GitHub Repository: <https://github.com/firepiratex/SC2002>

**1. Design Considerations and Design Approach Taken**

1. Design Considerations

We will be designing this application by using "Handler" classes as a “Controller” to separate method definitions from entity classes, which contains the data information attributes. The entity classes rely on these Handler classes to manipulate the data effectively. Additionally, we use Interface classes to manage the output terminal, ensuring that the application’s output is displayed clearly to the user. As we are not allowed to display on a graphical user interface (GUI), we will implement a pseudo Model-View-ViewModel (MVVM) architecture, where the program's output is presented in the terminal rather than through a GUI.

We chose to use csv files as a database as it is the only way we know to store and extract the data that is given. Our program reads the data from the user information from the csv files to create the necessary objects needed for the program and updates the csv files to store user created data in real-time.

We also attempted to make our code robust by implementing the SOLID design principles. This ensures that our code has full error checking and is able to adopt a user-friendly approach by having a simple and easy-to-read user interface flow and immediate prompts for user inputs whenever erroneous input is received.

1. Design Approach Taken

*Login Process*

Upon starting the HMS application, you will be greeted with the ASCII code of the word “HMS”. You will then be prompted to enter a UserID and the plain text password. The UserID is first checked against all the existing UserIDs in the Patient\_List and Staff\_List txt files.

When entering a password for a user, the plain text password will be hashed using SHA-256 which is a cryptographic hash function that produces a fixed-size output. This hashed password is compared with the stored passwords in the Patient\_Account or Staff\_List text file which ensures that passwords are safely secured in the text file. This text file is only used for storing the account information of the users which adheres to the Single Responsibility Principle (SRP).

*Data Storage via CSV*

To load and store data for the various roles, we will be utilising the “data” folder which contains all the csv files for the application including user and appointment details. The CSVHandler class acts as a boundary class as it can provide connections to external sources by reading and modifying CSV files from the data folder. When reading, it returns string values that correspond to the lines of the CSV file. It also allows to add, remove or modify individual lines in a selected CSV file. Once modified, the CSV file is updated in real-time to the existing CSV file.

In each CSV file, each line corresponds to the data of an individual entity, such as user or appointment. Each line is able to store all required attributes about a particular entity, such as strings, integers, and even arrays are stored by using square brackets as separators since it is crucial that the CSV file remains comma-separated about their columns.

To convert the data into entity classes, control classes also known as database managers, inherit from the CSVHandler class to transform each line returned by its functions into corresponding entity classes, and vice versa. For instance, the AppointmentManagement class reads the Appointment\_Log.csv file, allowing it to return an array of Appointment classes, each containing information about individual appointments. The AppointmentHandler class, distinct from AppointmentManagement, aids in processing data from the AppointmentManagement and offers specialized methods for appointment operations. This design helps prevent class bloat and adheres to the Single Responsibility Principle.

The use of these control classes facilitates dependency injection for classes reliant on entity classes. For example, a User class might need to retrieve a list of available appointments for selection. Instead of directly creating appointment classes within the user-related functions, the list of appointments provided by AppointmentManagement can be inserted into the user methods. This approach streamlines the code, minimizes future modifications, and simplifies the workflow for developers working on the User class. This demonstrates the Open/Closed Principle (OCP) by allowing for future enhancements without modifying existing code.

*User Roles: Patient, Doctor, Pharmacist and Administrator*

The Patient, Doctor, Pharmacist and Administrator classes inherit from the User class and they contain methods only specific to their respective roles. This exemplifies the Single Responsibility Principle. Since the User class has the basic attributes of each role, each role is able to inherit from User thus they can access User methods.

We used method overriding as a form of polymorphism by overriding the displayMenu method across various subclasses that inherit from the main User class. Each subclass, such as Patient, Doctor, and Administrator, is able to provide its own specific implementation of the displayMenu method which allows for tailored menu options relevant to each user role. This polymorphism enables the application to call the displayMenu method on a User reference and dynamically invoke the appropriate version of the method based on the actual object type at runtime. For example, when a Doctor instance calls displayMenu, it will show options pertinent to doctors, while a Patient instance will display options suitable for patients.

We also used method overloading through the use of the writeCSV and readCSV methods. It allows flexible handling of various CSV files. By defining multiple versions of these methods, we can tailor the input parameters to accommodate different data types and structures based on the specific CSV files being processed. For instance, the writeCSV method might be overloaded to accept parameters for writing patient data to the Patient\_List.csv file as well as appointment data to the Appointment\_Log.csv file, each with their unique data formats. Similarly, the readCSV method can be overloaded to read from different files such as Staff\_List.csv or Medicine\_List.csv, enabling the application to efficiently handle the diverse requirements of the system.

We will utilise the Singleton pattern which restricts the instantiation of a class to a single instance and provides a global point of access to that instance. This ensures that a class is instantiated only once throughout the application. By adding an instance attribute to the class intended to be a singleton, we can create a getInstance() method. This method allows us to retrieve the single instance of the class. If it is the first time accessing it, the method will initialize the class; otherwise, it will return the already created instance. Consequently, in the future, when we need to access the Patient and Staff, we do not need to reinitialize them, thus maintaining consistency and preventing errors. This is directly related to the Single Responsibility Principle.

*Creating Entities*

To be able to add entities into the CSV database, such as appointments, medicine and inventory, a method is invoked within their respective management packages to create a new entity object. This method converts the entity into a CSV line format and appends that line to the corresponding CSV file, enabling the storage and retrieval of entire entities.

Once an entity is created in the database, it contains a unique identifier, such as AppointmentID or MedicineID, so that we are able to reference them in the future. They also contain the UserID of the person who made the entity, such as the staff member who is managing appointments. It is saved in the Appointment\_Log CSV for appointments. This helps in identifying and tracking the entities created by different users.

It is worth noting that all data managers inherit from the CSVHandler class, since they require the reading and writing of CSV files. This design adheres to the Open-Closed Principle, because if we wanted to create a new database in the future, we can just extend from the CSVHandler again to access database editing capabilities, without the need to modify the inner workings of the CSVHandler.

*Appointment Details*

*Assumptions Made*

* There will not be multiple users logged in at the same time using the application as it may lead to data inconsistency from the delay in changes within the database.
* The data should always be available and is never corrupted to ensure readable data.
* Database is securely protected in a safe location as it contains sensitive information about users in plaintext for the CSV files.
* The doctors will work 9am to 6pm every day.

**2. Detailed UML Class Diagram**

**3. Testing**

* Test cases and results:

1. *Login*
2. Cannot login:

|  |  |
| --- | --- |
| Invalid user | Valid ID but wrong password |
|  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Patient | Doctor | Pharmacists | Administrator |
|  |  |  |  |

1. Users are asked to change their password when they login for the first time:

|  |  |  |  |
| --- | --- | --- | --- |
| Patient | Doctor | Pharmacists | Administrator |
|  |  |  |  |

1. After successful login, a different main page is displayed for different roles.

|  |  |  |  |
| --- | --- | --- | --- |
| Patient | Doctor | Pharmacists | Administrator |
|  |  |  |  |

1. After login, user can change password

|  |  |  |  |
| --- | --- | --- | --- |
| Patient | Doctor | Pharmacists | Administrator |
|  |  |  |  |

1. User cannot change a password to the default one (password)/old password. (Input error checking)
2. Patient

|  |  |
| --- | --- |
| View Medical Record | Update personal information |
|  |  |

1. View medical record & update personal information

|  |  |  |
| --- | --- | --- |
| Schedule | reschedule | Scheduled appointment |
|  |  |  |

1. Schedule, reschedule, view scheduled appointment
2. View available appointment, cancel, view past appointment outcome records

|  |  |  |
| --- | --- | --- |
| View available appointment | Cancel | View Past appointment records |
|  |  |  |

1. Doctor

|  |  |
| --- | --- |
| View | Update |
|  |  |

1. View & update patient medical records

|  |  |
| --- | --- |
| View | Set |
|  |  |

1. View personal schedule & set availability for appointments

|  |  |
| --- | --- |
| Accept/Decline | View |
|  |  |

1. Accept/Decline appointment requests, View upcoming appointment

|  |
| --- |
| Record |
|  |

1. Record Appointment Outcome
2. Administrator

|  |  |
| --- | --- |
| View | Manage |
|  |  |

1. View and manage hospital staff

|  |
| --- |
| View |
|  |

1. View Appointment details

|  |  |
| --- | --- |
| View | Manage |
|  |  |

1. View and manage medicine inventory

|  |
| --- |
| Approval of replenishment requests |
|  |

1. Approve Replenishment requests

**4. Reflection**

a. Difficulties Faced

When we were first going through the project, we realised that the requirement did not let us use any database application which restricts us in how we store our data and information. After reviewing the provided files, we realized we could leverage Excel/CSV files as a makeshift database to store our information.

As we worked to implement and optimize our code, we struggled to apply the concepts we learned in lectures, particularly around object-oriented design and the SOLID principles. The UML diagram, which was derived from our code, required multiple iterations as we sought to align our implementation with these principles.

Through this learning process, we recognised that our initial code structure was difficult to read and prone to breaking whenever we attempted to add or implement new features. This experience has highlighted the importance of the concepts that were taught to us in the lectures which ultimately helps us to become better programmers. We learned to appreciate the value of clean, scalable code and the need to adhere to design principles to create robust software.

b. Knowledge learnt from this course

From the lectures, we had learnt about the object-oriented (OO) concepts alongside the SOLID design principles. We tried to ensure that any necessary changes to our project could be implemented smoothly with minimal impact. We also learnt to manage the implementation of a relatively large project given a short timeframe by learning the importance of divide and conquer techniques so that we are able to do the tasks efficiently. Additionally, we took the initiative to self-learn and utilize libraries and built-in functions, such as java.time and CSV read/write methods using java.io which enhances our application’s functionality and robustness.

c. Area of improvements for the course

There were project specifications and descriptions which were vague and unclear. This made us confused. We had to come up with assumptions as information given was vague so some parts of the application were up to our interpretation. At first, the given excel files were in xlsm format which was incompatible with the java.io library. We had to email the professor to ensure that we did not go against any restrictions that were preplaced for the project. The suggested sample test cases were also not aligned to the features required of the application. The report requirements were also misaligned with the appendix. An example screenshot is attached in Figure XX. The figure on the left does not mention the “Testing” section to be included while the figure on the right mention about adding test cases.



