maintainability and scaling.

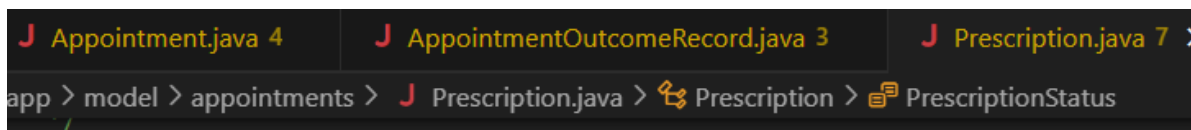


Figure 1: Every class is separated out into its packages

In `OptionGeneratorCollection.java`, every method is a static method that returns a list of `Options`. SRP is applied here because the class is solely responsible for generating option lists (e.g. `generateAdminMainMenuOptions`), allowing each option to fulfil its own functionality.

Menus are also separated based on their roles (`AdminMenuCollection`, `DoctorMenuCollection`, etc.) to keep modification to role-specific menus in one place, making the code organised and maintainable.

b. O - Open Closed Principle

The Open Closed Principle (OCP) states that software entities, including classes, modules and functions, are meant to be open for extension but closed for modification. This allows building new features atop of existing ones without modifying the logic of the existing code.

To demonstrate this, the `Menu()` abstract class outlines the expected behaviour of all menus, such as displaying a title, handling user input, and managing transitions. By creating subclasses (`InputMenu` and `OptionMenu`), the functionality of `Menu` can be extended without modifying its core implementation, which we do so using `InputMenu` and `OptionMenu`. In the future, to add a new menu type with unique functionality, a new subclass of `Menu` can be created rather than altering the `Menu` class itself. For example, `OptionMenu` adds the methods `setOptionGenerator()`, `shouldAddLogoutOptions()`, and `shouldAddMainMenuOption()` methods to add functionality specific to `Options` while still maintaining the core behaviours defined in `Menu`.

c. L - Liskov Substitution Principle

This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behaviour (i.e. subclasses behave similarly to the superclass).

To demonstrate this principle, we created a static `findUser()` method in the `UserService` class, which iterates through any `User` object (including `Staff` objects). This indicates that the `User` superclass can be freely substituted with any of its subclasses without throwing errors.

For implementing the menu user interfaces, a common `Menu` abstract class ensures that any specific type of menu (e.g., `OptionMenu`, `InputMenu`) will be returned through `getMenu()`. These specific menus inherit from the `Menu` abstraction, meaning that any `MenuState` instance can reliably return a `Menu` without requiring the client code to know the specific type of menu it will receive.

d. I - Interface Segregation Principle

Interface Segregation Principle (ISP) is defined such that a class should not be forced to implement interfaces that they do not use. To prevent such situations, smaller interfaces, created to cater to the needs of different classes, are preferred over one large general-purpose interface.

To demonstrate this, each class in the `user_credentials` package only implements the interfaces `StringValidator` and/or `IntegerValidator` only when its class requires validation of that data type.

```
src\app\model\users\user_credentials\Email.java:
16  /*
17  17: public class Email extends ValidatedData<String, String> implements StringValidator {
18
src\app\model\users\user_credentials>Password.java:
17  /*
18  18: public final class Password extends ValidatedData<String, String> implements StringValidator, IntegerValidator {
19  private final int MIN_LENGTH = 8;
src\app\model\users\user_credentials\PhoneNumber.java:
20  /*
21  21: public final class PhoneNumber extends ValidatedData<Integer, String> implements IntegerValidator, StringValidator {
22  // Constants START
src\app\model\users\user_credentials\Username.java:
20  /*
21  21: public final class Username extends ValidatedData<String, String> implements IntegerValidator, StringValidator {
22
```

Figure 2: Different classes within the package called `user_credentials` implement the `StringValidator` and `IntegerValidator`

This is also demonstrated by the `ISerializable` class, which is applied only on all base classes that need to be written into the CSV files, and only needs to implement the `serialize()` function to convert each object into a list of strings.

e. D - Dependency Inversion Principle

Dependency Inversion Principle (DIP) states that both high- and low-level modules should abstract implementation details, promoting loose coupling and, therefore, maintainability and flexibility.

DIP is demonstrated strongly through the `ISerializable` interface, which is implemented by all base (low-level) classes, which requires them to implement the `serialize()` method. On the other hand, the `DatabaseManager` (high-level) class doesn't depend on specific implementations within the base classes like `Patient` and `Doctor`. It only depends on the `ISerializable` interface, which is an abstraction. This means that any class implementing `ISerializable` can be passed into `processRow`, and `DatabaseManager` doesn't need to know the specific type.

```

public static void add(ISerializable o) {
    processRow(o, Table::addRow);
    changed = true;
}

public static void update(ISerializable o) {
    processRow(o, Table::updateRow);
    changed = true;
}

public static void delete(ISerializable o) {
    processRow(o, Table::deleteRow);
    changed = true;
}

```

```

public abstract class User implements ISerializable {
    // public abstract void displayUserMenu();
    private static int uid = 1;
    private final int userId;
    public static final String DEFAULTPASSWORD = "Easy2Type!";
}

public class Medication implements ISerializable {
    private static int uid = 1;
    private final int id;

    private final String name;
    private int stock;
    private int lowAlertLevel;
}

```

Figure 3: High-level DatabaseManager (left) follows DIP by depending on ISerializable, allowing low-level classes like Medication and User (right) to be implemented independently without direct coupling.

1.3.2 Design Patterns

a. Multi-Instance Finite State Machine (FSM)

We implemented a multi-instance FSM (inspired by [pytransitions](#)) for our frontend, with MenuService (machine) controlling the Menu that holds the MenuState (state) -collectively stored in a singleton enumeration, and points to the next menu (transition) and executes NextAction (action) upon state change. This approach provides the benefits of both singleton control and multi-instance flexibility, generating multiple context-specific Menu instances while ensuring a consistent, controlled base state.

The MenuState enumeration provides behaviour (e.g. getMenu()) that can be accessed and utilised across the application in a single, consistent way, allowing MenuService to retrieve the next menu instance with its unique callback functions. Hence, the MenuState allows menus to be derived from a stable, centralised state definition, while still being able to hold unique form data.

b. Model-View-Controller (MVC)

Our application design is based heavily on the MVC architecture: our packages are split into the folders /model, /controller, and /view. This allows us to heavily abstract away our implementation of the view (User Interface and Interactions) and the model (our base classes and business logic) from the controller (which communicates between the View and the Model). As such, our controllers are able to handle user input, retrieve data and render it on the View by calling the base models.

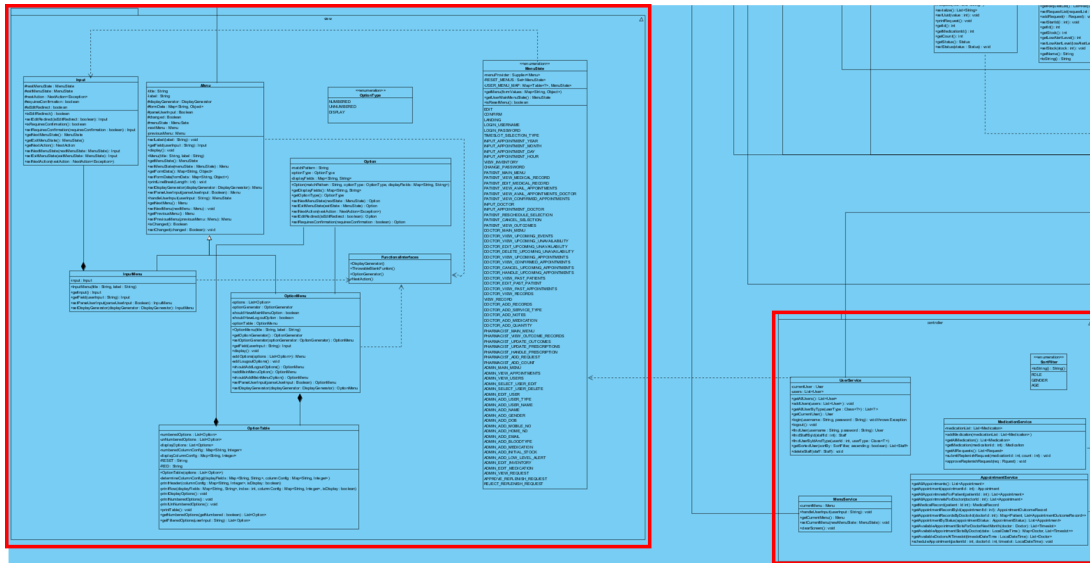


Figure 4: Controller (bottom right) is separated from View (bottom left).

1.4 Object-Oriented Programming Principles

1.4.1 Abstraction

Abstraction in object-oriented programming refers to the concept of hiding complex implementation details while exposing only the essential features of an object. It helps reduce complexity and allows the system to be more flexible, easier to maintain, and scalable.

In the context of our app, we provided all users with their own menu. This acts as an abstraction layer between the view layer and the underlying implementations. By providing a simplified interface for interacting with the system, the caller can know what output will result from the input without needing to know the inner workings of how each object is stored and manipulated. For instance, our `PatientMenuCollection`, `DoctorMenuCollection` and `PharmacistMenuCollection` all handle interactions made by the users. Meanwhile, users interacting with the menu classes do not need to understand the logic behind all of these classes in order to use them.

1.4.2 Encapsulation

Encapsulation refers to keeping data and methods in a single class while restricting access to some components. Access modifiers limit access to restricted data to prevent unwanted exposure/leaks.

Within our package structure, each package encapsulates its own data and methods. For example, all our base class constructors are not `public`, but rather `private` or `protected`. We set the constructors called by the high-level classes (Collections and Services) to be `private`, and they can only create an instance by calling `create()`. This builder method ensures the base class is immediately saved to the

“database” upon instantiation. Moreover, the constructors with params `List<String>` are protected and can only be called by the builder in the same package upon deserialisation. This also ensures that the “controller” and “view” cannot call these constructors and modify private variables directly.

Most of the fields within our base classes are declared as `private` (e.g. `uuid`, `id`, `name`, `password`). This is a key aspect of encapsulation, as it restricts direct access to these fields from outside the class. To access these fields, we provide `public` getters and setters instead. Fields, such as `id` and `name`, are also marked as `final`, which prevents modification after initialization. This helps in maintaining a consistent state for immutable properties.

1.4.3 Polymorphism

Polymorphism allows objects of different classes to take on multiple forms, or respond to the same method call in different ways. This principle becomes particularly useful for handling diverse object types in a consistent manner and promotes code reusability.

This is demonstrated within our code via interfaces, where different classes implement a general functionality that is relevant to them. This encouraged our program to be loose coupling, hence allowing additional changes without disrupting the entire code structure. For example, the interface `ISerializable` enables the base classes implementing this interface to be processed through a common method call, `serialize()`, without needing to know the specific class type. Polymorphism allows the Builder class to delegate the `serialize()` method call to different implementations depending on the actual type of base class.

1.4.4 Inheritance

Inheritance is employed across key components to promote code reuse, organisation, and scalability. For example, `User` is a base class to manage shared attributes and behaviours among different user types, such as `Staff` and `Doctor`. This setup defines fundamental properties like authentication details and user information once in `User`, which all specific user types can inherit and extend as needed.

The `Builder` classes standardised methods like `serialize()`, `buildInstance()`, `buildRow()`, `getRow()`, and `getInstance()`. Specialised builders inherit from this base, allowing for the creation of complex objects consistently while tailoring each builder to specific object types.

Similarly, in `Appointment` and `DoctorEvent`, inheritance organises shared logic, ensuring that general appointment functionality such as datetime is available while allowing for specific attributes like status and patient IDs, which cannot be found in doctor events.

Lastly, the `Menu` classes benefit from inheritance by inheriting core display and interaction methods, such as `handleUserInput`, which `InputMenu` and `OptionsMenu` inherit.

2. Proposed New Features

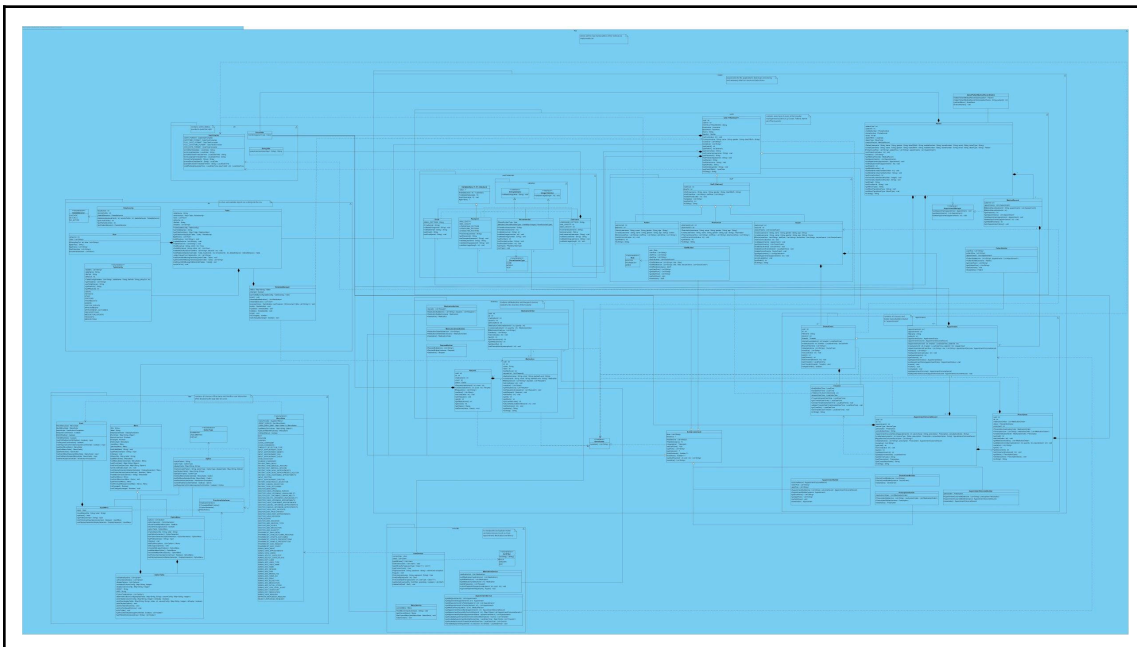
We implemented functional interfaces in Java, through the `DisplayGenerator`, `OptionGenerator`, and `NextAction`, which represent actions to be executed at various points within the interface logic.

The interfaces allow behaviours to be passed as method references, enabling flexibility in defining actions independently of the option/input that executes them. This setup specifies the next action and the next menu state that should be triggered immediately when an option is selected or input is provided.

By decoupling these actions from the specific objects, it allows for dynamic and adaptable menu navigation. Each option or input can lead to a different outcome or state transition without requiring hard-coded logic within the menu classes themselves, making the system flexible and scalable.

3. UML Class Diagram

View the full UML [here](#)



3.1 Explanation

We split up our system into a few different packages based on their main feature. Within the app package, we have the three main packages `controller`, `model` and `view`, per the MVC framework. For example, the `model` represents data related to business logic, containing all the user classes. Also, each package would have one entity and one control class, which implements the respective entity-specific interfaces to interact with their collection of objects. This design appropriately encapsulates the data and functionality of each package, while showcasing the loose coupling of classes by utilising a dependency relationship instead of an association relationship.

4. Test Case Demonstration

The following are selected test cases, expected outcomes and results ([view all test cases](#)):

1. View Medical Record

Medical Records are clearly listed when the function is called.

The patient can only view their own records, as the records of other patients are not displayed (information hiding).

[Image: A patient's information output detailing Patient Information and his/her Appointment History.]

```

Your Medical Record
Enter 'N' or 'MenuState' to return to the main menu.

Patient Details:
Patient ID: 1
Name: Alice Brown
Gender: Female
Date of Birth: Wednesday, May 14, 1989
Mobile number: +6591234567
Home number: +6566666666
Email: Alice.brown@example.com
Blood type: A+

Appointment Timeline
=====
Appointment Timeline      Patient Name  Doctor Name  Appointment Status  Service Type  Consultation Notes  Outcome Added
-----
Monday, July 22, 2024 11:00:00  Alice Brown  Sarah Lee  Completed  X-Ray  Test 1  True
Friday, November 8, 2024 14:00:00  Alice Brown  Sarah Lee  Completed  N/A  N/A  False
Saturday, November 9, 2024 12:00:00  Alice Brown  Sarah Lee  Completed  X-Ray  hihhih  True
Saturday, November 9, 2024 16:00:00  Alice Brown  Sarah Lee  Cancelled  N/A  N/A  False
Sunday, November 10, 2024 16:00:00  Alice Brown  Sarah Lee  Cancelled  N/A  N/A  False
Tuesday, November 12, 2024 16:00:00  Alice Brown  John Smith  Cancelled  N/A  N/A  False
Tuesday, November 12, 2024 16:00:00  Alice Brown  Sarah Lee  Cancelled  N/A  N/A  False
Wednesday, November 13, 2024 13:00:00  Alice Brown  Sarah Lee  Cancelled  N/A  N/A  False
Thursday, November 14, 2024 13:00:00  Alice Brown  John Smith  Pending  N/A  N/A  False
Saturday, November 16, 2024 12:00:00  Alice Brown  James Tan  Pending  N/A  N/A  False
Thursday, November 20, 2024 14:00:00  Alice Brown  John Smith  Pending  N/A  N/A  False

Select Action
=====
1. View Appointment Outcomes
2. Edit Medical Record
3. Return to main menu
4. Logout
5. Exit Application

```

10. Update Patient Medical Records

Doctors are able to add new diagnosis and treatment plans to a patient's medical record.

[Image: Medication of Ibuprofen has been added to the order from the appointment]

```

Appointment Outcome
=====
Patient Details:
Patient ID: 1
Name: Alice Brown
Gender: Female
Date of Birth: Wednesday, May 14, 1989
Mobile number: +6591234567
Home number: +6566666666
Email: Alice.brown@example.com
Blood type: A+

Appointment Details:
Friday, November 8, 2024 14:00:00
Appointment Status: Completed
Appointment Outcome Record
ID: 4
Appointment ID: 3
Service Type: X-Ray
Consultation Notes: "null"

Order ID Medication ID Prescription ID Quantity
-----
7 Paracetamol 4 10

Select Action
=====
add Add Medication
Return to main menu

Appointment Outcome
=====
Patient Details:
Patient ID: 1
Name: Alice Brown
Gender: Female
Date of Birth: Wednesday, May 14, 1989
Mobile number: +6591234567
Home number: +6566666666
Email: Alice.brown@example.com
Blood type: A+

Appointment Details:
Friday, November 8, 2024 14:00:00
Appointment Status: Completed
Appointment Outcome Record
ID: 4
Appointment ID: 3
Service Type: X-Ray
Consultation Notes: "null"

Order ID Medication ID Prescription ID Quantity
-----
7 Paracetamol 4 10
8 Ibuprofen 4 5

```

15. Record Appointment Outcome

Doctor records the outcome of a completed appointment.

Relevant updates are visible on the patient's end.

[Image: Service Type and Meds added]

```

Add Appointment Outcome
=====
Select an appointment to add outcome:

Select Timeline      Patient      Outcome Added
-----
Friday, November 8, 2024 14:00:00  Alice Brown  False
Saturday, November 9, 2024 14:00:00  Alice Brown  False
Saturday, November 9, 2024 16:00:00  Alice Brown  False
Sunday, November 10, 2024 16:00:00  Alice Brown  False
Thursday, November 12, 2024 16:00:00  Alice Brown  False
Thursday, November 14, 2024 13:00:00  John Smith  False
Sunday, November 17, 2024 11:00:00  James Tan  False

Appointment Service Type
=====
Select Service Type
-----
1 X-Ray
2 Consultation
3 Blood Test

Select Medication
=====
Please select an option:

Medication ID Name Stock Low Alert Level
-----
1 Paracetamol 120 10
2 Ibuprofen 80 10
3 Aspirin 70 10
4 Paracetamol 120 10
5 Lisinavel 80 10

Medication Quantity
=====
Enter medication quantity: (or enter 'q' to go back)
10

Appointment Outcome
=====
Patient Details:
Patient ID: 1
Name: Alice Brown
Gender: Female
Date of Birth: Wednesday, May 14, 1989
Mobile number: +6591234567
Home number: +6566666666
Email: Alice.brown@example.com
Blood type: A+

Appointment Details:
Friday, November 8, 2024 14:00:00
Appointment Status: Completed
Appointment Outcome Record
ID: 4
Appointment ID: 3
Service Type: X-Ray
Consultation Notes: "null"

Order ID Medication ID Prescription ID Quantity
-----
7 Paracetamol 4 10

Select Action
=====
add Add Medication
Return to main menu

```

19. Submit Replenishment Request

Pharmacists can submit replenishment requests for low-stock medications.

[Image: Replenishment request submitted for a low stock medication]

```
Submit Replenish Request

Please select the medication:
Select Medication ID Name Stock Low Alert Level
=====
1 1 Paracetamol 229 20
2 2 Ibuprofen 50 10
3 3 Acetaminophen 75 15
4 4 Atorvastatin 200 70
5 5 Lisinopril 50 29

Medication Quantity
=====
Please enter the quantity of medication to add: (or enter 'q' to go back)
50

Confirm Menu
=====
Please confirm your decision:
Select Action
=====
1 Confirm
2 Cancel
```

21. View Appointment Details

Administrator can view all appointments.

Patient ID, Doctor ID, status and date/time should be displayed clearly.

[Image: List of all appointments]

[illegible]

24. First-Time Login and Password Change

User is able to login with the default password and change it. Upon successful password change, the user can login with a new password.

[Image: User prompted to reset password on first login]

```

Login Password Menu
-----
Please enter your new password (or enter '\b' to go back)
|

```

27. Additional Actions Available with Filter

Doctors can easily filter into the unavailability dates menu and appointment dates menu. They can then transition to other menus seamlessly. *[Image: No filters]*

[Image: Busy filter applied allows to add busy dates]

```

Doctor Unavailability
-----
Event Type      Timestamp                                     Status
-----
Event           Sunday, November 17, 2024 14:00:00         Confirmed
Event           Monday, November 18, 2024 11:00:00         Confirmed
Event           Monday, November 18, 2024 12:00:00         Confirmed
Event           Tuesday, November 19, 2024 14:00:00         Confirmed

Select Action
-----
add      Add Busy Timestamp
reset    Reset filters
edit     Edit Busy Timestamp
delete  Delete Busy Timestamp
o        Return to main menu
l        Logout
x        Exit Application

```

[Image: Appointment filter allows them to filter into confirmed appointments (if they need to cancel) or RSVP to pending appointment requests]

[illegible]

29. Confirm Decisions

Confirmation is always requested before updating the data. This ensures that users confirm their submission once more.

```
Confirm Menu
-----
Please confirm your decision
Select Action
=====
Y          Confirm
N          Cancel
```

30. Restore previous state from wrong input

Using \q always allows the user to go back to the previous menu and restore its state. It is handled with precaution and users are unable to go back to a menu after data has been altered/logged out/on the main menu.

[Image: \q redirects user back to change the time slot]

```
select Available Doctor
Please select an option:
select Doctor
=====
1 Sarah Lee
2 John Smith
3 John Lee
4 James Tan
5 Jared Toh
6 Joel Lim
7 Grace Chen
select Action
=====
8 Return to main menu
(or enter '\q' to go back)
```

5. Reflections

The development of the application was particularly challenging due to the amount of refinement that was required in designing many parts of the app to ensure it was reliable. This was especially seen in our heavily abstracted frontend display and backend data handling, which allowed the high-level Controller layer to communicate between the View and Model without having to understand their implementations.

For the front end, we needed it to be robust and intuitive which allowed for seamless integration, and decided not to use the typical if-else structures within the controller to redirect users around the app. Instead, we placed importance on OOP concepts and implemented a finite state machine (FSM), where every “menu” was an object and called the same methods like `display()`, `handleUserInput()`, `nextAction()`, `nextMenu()` and `exitMenu()`. Such a design allowed us to chain multiple menu objects together, allowing us to move to the next and previous menu without losing context. Such a high-level, abstract design was much more complicated compared to simple conditional statements, and we easily spent over 3 full days just refining the base functions, as such an abstraction led to many bugs initially. While it was an arduous experience, we managed to push through with countless hours of coding and fulfilled our decision to implement our front end as a FSM.

Moreover, the implementation of ensuring data persistence was a challenging task due to the fragility of serialising each base class object. We utilised an extremely careful process where every base object was serialised into an `ArrayList<String>`, with attributes in the correct order. We then mapped each serialisation process with its deserialization process, through specialised `Builder` classes to recreate the same object with the same class. This process would have been much simpler with Object Relational Mapping (ORM) in a standard application setup and was extremely difficult to set up due to the sheer size of tables and understanding the structure of how data should be stored. Apart from the standard OOP concepts, having to design such a reliable data storage that resembled a “database”

challenged us to also think about Database design, which would undoubtedly be very useful in our Database module next semester.

Yet, even with our success in the heavy abstraction of frontend and backend, we feel that we could have approached these problems from a different angle to be more efficient in future coding projects. For example, we decided not to rely on the UML when developing the frontend and backend, due to our eagerness to set them up quickly. In hindsight, we believe that this was a mistake as we could have given more consideration to enforcing clean OOP concepts and seeking out potential bugs. As such, we believe we could have avoided countless refinements and refactorisations if we had considered these aspects earlier in our UML design.

As the hospital expands in its scale, HMS application may eventually be used by other healthcare professionals, including nurses, physiotherapists and others. With the loosely coupled architecture of our HMS application, future software engineers working on our code will find it easy to implement these new users into the system. Data security issues must also be addressed as healthcare data is sensitive, and there should be best practices put in place to minimise the likelihood of a data leak.