SC2002 – OBJECT-ORIENTED DESIGN AND PROGRAMMING

Project Title: Hospital Management System(HMS)

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	Lab Group	Signature /Date
Babu Sankar Nithin Sankar	CSC Y2	SCS6	Nithin Sankar B/17-11
Risha Sunil Shetty	CSC Y2	SCS6	Risha Shetty/17-11
Lau Zhan You	CSC Y2	SCS6	Lau Zhan You/17-11
Singh Janhavee	CSC Y2	SCS6	Janhavee/17-11
Tan Chang Lin	CSC Y2	SCS6	Tan Chang Lin/17-11

1. Executive Summary

This report explains the design considerations and Object-Oriented Programming (OOP) concepts applied in developing our Hospital Management System (HMS). The HMS is a Java Command Line Interface (CLI) application designed to manage hospital users and operations. The document showcases our Unified Modeling Language (UML) Class Diagram, the application of S.O.L.I.D design principles, and key OOP concepts that ensure the modularity and scalability of the application. It also includes the outcomes for various tested user roles, detailed test cases, and their results. Additionally, the reflection section discusses the challenges encountered during

development, the solutions implemented, and suggestions for future improvements. Overall, this report provides a comprehensive analysis of the design principles and OOP methodologies utilized in creating the HMS.

2. Design Considerations

2.1. Approach Overview

Our HMS was designed with object-oriented principles and S.O.L.I.D design in mind, ensuring a modular, maintainable, and scalable architecture. We used the Model-View-Controller (MVC) pattern to clearly separate responsibilities: models for entities, controllers for user input, and views for information presentation in the CLI. We also implemented Service classes, Interfaces, and Enums to handle business logic.

Each user role in the HMS has distinct responsibilities and access controls, implemented in separate classes. This approach provides flexibility, scalability, and a clear separation of concerns. By adhering to S.O.L.I.D principles, we ensured strong class relationships and a user-centric, structurally sound system that is reliable for hospital management.

2.2. Assumptions Made

a. The different stakeholders (Doctors/ Patients/ Pharmacists/ Administrator) are aware of how to use the HMS and how to operate it depending on their respective roles. Therefore, the information presented in the Views classes would be concise and not as descriptive.

b. All data that is saved in the comma separated values (CSV) files are properly formatted and would not require data cleaning when importing the data during the HMS start up.

2.3. Object-Oriented Concepts Used

2.3.1. Abstraction

We used abstraction to simplify the complexity of HMS by identifying essential features and temporarily hiding redundant details that could distract from understanding the system. For instance, interfaces were used abstractly to implement the finer details of Services and Views, as illustrated in Fig. 2.3.1a.

```
public class AppointmentService implements IAppointmentService {
    private static final String APPOINTMENT_FILE = "data/appointment.csv";
    private List<Appointment> appointments = new ArrayList();
```

Fig. 2.3.1a: AppointmentService implements IAppointmentService - an interface

2.3.2. Encapsulation

Encapsulation was used in our HMS application to safeguard the internal state of objects. We did so by making attributes private and allowing access or modification only through public getter and setter methods. This controlled approach prevented unauthorized data access by other classes.

2.3.3. Inheritance

In our HMS application, inheritance is used to establish a parent-child relationship between classes. The User class serves as the base, providing common attributes like hospitalID, password, and role, along with methods such as getHospitalID() and setPassword(). Subclasses for each user role—Patient, Doctor, Pharmacist, and Administrator—inherit these properties and methods while adding role-specific functionality. For example, the Patient class inherits from User and extends it with additional methods relevant to patients. This inheritance structure is illustrated in the UML Class Diagram in Section 3.

2.3.4. Polymorphism

Polymorphism was used to allow objects to take on multiple forms and behaviors. For example, the different subclasses of the User class are able to override methods such as the toString(), changePassword() and login() (Fig. 2.3.4a).

```
 */
@Override
public boolean login(String hospitalID, String password) {
    User user = users.get(hospitalID);
    return user != null && user.getPassword().equals(password);
}
```

Fig. 2.3.4a: Login() method

2.4. Design Principles

2.4.1. Single Responsibility Principle (SRP)

Our HMS application assigns each class a single, well-defined responsibility in order to ensure modularity, maintainability and extensibility of our application. This structure reduces interdependencies between classes which in turn simplifies debugging and also provides the option for future developments. For example:

a. User Model Classes

Each role such as Patient, Doctor, Pharmacist and Administrator has its own class. These classes focus on actions that are relevant to its own role while additional separate classes handle other responsibilities such as appointments, scheduling and inventory; amongst others.

For example, the Doctor class manages the patient's medical records and appointment outcomes while the Patient class manages the viewing and scheduling of appointments. This clear separation ensures that changes to a class functionality does not affect other classes.

b. Controller Classes

Our controllers manage the endpoint user inputs which is the flow of data and user interactions with our application. For example, the AppointmentController coordinates appointment-related requests like booking or rescheduling while the InventoryController handles the different medications stock level and replenishment requests. Each controller focuses on a single area of functionality, reducing their dependencies on each other.

c. Service Classes

The different service classes handle the different business logic for each system function. For example, AppointmentService manages the operations that are related to appointments such as verifying availability and confirming bookings while InventoryService manages inventory updates and alerting if medication stocks drop to threshold level. This distinct separation enables easy modification and testing of each of the service functionalities.

d. Interfaces

These are classes that define a set of related methods that must be implemented by a concrete class. For example, IAppointmentService , IInventoryService, IPatientView, etc.

e. Enums

This defines a set of constant values that will be used throughout our HMS application. For example, AppointmentStatus, InventoryStatus, UserRole, etc.

2.4.2. Open-Closed Principle (OCP)

Our HMS allows for easy extensibility without modifying existing code. By using interfaces, components are loosely coupled, enabling seamless integration of new features. For example, adding a new User role is as simple as implementing the IUserService interface without affecting existing functionality.

Services and views implement specific interfaces, ensuring flexibility and easy replacement. For instance, the AppointmentService implements the IAppointmentService interface, allowing us to add new scheduling logic without impacting the rest of the system. Similarly, IInventoryService enables future inventory management approaches through additional classes that implement the interface.

2.4.3. Liskov Substitution Principle (LSP)

Our HMS ensures that any subclass is able to replace its parent class without altering the application's expected behavior. This is done through consistent use of inherited attributes and methods in the derived classes, without introducing new pre-conditions or weakening any post-conditions.

In our case, the different User roles such as Patient, Doctor, Pharmacist and Administrator inherits a common functionality from the base class User. The base class contains shared attributes and methods whereas each subclass also provides its own additional unique methods that it requires. The following illustrates how our application adheres to LSP:

a. User Role Compatibility

The methods defined in the User base class were implemented consistently across the different User roles. This allows each subclass to be substituted anywhere a User instance is expected without disrupting the applications functionality.

b. Service Layer Implementation

Services in our application that handle various operations such as appointment scheduling and inventory updates were designed with LSP in mind. For example, AppointmentService could be easily substituted out by any class that provides appointment-related methods defined by its

interface. This allows for the flexibility in modifying or further extending the appointment handling functionality without impacting other application components.

2.4.4. Interface Segregation Principle (ISP)

Our implementation of interfaces were tailored to specific needs of the different classes. Classes that implement the interfaces use all the methods defined in the interface. This ensures that the classes only implement methods that are required. This modular interface structure minimizes dependency on unnecessary methods, making our application more adaptable and efficient. For example:

a. Role Specific Service Interface

Our application contains separate interfaces to handle role-specific functionalities. For example IPatientService provides methods relevant to the patient function while IPharmacistService provides methods relevant to the pharmacist functions. This segregation ensures that each role's functionality remains isolated, allowing for a more specialized service for the respective roles to be created without altering unrelated interfaces.

b. Additional Management Interfaces

Additional interfaces such as IInventoryService was created for inventory-related functions and IAppointmentService for scheduling. This further encapsulates the functionalities that are specific to inventory and appointment management, ensuring that only the relevant classes implement these interfaces.

2.5. Other Design Considerations

2.5.1. Foreign Key Association

In our HMS application, the use of foreign key associations are designed to establish relationships between entities and maintain data consistency across the various user roles and their associated records. This is done so instead of using object composition/aggregation.

When an object detail changes, the other objects that contain the updated object will still have the updated information about the object. This is due to the fact that only one instance of the updated object exists in the HMS application, thus ensuring data integrity. For example:

a. Patient and Appointment Association

Each appointment is linked to a specific patient through the patientID instead of the Patient objects *(Fig. 2.5.1a / Fig. 2.5.1b)*. This ensures that the patient records accurately correspond with their appointment history. This connection allows retrieval of patient-specific appointments, enhancing the organization and integrity of the respective patient records.

To retrieve the patient and appointment objects, the public getter methods are used to directly access the data store and return the required objects.

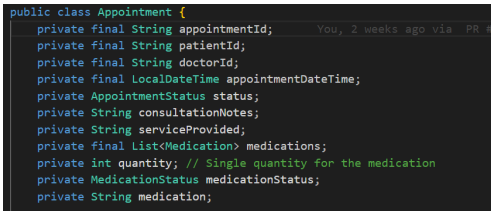


Fig. 2.5.1a: Appointment's patientId string attributes

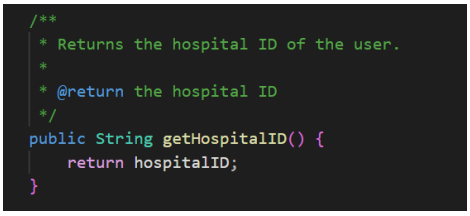


Fig. 2.5.1b: Public getter methods

2.5.2. Repository Pattern

In our HMS Application, we implemented a downscaled and simplified version of the Repository Pattern. This is done through the DataStore classes in the stores folder. The InventoryDataStore class acts as a repository for the data stored in the data files and there exists methods defined within to access this data. This pattern allows the data layer to be isolated from the rest of the application and allows the abstraction of the data storage implementation details.

This makes it easy to switch to different data storage implementations subsequently without affecting the rest of the HMS application significantly.

3. Additional Implementations

Apart from the required test cases, we implemented several additional features to enhance the functionality and security of our HMS. These include a robust billing system for accurate patient charges and payment processing, as well as password encryption to safeguard sensitive information and ensure secure access.

3.1. Billing

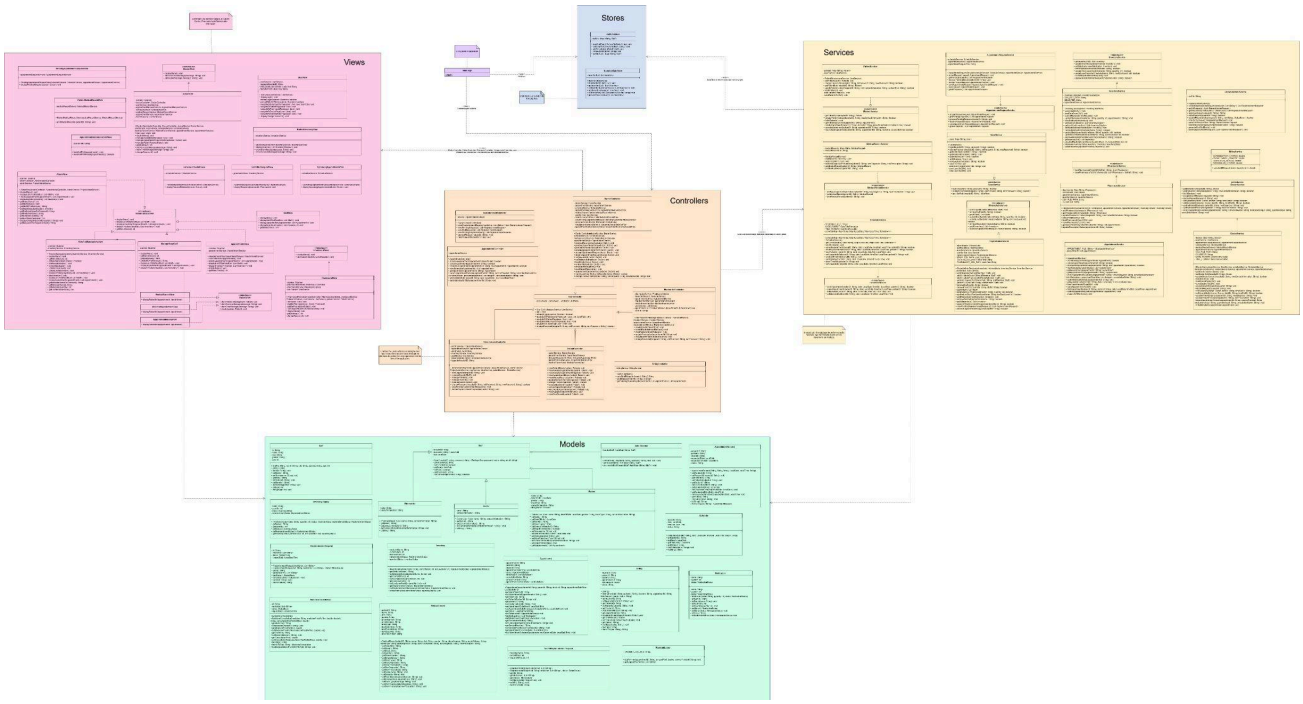
We implemented a billing function to efficiently handle patient charges, including consultations, and medications services. It ensures accurate calculations of total costs, and generates detailed invoices for patients, also giving them an option to make the payment. The system is designed to integrate seamlessly with other HMS modules.

3.2. Password Encryption

To enhance security, we implemented password encryption within the HMS to protect sensitive data. This ensures that passwords stored in the system are securely encrypted, preventing unauthorized access to CSV files. Additionally, a function was added for the administrator to securely retrieve and manage passwords when needed, ensuring both confidentiality and ease of access for authorized personnel.

4. UML Class Diagram

Our UML Class Diagram with UML package notation shows the class relationships and dependencies between different classes and packages. The notes in the diagram provide additional explanation about some of the design choices and patterns for the HMS. A clearer diagram image is available in a separate file named `hms-uml-classdiagram.jpg`.



5. Functional Tests and Results

We implemented a suggestive but not exhaustive list of test cases to validate the core functionalities, ensure proper behavior of the application components, and prevent any error from breaking the system. Additional End-to-End Tests can be found in the attached pdf named e2e-tests.pdf.

5.1. Authentication

S/N	Test Description	Expected Result	Pass/Fail
01	**[Fail to login]** (a) Enter wrong/invalid userID for the selected role. (b) Enter wrong/invalid password for a correct userID.	When (a) and (b) occurs, the user would not be able to login. The user is given 3 attempts before being prompted if they want to exit.	Pass
02	**[Change Password]** (a) Upon successful login, the user changes their password by entering the old password. (b) Upon successful login, the user changes their password by entering the new password.	(a) User fails to change the password and an error message is shown. (b) Passwords changed. Logging into the same account with the new password is required subsequently after logging out. Relevant data files are updated.	Pass

5.2. Patient Functionality

S/N	Test Description	Expected Result	Pass/Fail
01	**[Create Appointment]** (a) Patient keys in the doctorID of the doctor they would like to see, an appointment date and select an appointment time slot for that day.	(a) Patients successfully booked an appointment with the specified doctorID on a particular date and time. Relevant data files are updated. (b) Patient chooses a date with a scheduled appointment, and an error message is thrown to inform them to choose another day. (c) Patient has indicated an invalid date format, and an error message is thrown to guide them on the format required. (d) Patient indicates an unavailable time slot, and an error message is	Pass

S/N	Test Description	Expected Result	Pass/Fail
		thrown to inform them to choose another time slot.	
02	**[View Available Appointment Slots]** (a) Patient enters the doctorID and date to view the available time slots of that particular doctor on that particular date.	(a) Patient successfully keys in a valid doctorID and a valid date with available time slots to view the list of available time slots. (b) Patient indicates an invalid date with no available time slots, and an error message is thrown to inform the patient.	Pass

5.3. Doctor Functionality

S/N	Test Description	Expected Result	Pass/Fail
01	**[Update Patient Medical Record]** Doctor chooses either; 1. Add new diagnosis for patient a. Doctor enters the diagnosis for the patient. 2. Add new prescription for patient a. Doctor enters the prescription for the patient.	Doctor chooses option 1: (a) The existing diagnosis in the "New Diagnosis" column in the data file will be concatenated with the content in the "Past Diagnosis" field. (b) The diagnosis entered by the doctor will be updated in the "New Diagnosis" columns of the data file. (c) Successful update message is displayed Doctor chooses option 2: (a) The existing prescription in the "New Prescription" column in the data file will be concatenated with the content in the "Past Prescription" field. (b) The prescription entered by the doctor will be updated in the "New Prescription" columns of the data file. (c) Successful update message is displayed	Pass
02	**[Accept or Decline Appointment Request]** Doctor chooses either; 1. Accept Appointment Request a. Doctor enters the RequestID of the	Doctor chooses option 1: (a) Doctor enters requestID (b) Display error message if requestID entered does not exist (c) the doctor's personal schedule, the status of the slot requested by the	Pass

	appointment request the doctor wishes to accept. 2. Decline Appointment Request a. Doctor enters the RequestID of the appointment request the doctor wished to decline	patient will change from “Available” to the patientID of the patient that requested the appointment slot. Relevant data files are updated. (d) The status of the Appointment request is changed from “Pending” to “Confirmed”. Relevant data files are updated. Doctor chooses option 2: (a) Doctor enters requestID (b) Display error message if requestID entered does not exist. (c) The status of the Appointment request is changed from “Pending” to “Canceled”. Relevant data files are updated.	

5.4. Error Handling

S/N	Test Description	Expected Result	Pass/Fail
01	**[Invalid Inputs]** Entering an invalid input when for userID, password or when selecting from the menu list.	A message that prompts the user either to provide a valid input or the error occurred.	Pass

6. Reflection

6.1. Difficulties Encountered

We initially faced challenges integrating different parts of the code due to differing ideas on how inputs and outputs would be passed. To resolve this, we broke the integration down into smaller subproblems, solving them one at a time, and then adjusted the remaining code accordingly.

Secondly, we faced initial difficulties in the earlier stages when trying to adhere to the SRP, especially without a well-defined system architecture. Balancing SRP with its practical implementation was challenging as each module's functionality had to be isolated while ensuring seamless interaction between the models, controllers, services and data. Subsequently, we overcame this challenge by implementing the MVC architecture, supplemented with services and stored packages- enabling us to separate the application's concerns effectively.

6.2. Knowledge Learnt

From brainstorming the implementations to the actual development of our HMS application, our team gained a much deeper understanding and appreciation of various OOP concepts, S.O.L.I.D principles, and best practices in software development. We learned the importance of creating a clear UML class diagram to accurately represent the application's functionality, ensuring all team members shared the same understanding of classes, inputs, and outputs. Adopting a well-organized architecture, particularly the MVC model, proved essential for our codebase development. We also realized that adhering to specific conventions and principles made our development process more efficient. Using S.O.L.I.D principles enabled easier extension and maintenance of the HMS application, while strict adherence to naming conventions, such as camel casing and using "I" to prefix interfaces, ensured consistency. Moreover, domain-driven class names helped make the functionalities of each file more understandable and accessible, contributing to a more cohesive and maintainable application.

6.3. Further Improvements

In the future, we could enhance the CLI interface to provide a more user-friendly and intuitive experience by incorporating features like auto-complete and detailed command descriptions, making it easier for users to navigate the system. Additionally, we could increase the versatility of our HMS application by introducing a user registration feature, allowing users to create accounts directly through the CLI. Implementing a notification system could also add value by reminding patients of upcoming appointments or alerting pharmacists about low inventory. Furthermore, a smart chatbot could be developed to assist patients in accessing information about appointments, medications, or other hospital-related queries. This could be achieved through a RAG workflow, embedding large volumes of information into vector embeddings, and storing it in a vector database for efficient querying. These enhancements would significantly improve user engagement and the overall functionality of the HMS application.