**SC2002:**

***A logo with a lion on it

Description automatically generated*Object-Oriented Design Analysis & Programming**

Github: <https://github.com/Pytode2000/SC2002_SCED_Group_4>

**Declaration of Original Work for SC2002 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** | **Signature / Date** |
| LIEW WEI JIE | SC2002 | SCED | / 15-11-2024 |
| LIM ZU LIANG | SC2002 | SCED | / 15-11-2024 |
| JACOB ONG JIA CHUN | SC2002 | SCED | / 15-11-2024 |
| TAN YI XIANG, DYLAN | SC2002 | SCED | / 15-11-2024 |

**Table of Contents**

[Chapter 1. Introduction 2](#_Toc182597780)

[Chapter 2. Design Considerations 4](#_Toc182597781)

[Chapter 2.1. Design Goals 4](#_Toc182597782)

[Chapter 2.2. Design Approach 4](#_Toc182597783)

[Chapter 2.3. Applied Design Principles 6](#_Toc182597784)

[Chapter 2.4. Design Highlights 7](#_Toc182597785)

[Chapter 3. Additional Features & Enhancements 8](#_Toc182597786)

[Chapter 4. Functional Tests 9](#_Toc182597787)

[Chapter 4.1. Patient Actions 9](#_Toc182597788)

[Chapter 4.2. Doctor Actions 10](#_Toc182597789)

[Chapter 4.3. Pharmacist Actions 11](#_Toc182597790)

[Chapter 4.4. Administrator Actions 12](#_Toc182597791)

[Chapter 4.5. Login System & Password Management 12](#_Toc182597792)

[Chapter 5: Reflecting on our Voyage 13](#_Toc182597793)

[5.1. Polymorphism of Challenges 13](#_Toc182597794)

[5.2. Debugging the Group Dynamics 13](#_Toc182597795)

[5.3. Committing Knowledge Gained to Brain Repository 13](#_Toc182597796)

## **Chapter 1. Introduction**

The Hospital Management System (HMS) is designed to streamline the administration of medical institutions by leveraging Object-Oriented Programming (OOP) principles and robust architectural patterns. This system aims to provide a centralized solution for managing patient records, staff roles, appointments, and billing processes.

By focusing on usability, extensibility, and security, the HMS caters to diverse user roles, including patients, doctors, pharmacists, and administrators. With its modular design and role-based access control, the HMS ensures seamless navigation and robust data protection, while supporting future enhancements with minimal disruption. This report outlines the key design considerations, applied principles, and implementation strategies.

**\*** Due to the complexity and size of the HMS program's UML diagram, including it directly in this document would render it unreadable. As such, we have extracted the UML diagram and will submit it separately alongside this report for clarity and ease of review.

## **Chapter 2. Design Considerations**

### **Chapter 2.1. Design Goals**

The HMS design is guided by the following primary objectives:

* **Usability:** Ensure a simple and intuitive CLI that offers role-specific functionalities, enabling effortless navigation for all user types.
* **Extensibility:** Build a system that accommodates future requirements or enhancements, such as adding new roles or features, with minimal modifications to existing code.
* **Modularity:** Adopt a modular architecture where each component has a distinct responsibility and operates independently, simplifying future feature integration.
* **Security:** Implement role-based access control to protect sensitive data, ensuring that only authorized users can access specific functionalities.

### **Chapter 2.2. Design Approach**

#### Chapter 2.2.1. Boundary-Control-Entity Pattern

Our HMS is designed with a focus on high cohesion and loose coupling, adhering to Object-Oriented Programming (OOP) best practices. It employs the Boundary-Control-Entity (BCE) pattern, a structured approach akin to the Model-View-Controller (MVC) model. This pattern enforces a clear separation between the system's presentation layer, business logic, and data layer, ensuring maintainability and scalability.

A blue rectangular diagram with black circles

Description automatically generated with medium confidence A screen shot of a computer program

Description automatically generated

Figure 1

* **Boundary Classes:** Act as the user-facing layer, responsible for capturing input and displaying information. By limiting all user interactions to boundary classes, the system enforces a strict separation between the interface and internal logic.
* **Controller Classes:** Serve as intermediaries, processing input from boundary classes and coordinating data flow to and from entity classes. This encapsulation of control logic ensures modularity and reduces dependencies, making the system more abstract and maintainable.
* **Entity Classes:** Represent the application's core data structures, such as patient information, staff records, and billing details. These classes are designed to function independently, ensuring reusability across different components of the HMS.

Conclusion: This BCE structure ensures the HMS remains flexible, maintainable, and scalable, allowing for seamless adaptation to evolving requirements.

#### Chapter 2.2.2. Application of Oriented-Object Concepts

The HMS leverages key Object-Oriented Programming (OOP) principles to ensure the system is structured, adaptable, and intuitive.

* **Abstraction:** Simplifies complex internal logic by exposing only essential functionalities through user-friendly methods and interfaces, reducing user complexity and enhancing usability.
* **Encapsulation:** Protects the internal state of objects by bundling data and behaviour within their respective classes (e.g., *Patient*, *Appointment*). Access and modifications are restricted to well-defined interfaces, ensuring modularity and security.
* **Inheritance:** Minimizes redundancy by creating a hierarchy where subclasses inherit common attributes and methods from parent classes. For example, the User class serves as the base for roles like *Patient*, *Doctor*, *Pharmacist*, and *Administrator*, enabling shared properties such as *userId* and *userRole*, while allowing role-specific customizations.
* **Polymorphism:** Increases flexibility by enabling different classes to respond uniquely to the same method calls. This is particularly useful for handling user inputs, allowing roles such as *Doctor* and *Patient* to invoke the same method but execute distinct behaviours based on their role-specific implementations.

### **Chapter 2.3. Applied Design Principles**

#### Chapter 2.3.1. Application of SOLID Principles

Our team embraced the SOLID principles as a guiding framework to ensure our HMS adheres to software development best practices:

* **Single Responsibility Principle (SRP):** Each class is designed to handle a single, well-defined responsibility, preventing overlapping functions and aligning with the BCE architecture. For example, the *AccountController* manages user authentication and account-related tasks, while *Patient*, *Doctor*, *Pharmacist*, and *Administrator* classes define role-specific attributes and behaviours. This separation of concerns simplifies modifications, and updates without affecting unrelated components.
* **Open-Closed Principle (OCP):** Classes are built to be extendable without modifying existing code. For example, the Staff class abstracts shared attributes and behaviours for roles like *Doctor*, *Pharmacist*, and *Administrator*. For example, adding new roles, such as *Nurse* or *Technician*, simply involves creating subclasses that inherit from *Staff*, preserving the existing class hierarchy and ensuring scalability.
* **Liskov Substitution Principle (LSP):** Subclasses are interchangeable with their parent classes, ensuring that the system's behaviour remains consistent. In the HMS, subclasses like *Patient*, *Doctor*, *Pharmacist*, and *Administrator* inherit from the *User* class, making them substitutable for the parent class without breaking functionality.

A screen shot of a computer program

Description automatically generated

Figure 1.3 Code snippet of LSP implementation.

* **Interface Segregation Principle (ISP):** Interfaces are specific and focused, allowing classes to implement only the methods they require. For example, user-facing classes such as *DoctorMenu* or *AdminMenu* implement the *MenuInterface*, as depicted in Figure 1.4. Backend classes or components that do not involve user interaction are not burdened with irrelevant methods.A screenshot of a computer screen

  Description automatically generated

Figure 1.4 UML snippet of MenuInterface and its implementation

* **Dependency Inversion Principle (DIP):** High-level modules depend on abstractions rather than concrete implementations. This decoupling is achieved through dependency injection, simplifying integration and promoting flexibility. For instance, the HMS relies on a generic *Menu* abstraction rather than hardcoding implementations like *DoctorMenu* or *AdminMenu*. When a new role is added, we only need to implement the *Menu* interface without modifying higher-level classes. This means that high-level code is decoupled from the specific menu implementations.

#### Chapter 2.3.2. Design Patterns

The HMS employs several design patterns to enhance scalability, maintainability, and system control:

* **Factory Pattern:** Facilitates the creation of different user roles dynamically, ensuring the system can easily accommodate new roles without extensive modifications.
* **Singleton Pattern:** Ensures the HMS operates with a single-entry point, simplifying initialization and centralizing control over the application’s workflow.
* **Observer Pattern:** Allows objects to notify others of state changes, fostering a loosely coupled architecture that enhances maintainability and adaptability.

### **Chapter 2.4. Design Highlights**

#### Chapter 2.4.1. Class Hierarchies

The core class hierarchy centres around the *User* base class, from which role-specific subclasses (e.g., ***Patient***, ***Doctor***, ***Pharmacist***, ***Administrator***) inherit. To further distinguish responsibilities, we have categorised ***Doctor***, ***Pharmacist*** and ***Administrator*** under a ***Staff*** category to distinguish staff roles from patients in the HMS. This hierarchy allows each subclass to implement unique attributes and behaviours specific to each role, while common attributes and methods are encapsulated within the *User* class, promoting reusability and reducing redundancy across the system.

#### Chapter 2.4.2. Classes Interactions

In the HMS, boundary classes act as the front-facing interface which the user interacts with. These classes capture user input and forward it to the appropriate controller classes. The controller classes then coordinate the request by operating on the entity objects, updating or retrieving information via getters and setters.

## **Chapter 3. Additional Features & Enhancements**

**General**

|  |
| --- |
| 1. **Forget Password Request:** We added an option for users to request a password reset to default, pending Administrator approval. 2. **Password Decoupling and Hashing:** User passwords are securely hashed and stored separately from other personal information to enhance data protection. 3. **User-Friendly Interface:** Information is sorted and displayed neatly (typically in tables), and we allow users to make selections by entering an index (e.g., 1 or 2), simplifying navigation. 4. **Aesthetically Pleasing Interface:** Our team enhanced the program's appearance with features like styled headings, role-specific colours, and more for an improved user experience. 5. **Stringent Validation:** Ensures accurate input (e.g., valid data formats) and enforces logic (e.g., prevents dispensing medicine if stock is insufficient). |

**Patient**

|  |
| --- |
| 1. **Registration:** First-time users can easily register themselves as patients within our program. 2. **Rate Doctors:** After their appointments, patients can rate and provide feedback on doctors, enhancing care quality and sharing valuable experiences with others. 3. **View and Pay Bills:** Patients can view and pay their medical bills directly through the system. |

**Doctor**

|  |
| --- |
| **View Patient Ratings:** Doctors can view patient ratings, including an average score, to assess their performance and gauge patient satisfaction, helping them identify areas for improvement. |

**Administrator**

|  |
| --- |
| 1. **Manage Forgot Password Requests:** Administrators can approve or deny password reset requests for users who have forgotten their credentials. 2. **Bill Patients:** After each appointment, Administrators can bill patients accordingly for services rendered and medicine prescribed. |

## **Chapter 4. Functional Tests**

### **Chapter 4.1. Patient Actions**

Patient Main Menu Test Case 1: View Medical Record

|  |  |
| --- | --- |
|  |  |

Test Case 2: Update Personal Information Test Case 3: View Available Appointment Slots

|  |  |
| --- | --- |
|  |  |

Test Case 4: Schedule an Appointment

|  |
| --- |
|  |

Test Case 5: Reschedule an Appointment

|  |
| --- |
|  |

Test Case 6: View Available Appointment Slots

|  |
| --- |
|  |

Test Case 7: View Scheduled Appointments Test Case 8: View Past Appointment Outcome Records

|  |  |
| --- | --- |
|  |  |

### **Chapter 4.2. Doctor Actions**

Doctor Main Menu Test Case 9: View Patient Medical Records

|  |  |
| --- | --- |
|  |  |

Test Case 10: Update Patient Medical Records

|  |
| --- |
|  |

Test Case 11: View Personal Schedule Test Case 12: Set Availability for Appointments

|  |  |
| --- | --- |
|  |  |

Test Case 13: Accept or Decline Appointment Requests

|  |
| --- |
|  |

Test Case 14: View Upcoming Appointments

|  |
| --- |
| A black screen with white text  Description automatically generated |

Test Case 15: Record Appointment Outcome

|  |
| --- |
|  |

### **Chapter 4.3. Pharmacist Actions**

Pharmacist Main Menu Test Case 17: Update Prescription Status

|  |  |
| --- | --- |
|  |  |

Test Case 16: View Appointment Outcome Record

|  |
| --- |
|  |

Test Case 18: View Medication Inventory

|  |
| --- |
|  |

Test Case 19: Submit Replenishment Request

|  |
| --- |
| A screenshot of a computer screen  Description automatically generated |

### **Chapter 4.4. Administrator Actions**

Administrator Main Menu Test Case 20: View and Manage Hospital Staff

|  |  |
| --- | --- |
|  |  |

Test Case 21: View Appointments Details

|  |
| --- |
|  |

Test Case 22: View and Manage Medication Inventory

|  |
| --- |
| A screenshot of a computer program  Description automatically generatedA screenshot of a computer program  Description automatically generated |

Test Case 23: Approve Replenishment Requests

|  |
| --- |
|  |

### **Chapter 4.5. Login System & Password Management**

Starting the Program Exiting the Program

|  |  |
| --- | --- |
|  |  |

Test Case 24: First-Time Login and Password Change Test Case 25: Login with Incorrect Credentials

|  |  |
| --- | --- |
|  |  |

## **Chapter 5: Reflecting on our Voyage**

### **5.1. Polymorphism of Challenges**

Our journey in SC2002 Object-Oriented Design & Programming has been nothing short of challenging yet rewarding. Translating theoretical concepts like Polymorphism, SOLID principles, and the Boundary-Control-Entity (BCE) model into practical solutions tested our ability to implement clean and effective designs. What seemed logical in lectures often revealed unexpected complexities during application, forcing us to iterate and refine our approach.

The scale of the project pushed us to manage heavily interconnected classes and roles while adhering to decoupling principles. Balancing this complexity alongside overloaded academic schedules added significant pressure. Late nights became the norm as we navigated deadlines, troubleshooting code and refining workflows. On top of this, we dealt with an unresponsive teammate, which further strained our group dynamics and motivation.

### **5.2. Debugging the Group Dynamics**

To address these hurdles, we established clear communication, regular check-ins, and rotated leadership roles to manage workloads. Open discussions allowed us to realign our tasks and maintain progress. For the “Null Pointer” teammate issue, we sought support from our T.A. and professor, ultimately resolving the situation and improving team cohesion.

### **5.3. Committing Knowledge Gained to Brain Repository**

This project transformed abstract concepts into real-world skills. Polymorphism, SOLID principles, and design patterns became tools we could wield confidently. Beyond technical growth, we gained invaluable project management experience—learning to communicate effectively, adapt to setbacks, and distribute tasks efficiently. Above all, we developed resilience, approaching challenges with flexibility and determination.

While the journey was demanding, it sharpened both our technical expertise and collaborative abilities, leaving us better prepared for future projects and professional endeavours.

**End**