NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC2002 OBJECT ORIENTED DESIGN & PROGRAMMING

HOSPITAL MANAGEMENT SYSTEM

Report of Project Structure Design & Functionality

AY24/25 Sem 1 | SCEA, Group 5

GitHub Main Page and Repository

JavaDoc

Setup Instructions

Declaration of Original Work for SC/CE/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 the 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
Kumar Advaith U2323530B	SC2002	SCEA - Grp 5	KUMAR ADVAITH
Lam Wai Jun U2321478F	SC2002	SCEA - Grp 5	LAM WAI JUN
Shu Xuanyu U2320246L	SC2002	SCEA - Grp 5	SHU XUANYU
Singh Gunraj U2323772B	SC2002	SCEA - Grp 5	SINGH GUNRAJ
Tan Zhi Xin U2321371F	SC2002	SCEA - Grp 5	TAN ZHI XIN

1. DESIGN CONSIDERATIONS

1.1 Overview of the Design approach

The Hospital Management System (HMS) was designed with a **modular structure** that adheres to **Object-Oriented Principles** and **SOLID design principles**. This structure enhances the HMS's **extensibility, maintainability, and ease of testing**.

To achieve modularity, the HMS application is divided into distinct packages, each with a well-defined responsibility. Additionally, a **data storage approach using text files** serves as a simplified repository for persistence, handling core data like appointments and inventory levels. This choice prioritizes ease of access and modification, though it could be upgraded to a database for scalability in future iterations.

1.2 Object-Oriented Concepts

1.2.1 Abstraction

Abstraction was employed in the Hospital Management System (HMS) to reduce complexity by hiding implementation details and exposing only essential functionalities to other parts of the application.

The User class is an abstract superclass that defines common attributes and methods for all user types (e.g., Doctor, Patient, Administrator). By defining a core set of behaviors in User, specific roles only need to implement their unique functionality, without altering shared behaviors.

For instance, the displayMenu method is declared abstract in User, and each subclass implements it based on its role-specific requirements. This method shows the User class exposing a common interface while allowing subclasses to implement their unique displayMenu behavior, illustrating the use of abstraction.

1.2.2 Encapsulation

Each class, such as Patient, Doctor, and Appointment, uses private fields for sensitive information, such as userID, appointmentDate, and medicalRecord. This restricts direct access to these fields from outside the class, enforcing **data hiding**.

To allow controlled access, each class provides public getter and setter methods where necessary. This enables other parts of the system to interact with the data without exposing the underlying structure or allowing unrestricted modification.

For instance in the Patient class we have applied encapsulation on a property medicalRecord by restricting direct access to medicalRecord while exposing controlled access through getMedicalRecord() and setMedicalRecord().

1.2.3 Inheritance

Inheritance is used in the Hospital Management System (HMS) to create a hierarchical relationship among different types of users, allowing shared behaviors and properties to be inherited from a common parent class. This approach promotes code reuse and consistency across similar entities.

The User class serves as a base class for user roles, such as Doctor, Patient, Administrator, Receptionist, and Pharmacist. These subclasses inherit common properties (e.g., userID, password) and methods from User, reducing redundancy and enabling each role to share a unified interface while adding specific functionality where needed.

For example, all user roles inherit the displayMenu() method, but each subclass implements it differently according to the role's responsibilities.

In our code, Doctor inherits from User and overrides displayMenu to provide a role-specific implementation, illustrating inheritance in the user hierarchy.

1.2.4 Polymorphism

Polymorphism in the HMS allows objects to take on multiple forms, enabling methods to be used interchangeably across different user roles and facilitating dynamic behavior within the system. The displayMenu() method in the User class is a prime example of polymorphism. It's defined as an abstract method in User and is implemented by each subclass (e.g., Doctor, Patient). This polymorphic behavior enables the HMS to call displayMenu() on a User object, regardless of the specific subclass, and each role will display its respective menu.

In our code, the showUserMenu() function can accept any subclass of User and call displayMenu(). The appropriate menu is displayed based on the actual object type (e.g., Doctor, Patient), demonstrating polymorphism in action.

1.3 Applied Design Principles

1.3.1 Single Responsibility Principle (SRP)

The Hospital Management System (HMS) adheres to the Single Responsibility Principle by organizing classes into packages with each package handling a distinct responsibility. This ensures that each class focuses on a specific task, making the system modular, easier to maintain, and adaptable to future requirements.

1. **User Management**:
    - Classes like Doctor, Patient, Administrator, Receptionist, and Pharmacist each manage different user roles, with specific responsibilities tailored to their functions. For instance, a Doctor is responsible for scheduling and managing appointments, while Patient focuses on viewing appointments and medical records. This separation ensures that each role can be updated independently.
2. **Appointment Management**:
    - The AppointmentManager class is solely responsible for scheduling, rescheduling, and canceling appointments. It encapsulates appointment-specific logic, allowing other parts of the system to use its functionality without interfering with the internal details of the appointment management process.
3. **Inventory Management**:
    - InventoryManager and Inventory classes handle the stock management of medicines. InventoryManager focuses on tracking and updating inventory levels, while Inventory holds the list of medicines. By isolating inventory functionality, the system is more maintainable, and adjustments to stock management can be made without impacting other modules.
4. **Billing and Medical Records**:
    - The Billing and MedicalRecords components manage billing and patient records, respectively. Billing takes care of generating invoices and managing payment details, while MedicalRecords deals with patient history and treatment records. Each class operates independently, adhering to SRP by managing only the responsibilities relevant to its domain.

1.3.2 Open/Closed Principle (OCP)

The HMS system is designed to be open for extension but closed for modification, allowing future features to be added with minimal changes to existing code.

Each major component of the system, such as appointments, inventory, and billing, is managed by dedicated manager classes. These manager classes define a set of methods for interacting with the underlying data without exposing the implementation details. For instance, new functionality in appointment handling or inventory management can be added by extending AppointmentManager and InventoryManager, respectively, rather than modifying them directly.

Extending User Roles:

The User class allows new roles to be created by extending it. For example, if a new role such as LabTechnician is needed, it can be added as a subclass of User with specific methods, without altering existing roles or the User superclass.

User Class:

The User class is an abstract superclass that serves as a foundation for user roles like Doctor, Patient, Administrator, etc. It defines the core attributes and methods that all users need, such as userID and password. Each subclass extends User, adding specific fields and methods that apply only to that role.

For instance, if a new role such as LabTechnician needs to be added, it can inherit from User and implement its own functionalities without changing the existing user roles.

1.3.3 Liskov Substitution Principle (LSP)

The Hospital Management System (HMS) follows the **Liskov Substitution Principle** by designing derived classes that can seamlessly substitute their base classes without altering the correctness or behavior of the system. This is achieved by ensuring that each subclass adheres to the expectations set by the User superclass, without introducing additional preconditions or altering postconditions.

User Hierarchy and LSP Compliance:

The User class defines a common interface and behavior for different user roles (e.g., Doctor, Patient, Administrator), with each role inheriting from User.

For example, methods like displayMenu() are implemented in a way that allows any subclass to substitute User in polymorphic contexts. This means that any function expecting a User object can operate correctly with any of its subclasses without needing to know the specific type. Each subclass (e.g., Doctor, Receptionist) adheres to the guarantees provided by User and does not impose stronger conditions. This ensures compatibility and prevents runtime errors or unexpected behavior when substituting different types of users. For instance, both Doctor and Patient classes can be used wherever User is expected, as they conform to the same interface, fulfilling LSP.

1.3.4 Interface Segregation Principle (ISP)

While the HMS does not explicitly use interfaces, it follows the **Interface Segregation Principle** in the sense that each class implements only the methods necessary for its specific role, ensuring that classes are not burdened with irrelevant functionality. This implicit ISP approach is achieved by designing each user role and manager class with a distinct set of responsibilities.

1. **Role-Specific Responsibilities**:
    - Each subclass of User implements only the functionality required for its role. For instance, Doctor includes methods to view appointments and manage patient records, whereas Pharmacist has methods for managing prescriptions. By separating these responsibilities, each role is only equipped with the methods it actually uses, preventing dependency on unnecessary functionality.
2. **Encapsulation of Manager Classes**:
    - Classes like AppointmentManager and InventoryManager provide a clear set of methods for managing appointments and inventory, respectively. This limits external classes to specific interactions (e.g., scheduleAppointment, updateStock) without exposing irrelevant operations, thus following ISP by dividing functionality based on service-specific needs.

1.3.5 Dependency Inversion Principle (DIP)

The Hospital Management System (HMS) adheres to the **Dependency Inversion Principle** by ensuring that high-level classes depend on abstractions rather than specific implementations.

This allows components to interact with one another through defined interfaces and abstract classes, improving flexibility and scalability by minimizing coupling between modules.

(a) Manager Classes as Abstractions

In HMS, high-level classes like Reception and Administrator interact with **manager classes** (e.g., AppointmentManager, InventoryManager) rather than handling appointment or inventory data directly. These manager classes encapsulate specific functionalities, acting as an abstraction layer that reduces dependency on the underlying data structures.

(b) Appointment and Inventory Management:

Reception uses AppointmentManager to handle appointment-related operations without needing to understand the internal data storage or scheduling logic. Similarly, InventoryManager provides an interface for managing inventory data.

This setup allows components like Reception and Administrator to rely on AppointmentManager and InventoryManager abstractions, making it easier to modify or replace appointment and inventory management implementations without impacting the rest of the system.

For example, Reception depends on the AppointmentManager abstraction, promoting DIP by separating the logic for scheduling appointments from the high-level user interaction logic.

(c) Billing and Invoice Management:

Billing encapsulates invoice creation & payment calculation. Reception interacts with Billing to handle payment tasks without depending on concrete details of the billing logic, making it easy to modify how invoices are managed without affecting classes that rely on it.

The User abstract class establishes a high-level abstraction for different user roles. Subclasses like Doctor, Patient, and Administrator extend User, ensuring that any references to User can interact with these roles interchangeably.

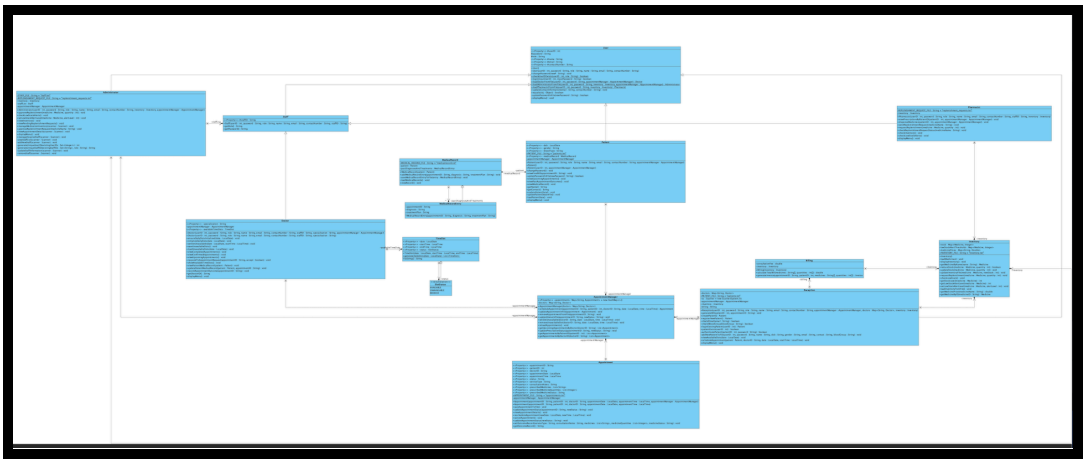
1. **User Abstraction**:
    - The User class provides a common interface for user roles. This means that high-level components interacting with users do so through User rather than any specific subclass, reducing dependency on particular role implementations and making it easier to introduce new roles without affecting existing functionality.

This setup ensures that high-level modules depending on User are unaffected by changes to specific user roles, promoting DIP by relying on the abstract User class instead of concrete implementations.

1.4 Assumptions Made

1.4.1 User Authentication: All staff members added by the administrator are assigned a non-changeable User ID and a default password, which can be changed after their initial login.
1.4.2 Doctor Availability: Doctors are assumed to be available for consultations every day, with their appointment slots automatically initialized daily. This setup allows patients to schedule appointments without gaps in doctor availability. Doctors have the option to mark specific slots as unavailable on particular dates if needed.
1.4.3 Lunch Break Scheduling: All doctors have a scheduled lunch break at 1:00 PM, during which appointments cannot be booked. This time slot is automatically marked as unavailable.
1.4.4 Inventory Replenishment: When the administrator approves a replenishment request and the stock level is found to be below the set threshold, the inventory is automatically updated to twice the threshold value to ensure sufficient stock availability.
1.4.5 Single Administrator: The system assumes that there is only one administrator account with full control over functionalities of managing staff, inventory, and appointments. There is no functionality for multiple administrators or delegation of administrative tasks.
1.4.6 Appointment Slot Duration: Each appointment slot is assumed to have a fixed duration of 30 minutes. This duration is consistent for all doctors and cannot be adjusted individually.
1.4.7 Default Appointment Status: All appointments that are scheduled by the patients are to start with a default status of "scheduled". An appointment is confirmed only when the doctor changes the status to "confirmed". Only then the slot becomes unavailable to other patients.
1.4.8 Inventory Quantity in Integer Units: Inventory quantities are assumed to be in whole numbers, with no support for decimals. This is relevant for dosage management in prescriptions.
1.4.9 Fixed Threshold Values for Replenishment: The threshold values for inventory replenishment are assumed to be set by the administrator and are consistent across different medicines. There is no functionality for adjusting threshold values per item.
1.4.10 Medicine Status: Doctor always sets the prescribed medicine status to = "Prescribed". We assume there will be no typing errors in this.

2. DETAILED UML CLASS DIAGRAM - (ATTACHED SEPARATELY)



3. ADDITIONAL FEATURES & FUNCTIONALITIES:

3.1 Bill Generation: The system generates bills for patients based on the services provided, including consultation fees, prescribed medications, and any other applicable charges. This bill can be viewed and printed by the patient or staff as needed.

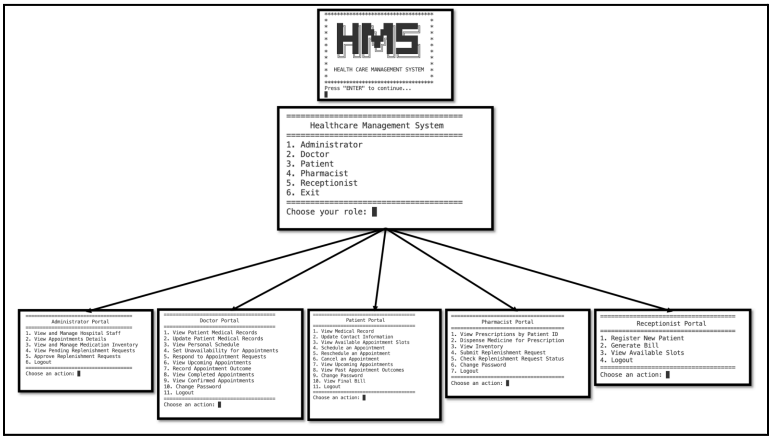
3.2 Flexible Appointment Booking and Scheduling: Patients can book appointments with doctors by selecting specific dates and times. The system displays available slots for each doctor, allowing patients to choose based on their convenience and the doctor's availability.

3.3 Appointment Date Validation: The system restricts users from booking appointments in the past. This validation ensures that all scheduled appointments are set for valid, future dates only.

3.4 Dynamic Inventory Management for Prescribed Medicines: If a doctor prescribes a medicine that is not currently in the inventory, the administrator has the option to add that specific medicine to the inventory. This feature ensures that all prescribed medicines are available for patients to collect from the pharmacy.

4. TESTING

4.1 User Roles and Functionalities:



4.2 Additional Functional Tests and Results

4.2.1 Bill generation

```
====================================
          Receptionist Portal
====================================
1. Register New Patient
2. Generate Bill
3. View Available Slots
4. Logout
====================================
Choose an action: 2
Enter Patient ID: 412
Enter Appointment ID: A1731733760399
-------- Invoice --------
Appointment ID: A1731733760399
Patient ID: 412
Consultation Fee: $50.0
Medicines:
Medicine             Quantity   Price
cisplatin            10         $121.00
------------------------
Total Amount: $171.00
```

```
====================================
            Patient Portal
====================================
1. View Medical Record
2. Update Contact Information
3. View Available Appointment Slots
4. Schedule an Appointment
5. Reschedule an Appointment
6. Cancel an Appointment
7. View Upcoming Appointments
8. View Past Appointment Outcomes
9. Change Password
10. View Final Bill
11. Logout
====================================
Choose an action: 10
Enter Appointment ID to view final bill: A1731733760399
-------- Invoice --------
Appointment ID: A1731733760399
Patient ID: 412
Consultation Fee: $50.0
Medicines:
Medicine             Quantity   Price
cisplatin            10         $121.00
------------------------
Total Amount: $171.00
Press "ENTER" to continue...
```

4.2.2 Flexible Appointment Booking and Scheduling

```
Enter date to view available slots (yyyy-mm-dd): 2024-11-19
=======================================================
| Doctor ID | Name                | Specialization   |
=======================================================
| D003      | Emily Brown         | ONCOLOGY         |
| D002      | Sarah Johnson       | CARDIOLOGY       |
| D009      | Barbara Purple      | NEPHROLOGY       |
| D008      | Robert Blue         | GASTROENTEROLOGY |
| D007      | Patricia Grey       | GYNECOLOGY       |
| D006      | James Black         | DERMATOLOGY      |
| D005      | Linda White         | ORTHOPEDICS      |
| D004      | Michael Green       | PEDIATRICS       |
=======================================================
```

```
Enter the Doctor ID to view available time slots: D002
Available Time Slots for Doctor Sarah Johnson on 2024-11-19:
================================================
| Date       | Start Time | End Time | Status       |
================================================
| 2024-11-19 | 09:00      | 09:30    | AVAILABLE    |
| 2024-11-19 | 09:30      | 10:00    | AVAILABLE    |
| 2024-11-19 | 10:00      | 10:30    | AVAILABLE    |
| 2024-11-19 | 10:30      | 11:00    | AVAILABLE    |
| 2024-11-19 | 11:00      | 11:30    | AVAILABLE    |
| 2024-11-19 | 11:30      | 12:00    | AVAILABLE    |
| 2024-11-19 | 12:00      | 12:30    | AVAILABLE    |
| 2024-11-19 | 12:30      | 13:00    | AVAILABLE    |
| 2024-11-19 | 14:00      | 14:30    | AVAILABLE    |
| 2024-11-19 | 14:30      | 15:00    | AVAILABLE    |
| 2024-11-19 | 15:00      | 15:30    | AVAILABLE    |
| 2024-11-19 | 15:30      | 16:00    | AVAILABLE    |
| 2024-11-19 | 16:00      | 16:30    | AVAILABLE    |
| 2024-11-19 | 16:30      | 17:00    | AVAILABLE    |
================================================
```

4.2.3 Appointment Date Validation

```
Choose an action: 4
Enter the Doctor ID to book an appointment: D002
Enter date (yyyy-mm-dd) for appointment: 1900-11-23
Date cannot be in the past. Please enter a future date.
```

4.2.4 Dynamic Inventory Management for Prescribed Medicines

```
Choose an action: 5
Enter the name of the medicine to approve replenishment: Med1
Medicine 'Med1' does not exist in inventory.
Do you want to approve adding it to inventory? (yes/no): yes
Enter Medicine Name: Med1
Enter Stock Quantity: 10
Enter Stock Threshold: 5
Enter Price per Unit: 3.0
Inventory saved successfully.
Medicine 'med1' added successfully.
Replenishment request approved and medicine added to inventory.
```

5. REFLECTION

5.1. Difficulties Encountered & Solutions to Conquer

5.1.1 Ensuring Adherence to the Single Responsibility Principle & Separation of Concerns: One of the main challenges was designing each class with a single responsibility, especially as the complexity of the HMS grew with different user roles and management functionalities. Initially, some classes tended to become too large as they handled multiple responsibilities. We overcame this by refactoring our design to ensure each class focused on a specific task (e.g., separating appointment logic into AppointmentManager and inventory logic into InventoryManager). This approach helped create a clear separation of concerns and made the code more modular and easier to maintain.

5.1.2 Managing Data Persistence Across Sessions:
Another challenge was implementing data persistence using text files to store user data, appointments, and inventory information. Since Java does not natively support database-like storage in text files, we encountered difficulties in designing a structure that could reliably read and write data while maintaining data integrity. To address this, we created methods to save and load data systematically whenever the program starts or exits. However, this process highlighted limitations in using text files, as complex relationships (like appointments tied to specific doctors) required careful handling to prevent data inconsistencies.

5.2. Knowledge Learnt from this course

5.2.1 Importance of Object-Oriented Design and SOLID Principles:

- Throughout the project, we gained a deeper understanding of the importance of Object-Oriented Design principles, particularly SOLID principles. By enforcing these principles in our design, we saw firsthand how they improve code modularity, flexibility, and maintainability. This experience emphasized the value of clear, single-purpose classes and reduced dependencies, which made our code more organized and manageable.

5.2.2 Practical Application of Polymorphism:

- Polymorphism became an invaluable tool in designing flexible user interactions. Implementing displayMenu() polymorphically across various user roles gave us a practical understanding of how polymorphism can simplify code by allowing a single interface to handle multiple types. This experience taught us the value of designing with an abstract superclass to enforce consistent behavior while allowing role-specific implementations.

5.3. Further Improvement

5.3.1 Polymorphism in Appointment and Inventory Management: If the HMS were extended to support various types of appointments or inventory items, polymorphism could be leveraged to manage these different types interchangeably. For instance, if Appointment had multiple subtypes, methods in AppointmentManager could handle each type polymorphically.

5.3.2 Potential for Explicit Interfaces: If HMS were expanded further, interfaces could be introduced to formalize this segregation. For example, interfaces like IAppointmentService or IInventoryService could define relevant methods for appointment and inventory management. This would allow different implementations to be swapped without modifying the system, aligning closely with ISP.
Future versions could introduce interfaces for IAppointmentService, IInventoryService, or IBillingService to further decouple high-level classes from specific implementations, aligning even more closely with DIP.

5.3.3 Implementing a Database for Better Data Management: To enhance data integrity and support more complex queries, a database could replace text file storage for handling users, appointments, and inventory. Using a database would simplify data management, reduce file handling issues, and make it easier to add relationships between entities.

5.3.4 Improving Data Security with Password Hashing:Currently, user passwords are stored as plain text in files, which poses security risks. Implementing password hashing would enhance security, ensuring that sensitive information is protected in case of data breaches or unauthorized access.