SC2002: OBJECT ORIENTED DESIGN & PROGRAMMING

HOSPITAL MANAGEMENT SYSTEM

Project Design and Functionality Report

AY24/25 Semester 1

<u>Declaration of Original Work for SC2002/CE2002/CZ2002 Assignment</u>

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
Goh Bo Jun, Issac(Wu Bojun)	SC2002	SCSG - 1	U2321547G	*[signature]* 20/11/2024

Chew Wei Hao, Kovan			U2322227A	20/11/2024
Chananam Kulpatch			U2320130D	20/11/2024
Shreyes Sudarshan Krishna			U2321087C	20/11/2024
Tan Yan Kai Noel			U2322141K	20/11/2024

Table of Content

1. Design Considerations

1.1 Approach

Hospital Management System (HMS) allows users of different roles: DOCTOR, PHARMACIST, PATIENT, ADMINISTRATOR to log in to the system. Designed with the SOLID design principles. Two Additional features that we believe stood out was the payment system that allows Patients to pay the Outstanding Medical bills to pay and a Generated GUI for the Calendar in real time based on month and date.

1.2 Assumptions

- No concurrent usage (i.e. single running instance) of the application.
- Used CSV as the file extension to store data
- Payment method available: Credit Card
- Pharmacists will help check for the replenishment of stocks and will be notified if stocks went down a certain range.
- Scheduling of the Time slot for the appointments are made within the time frame of the operating hours and organised in every 1 hour

1.3 Design Principles

a. S - Single Responsibility Principle

(SRP) ensures that each class has only one function. If the current class needs any particular function, It can pass the responsibility to the class with that function. Maintainability is made easier and modification for the source code is avoided.

In our app, SRP is implemented by segregating the packages into segments, and localising them according to their functions that it needs.

We have a GenerateIdHelper class with the static key functions of generating ID with UUID for the entities and objects. Single responsibility is used here as the sole purpose of this class is to generate ID which is crucial for the existence of our data. Now, operations like scheduling an appointment can easily rebound the responsibility of generating a new ID for the Appointment to this class.

We have a Validators folder which holds the ContactInfoValidator, CreditCardValidator and PasswordValidator, all which have one sole responsibility which is to ensure the validity of the respective input fields given by the user.

b. O - Open Closed Principle

We definitely used the (OC) principle in our project as our objects implement the interface that requires logical abstract methods to be implemented by the respective subclasses.

For example, the DataManager interface that has add(), update(), delete(), retrieve(), retrieveAllI(), writeAllI() and getList() abstract methods that are definitely needed for the reliability of the data saved to and retrieved from the CSV.

Now the respective Data Managers for each object that exist in this app like the Appointment, Replenishment, Staff, Patients, Medical bills and Medication Request can implement the interface and declare the codes accordingly. Furthermore, we use the Generic concept to add an additional layer of abstraction on the types available to reduce error prone codes. Types available are definitely bounded on the entities like User, Appointment, Medication etc.

In this case, We do not need to modify additional codes or check the instance or reference type of the Data Manager in order to obtain the relevant methods every time there is a newly created DataManager for an arbitrary object since all of the DataManagers have already implemented the interface DataManager and abstraction is achieved. Maintainability of code is achieved.

c. L - Liskov Substitution Principle

(LS) is a principle that we definitely use throughout our app where our focus is to ensure the derived classes inherit/implement base classes that are "compatible" to it.
There will not be an instance where the derived classes had to edit the methods inherited from base classes, which also meant that derived classes are just partially the base classes.

d. I - Interface Segregation Principle

(IS) principle was also used to design our app whereby we only allow classes to implement interfaces that are "relevant" to it logically. One example is the PasswordInterface which the LoginService implemented. LoginService which handles the business logics for the authorization and authentication

of users implemented PasswordInterface for the checkFirstLogin, validate and changePassword functions which is also an intersection between the LoginService and PasswordInterface where the latter is the subset in terms of functions.

e. D - Dependency Inversion Principle

Dependency Inversion Principle (DIP) states that the high level or (the child) and the low level or (the parent) should depend on abstraction and the parent "closest to the method" will implement an interface which can be injected as an Dependency in the "child" constructor or function.

We adopt this concept on our app with a great example shown in all respective data managers which inherited the DataManager interface for abstraction and services/controllers can use the dependency for dynamic binding to only call the methods required and because this Singleton DataManagers are initialised in the beginning, we can easily use it in the services' constructors.

Due to the dynamic binding, Modularization is achieved in our app because our high level classes do not need to get redundant methods from the low level classes. Oftentimes, (DIP) and (OC) principles are inseparable for dynamic binding to happen.

1.3.2 Design Patterns

a. Singleton

We implemented a singleton design pattern (inspired by) for our data management. This is to ensure that there is only one centralised instance managing any data throughout the application. This pattern is ideal for managing shared resources such as database connections or file-based data, as it prevents the creation of redundant instances and ensures data consistency. By implementing the Singleton, we provide a single point of access for all operations such as staff records, which simplifies maintenance and avoids issues like duplicate data or race conditions.

In our implementation, the pattern is enforced using a private static instance variable and a public getInstance() method, which initialises the instance only once and reuses it thereafter. This design reduces memory usage and aligns with the principle of controlled resource access, making it well-suited for managing staff data across the various components of the hospital management system.

b. Model-View-Controller (MVC)

The **Model-View-Controller (MVC)** design pattern was adopted for structuring the application to promote separation of concerns and maintainability. The **Model** is represented by the entities package,

which encapsulates the core data structures, such as staff, patients, and their associated details. This ensures that the data layer is independent of the user interface and service logic.

The **View** is implemented through the various menu classes, which serve as the user interface, presenting options to users and collecting input. This layer focuses solely on interaction without directly handling business logic.

The **Controller** layer is handled by the service classes, which coordinate the interactions between the Model and the View. These services implement specific functionalities like managing appointments or retrieving medical records.

By using MVC, the application achieves a modular structure where changes to one layer, such as modifying the menu layout, do not impact the underlying data or service logic, ensuring scalability and ease of maintenance.
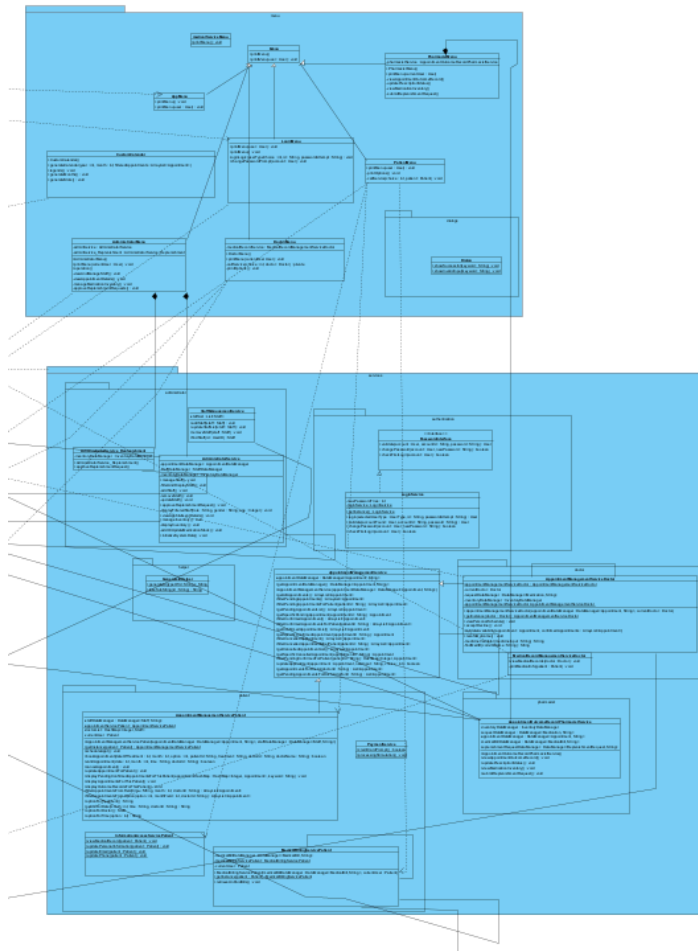

Figure 4: Controller (bottom) is separated from View (Top).

1.4 Object-Oriented Programming Principles

1.4.1 Abstraction

Abstraction will mean only showing the relevant functions to the sub classes or high level modules. Complexity of the app is minimised but no doubt generalisation or realisation between classes will not be eradicated since it is OOP.

However the scalability of the app is achieved and the flexibility for new programmers to modify is made easier as confusions during modification of codes are reduced.

Our app's folders with the classes are also organised in the localised and logical manner. Services for the different role will be placed accordingly so that abstraction is achieved.

Our menus that implement the abstract class "Menu" is one great example since modification will only be to the respective menus for the abstract method, PrintMenu().

This function will do the similar thing of printing out the options in the menu but for different roles.

1.4.2 Encapsulation

Encapsulation should always be used in the app to avoid unknown modifications made by other classes which can be detrimental for debugging.

It is because of loose coupling, the members in each class are only accessible to the respective class and restricted to the other classes. There will not be a direct access/modification for other classes.

Though it is not strongly recommended to place the PRIVATE access modifier to functions as we still need GETTER/SETTERS that allow the object instance of the respective class to still be able to access the members.

One great example which works well with the Singleton approach is our Data Managers where we ensure each only has 1 instance throughout the app.

Private access modifiers are for these instances so that no other modules can modify the instances of these singletons (e.g create new).

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 DataManager enables the data management classes implementing this interface to have similar functionalities through common method calls, without needing to know the specific class type. Polymorphism allows the data management classes to delegate the writeAll() method call to different implementations depending on the actual type of base class at the end of the program.

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 user information once in User, which all specific user types can inherit and extend as needed.

The DataManager classes standardised methods like writeAll(), update(), add(). Specialised managers inherit from this base, allowing for the creation of complex objects consistently while tailoring each manager to specific object types.

Lastly, the Menu classes benefit from inheritance by inheriting core display and interaction methods, such as printMenu, and utilising the User object to perform specific functionality for the specific menu.

2. Proposed New Features

We have 2 new unique features added to the HMS app that are crucial to operate the daily operations for a hospital.

1) Real-time GUI of a Calendar for that month during appointment scheduling. Legends: Taken(T) or date passed (X) will be marked on the dates in the calendar.

This helps the patients to easily be able to visually and clearly see the dates that they cannot pick which save time for them.

Generating the Calendar object with the relevant legends and slots, are made with the Singleton approach to prevent memory leak and save memory resources.

2) Medical bills / Transactions. We want patients to be able to be notified of their outstanding bills and be able to pay them electronically.

    Pharmacists will be the one to generate and issue the electronic medical receipts to the patients that COMPLETED their appointment on that day.

    App will check if the outstanding bills are UNPAID for more than 30 days and if true, an additional penalty fee of $10 will accrued under the patient

    Payment stimulation is done for Credit Card only such that the Payment Gateway will verify users' CC details are in the correct format. These business logics are handled by the PaymentService class.

Every entity in our app that is of data to CSV will definitely have their own ID. We used the primary key - foreign key approach from SQL relationship ideology to establish relationships between data logically. Association, "Has-A" relationship is used here in this case.

3. UML Class Diagram

View the full UML here
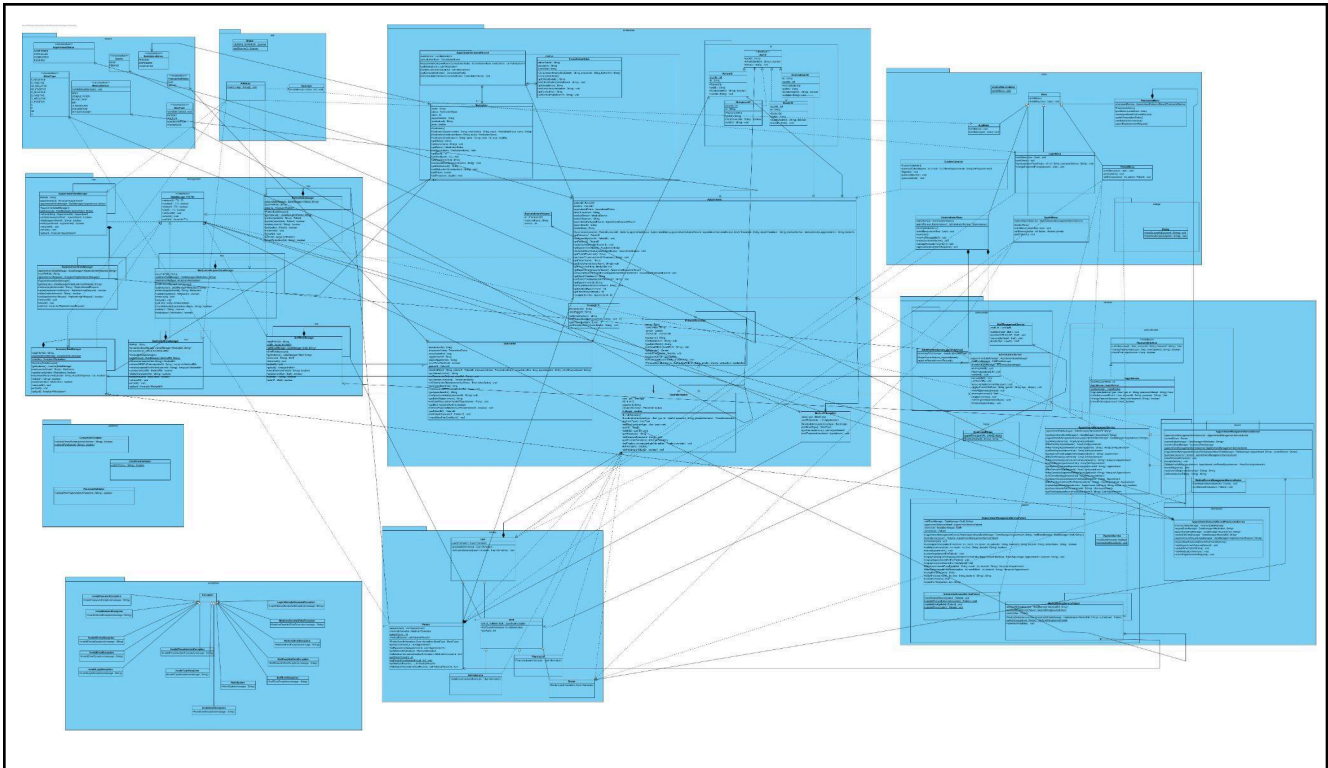
3.1 Explanation

Since packages are diversified and files are grouped in logical order, our UML diagram are much more neat and tidy.

We followed the MVC framework though naming convention may slightly differ for controllers as we named it as services.

Stronger association, composition are used logically for the entities/models like Appointment (the whole) and the ConsultationNotes (the part). The lifecycle of this ConsultationNotes depends on the existence of the Appointment. This is very crucial as we are utilising the memory resources appropriately by "destructuring" objects appropriately.

Every Data Manager implements the interface which proves to have a realisation relationship. So the functionalities of these data managers have similar purposes and dynamic binding can be achieved.

Each service requires at least one respective Data Manager for data reliability and maintainability. Dependency injection is involved such that each service requires it in the constructor. These initialised data managers are stored to the members of the service but because Data Managers are singletons, general association is used.
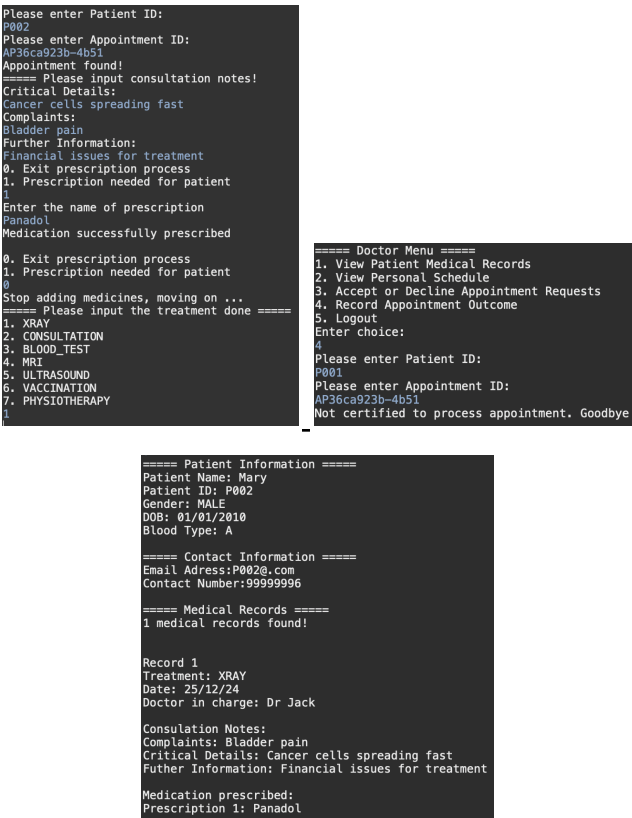
4. Test Case Demonstration

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

Test Case 1: View Medical Record (Patient)

[Image: Output of Patient's Information and his/her Appointment History.]

As a **Patient**, they can view their personal information and past medical records when the function is called. The patient is not allowed to see other patients' records (Encapsulation).

Test Case 15: Record Appointment Outcome (Doctor)

[Image: Recording appointment outcome by entering consultation notes, treatment provided and prescription request]

Doctor can only record appointment outcome for a patient that is accepted by the **Doctor** from the **Patient's** appointment request. Prescription will refer to medicine inventory to ensure the medication prescribed is in stock or exists. If found, medication request will be sent to **Pharmacist** and appointment outcome is updated from CONFIRMED to COMPLETED. **Doctor**/**Patient** can then view this medical record outcome by selecting the view medical record option within their respective menus.

Test Case 17: Update Prescription Status (Pharmacist)

medication_request

Appointment_ID	Medication_ID	status	Medication_name
AP964507f3-6580	Med_002	DISPENSED	Panadol
AP37e57f6d-0e98	Med_002	DISPENSED	Panadol
APfecf0684-665e	Med_002	PENDING	Panadol
AP36ca923b-4b51	Med_002	PENDING	Panadol

[Image: Before approving medication request

```
Pharmacist Menu:
1.View Appointment Outcome Record
2.Update Prescription Status
3.View Medication Inventory
4.Submit Replenishment Request
5.Logout
Enter choice:
2
Updating Prescription Status
Enter the appointment Id to update the status of prescription:
AP36ca923b-4b51
Prescription for medication ID: Med_002 has been updated to the state DISPENSED.
Receipt Transaction ref: TRce1aa8ef-afa3 generated
```
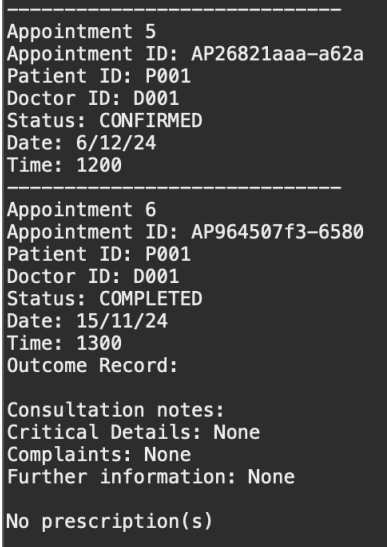
[Image: Approving medication request

medication_request

Appointment_ID	Medication_ID	status	Medication_name
AP964507f3-6580	Med_002	DISPENSED	Panadol
AP37e57f6d-0e98	Med_002	DISPENSED	Panadol
APfecf0684-665e	Med_002	PENDING	Panadol
AP36ca923b-4b51	Med_002	DISPENSED	Panadol

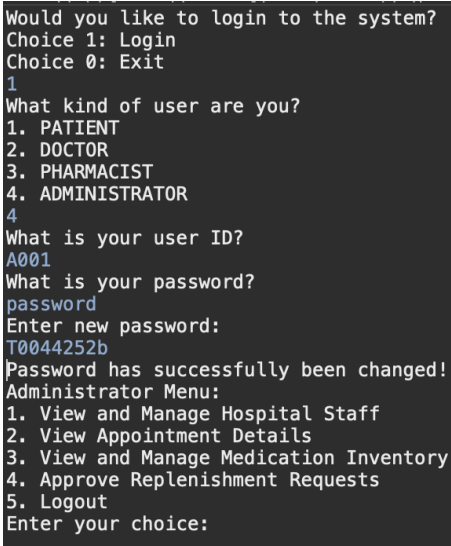**[Image: Before approving medication request**

W.r.t to appointment ID AP36ca923b-4b51 that was previously recorded by the **Doctor**, the pending medication request will be overlooked by the **Pharmacist** who is in charge of dispensing the medication. The status of the medication will then switch from PENDING to DISPENSED and a transaction receipt will be generated to process payment by the **Patient**

Test Case 21: View Appointment Details(Administrator)

``` ------------------------------ Appointment 5 Appointment ID: AP26821aaa-a62a Patient ID: P001 Doctor ID: D001 Status: CONFIRMED Date: 6/12/24 Time: 1200 ------------------------------ Appointment 6 Appointment ID: AP964507f3-6580 Patient ID: P001 Doctor ID: D001 Status: COMPLETED Date: 15/11/24 Time: 1300 Outcome Record: Consultation notes: Critical Details: None Complaints: None Further information: None No prescription(s) ``` **[Image: Display all appointments]**	**Administrator** can view all appointments regardless of their status. For all appointments, it will display details Patient ID, Doctor ID, status, date and time. For appointments that have a COMPLETED status, it will print extra information of the appointment outcome record such as the consultation notes and prescription

Test Case 24: First-Time Login and Password Change)

``` Would you like to login to the system? Choice 1: Login Choice 0: Exit 1 What kind of user are you? 1. PATIENT 2. DOCTOR 3. PHARMACIST 4. ADMINISTRATOR 4 What is your user ID? A001 What is your password? password Enter new password: T0044252b Password has successfully been changed! Administrator Menu: 1. View and Manage Hospital Staff 2. View Appointment Details 3. View and Manage Medication Inventory 4. Approve Replenishment Requests 5. Logout Enter your choice: ``` **[Image: Password change prompt for first-time login]**	All users will be required to change their password to a specific format.

5. Reflections

Developing and creating the Hospital Management System was particularly tedious as it required a lot of planning of the designs to ensure it was reliable and fulfil the requirements as stated. This was especially seen in our heavily abstracted frontend display and backend data handling, which allowed the services(Controller) to communicate between the View(Menu) and Model(Entities) without having to understand their implementations..

For the front end, we aimed to create a robust and intuitive design that allowed for seamless user interactions while maintaining flexibility and scalability. Instead of relying on typical **if-else structures** within the controller to manage navigation and logic, we adhered to Object-Oriented Programming (OOP) principles by organising our front end around a **menu-service interaction model**. Each "menu" was implemented as a class, encapsulating its display logic, user input handling, and the ability to invoke the corresponding service layer methods.

The design focused on creating a clear separation of concerns, where each menu acted as a mediator between the user and the underlying business logic. For example, the **Patient Menu** class provided options such as viewing medical records, updating personal information, scheduling appointments, and more. Each menu item invoked specific methods in service classes like InformationAccessService, AppointmentService which contained the business logic for those actions.

Ensuring data persistence was challenging, as we implemented a custom data management system using the Singleton pattern instead of relying on Object Relational Mapping (ORM) tools. This ensured centralised control, with objects stored in structured files converted into their respective types using helper methods. We designed the system to handle edge cases like corrupted data and ensured thread safety within the Singleton. While complex, this approach provided a lightweight, file-based "database" for the application. This experience not only deepened our understanding of design patterns but also improved our skills in data handling and modular software architecture.

Despite our success in heavily abstracting both the frontend and backend, we recognize areas for improvement in future projects. Specifically, we chose to forgo creating UML diagrams during development, prioritising speed over planning. In hindsight, this decision was a mistake, as it led to missed opportunities for enforcing clean OOP principles and identifying potential bugs early. This oversight resulted in numerous refinements and refactorings that could have been avoided with better upfront design.

As the hospital grows, the HMS application could expand to include other professionals, such as nurses and physiotherapists. Its loosely coupled architecture ensures future developers can easily integrate these users into the system. Additionally, robust data security measures will be critical to protect sensitive healthcare information and prevent data breaches.