**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  (SC2002/CE2002 CZ2002) | Lab  Group | Signature /Date |
| Wong Yi Jie | SC2002 | SCSF | 14/11/2024 |
| Tan Wee Choon Waylen | SC2002 | SCSF | 14/11/2024 |
| Seow Jing Woon | SC2002 | SCSF | 14/11/2024 |
| Saeng-Nil Natthakan | SC2002 | SCSF | A black text with a white background  Description automatically generated14/11/2024 |
| Gopashish Harikrishnan | SC2002 | SCSF | 14/11/2024 |

Important notes:

1. Name must **EXACTLY MATCH** the one printed on your Matriculation Card.
2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

## Design Consideration

### Overview of MVC Design Pattern

*Reference:* (Jain, 2024)

## In developing our Hospital Management System (HMS), our team has chosen the Model-View-Controller (MVC) design pattern, dividing our application into three primary components: the **Model, View, and Controller.** This architecture supports a modular approach that separates the application's data (Model), user interface (View), and logic (Controller), which enhances both **maintainability and extensibility.**

***Model:*** This component encapsulates the data and core logic. Each entity in our system, such as patient records, is represented through classes that use **encapsulation** principles, with **set** and **get** methods for controlled data access. The model is also responsible for handling operations like creating, updating, or deleting records, keeping these functions centralized and independent of the interface.

The **Model** consists of classes that represent the core business logic, including entities like **User, Appointment**, and **InventoryItem** These models interact with the database or data layer and provide methods for retrieving and updating information. These classes encapsulate the attributes and behaviors of each entity.

***View:*** The view is dedicated to rendering the **user interface and displaying outputs** to users. By keeping all UI-related operations within this component, we maintain a clean separation between data representation and display, which aids in testing and provides flexibility to update the UI without impacting the underlying logic.  
  
***Controller:*** The controller acts as the **application's brain,** orchestrating communication between the Model and the View. It holds the **core logic and functions** that facilitate user interactions, enabling data to flow between the Model and View effectively. This separation also reduces dependencies, as the View can interact with the Model solely through the Controller.

## **Extensibility and Maintainability**

The MVC pattern inherently supports extensibility by allowing us to work on individual components independently. For instance, we can expand our Model to add new data fields or operations without altering the Controller or View. Similarly, we can adjust the UI in the View component without affecting the Model’s data-handling mechanisms.

This separation also aids in maintainability:

* **Code Readability** improves as each component has a defined purpose, making it easier for future developers to locate and understand different parts of the system.
* **Testing** becomes simpler, as each component can be tested in isolation. We can validate the Model’s data functions independently, verify the View’s display functionality, and confirm the Controller’s logic without **overlapping dependencies.**
* **Bug Fixing** becomes easier as we do not have to worry about one portion of the code breaking the other portions of the code’s functionality. Isolating the issue is made easier thanks to this.

## **Assumptions Made**

1. **Persistence**: Data is serialized to files (appointments.ser, inventory.ser). A database system can replace this without changing the controllers.
2. **User Roles**: Extensibility is key. New roles can be added by extending User.
3. **Error Handling**: The user input is validated (e.g., ensuring that a number is entered where expected), which helps prevent crashes or incorrect behavior.

## **Reflection on Design Patterns**

The MVC pattern brings a **clear separation of concerns** that improves software quality and simplifies updates. Our HMS design is more robust and adaptable as a result, supporting our goal of providing a system that is easy to maintain, extend, and adapt to future requirements. Overall, MVC has enabled us to construct a system where we can manage complexity effectively, allowing each component to evolve independently while still contributing to a cohesive and functional whole.

## **SOLID Principles:**

### **1. Single Responsibility Principle (SRP):**

Each class in the MVC architecture has a single responsibility:

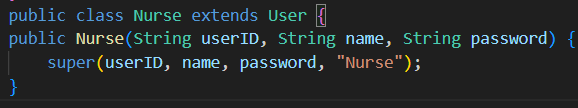
* **Model** handles data and business logic.
* **View** is responsible for displaying information.
* **Controller** handles user inputs, processes them, and updates the model and view.

**Example**: In the **AdminView** class, the methods **displayMenu(), displayStaffList() and getUserChoice()** all handle specific tasks related to displaying menus and user input, without involving logic or data processing.

### **2. Open-Closed Principle (OCP)**

**Definition**: Classes should be open for extension but closed for modification.

**Application in Code**:  
By using inheritance and polymorphism, we ensure that new features (e.g., new user roles or appointment types) can be added without modifying existing classes.

**Example**: The **User** class is designed as a base class for all user roles. New user roles (e.g., **Nurse**) can extend **User** without altering existing implementations.

### **3. Liskov Substitution Principle (LSP)**

Our code adheres to the **Liskov Substitution Principle (LSP)** because the **Administrator** class, as a subclass of **User**, inherits all the expected behaviors of the **User** class, such as managing user information (ID, name, role, etc.) without introducing any inconsistencies or exceptions. Subclasses like **Administrator** can be substituted for **User** in any part of the program, and the expected functionality remains intact. This ensures that the program can operate correctly regardless of whether it uses a **User** or a subclass like **Administrator**.

Eg.

### **4. Interface Segregation Principle (ISP)**

#### **Single Responsibility Focus**:

* The **AdminController** class focuses on the specific responsibilities of an administrator in a hospital management system. It deals with managing staff, appointments, and inventory. It does not implement generic or unrelated functionality.

#### **Separation of Concerns**:

* The **AdminController** class relies on separate classes like **AdminView**, **Doctor**, **User**, and **InventoryItem** to handle the details of displaying information and representing data. This keeps the responsibilities of **AdminController** focused on coordinating between the model (data) and view (UI), rather than having it directly handle or implement logic for each aspect.

#### **Small Interfaces**:

* While the **AdminController** class itself is large due to the number of features it handles, it interacts with smaller, more focused interfaces. For instance, **AdminView** is responsible for handling all user interactions, such as displaying menus and getting user input. The controller does not need to implement any UI-related behavior directly.
* Methods like **loadUsers(), saveUsers(), viewDoctorSchedulesMenu()**, etc., are highly focused and encapsulate a single action. For example, **addInventoryItem(), updateInventoryItem(),** etc., are distinct actions related to inventory management, without trying to mix those actions with unrelated responsibilities.

#### **Minimal Dependency on Specifics**:

* The **AdminController** depends on interfaces (or abstract classes) like **User**, **Doctor**, and **InventoryItem**, rather than concrete implementations, which allows for flexibility and keeps the controller from becoming tightly coupled to specific implementations. This makes it easier to extend or modify the functionality later without violating ISP.

### **5. Dependency Inversion Principle (DIP)**

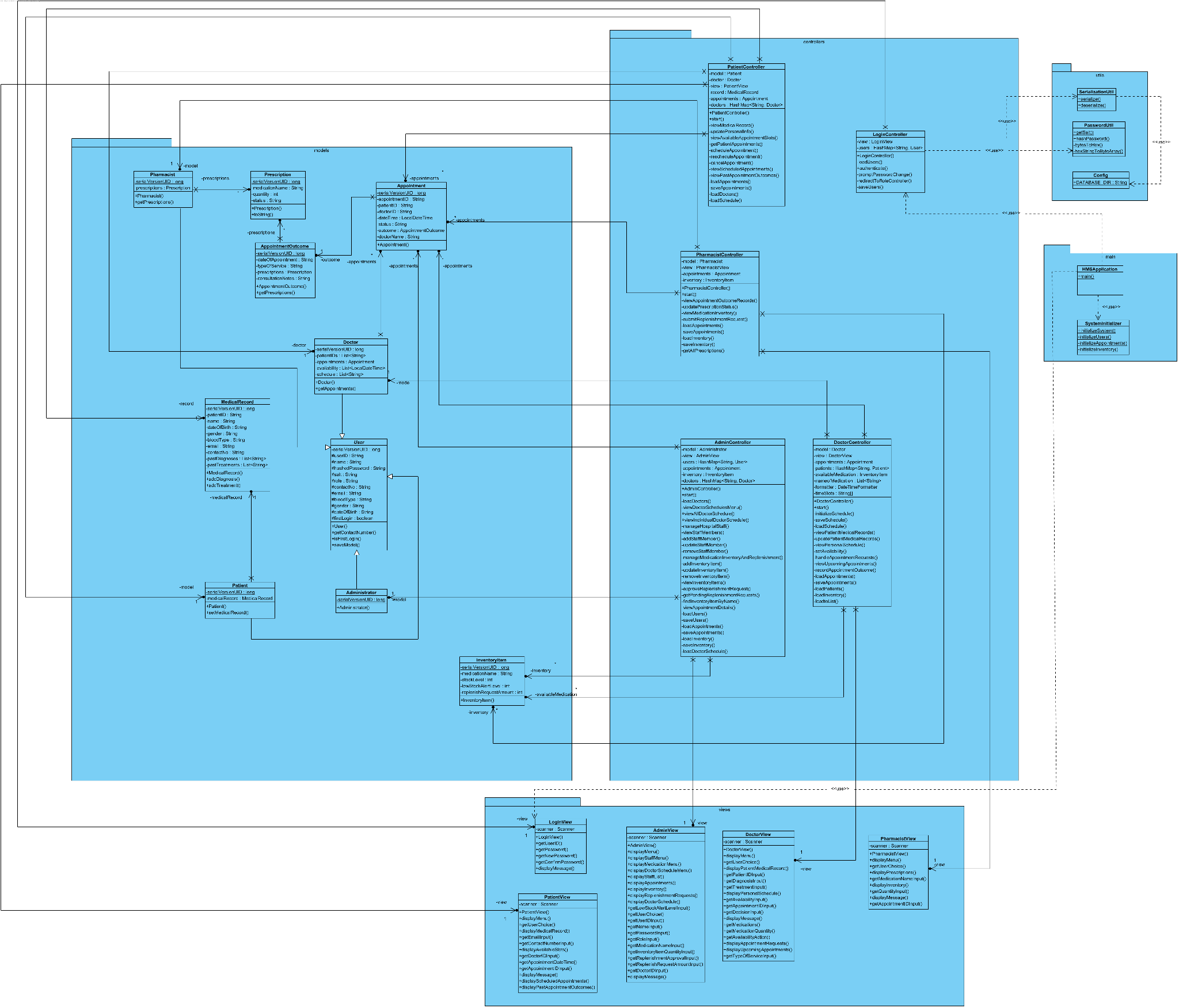
**Example: AdminView** class does not directly rely on lower-level details like the implementation of data storage or business logic. Instead, it interacts with models such as **User**, **Appointment**, and **InventoryItem**, which abstract the business logic and data handling. This separation ensures that the view layer is independent of the underlying logic, following the DIP’s guideline that high-level modules (like **AdminView**) should not depend on low-level modules (like data handling and business logic).

**Example**:

The class is abstract in terms of the data it works with (it could work with any **List<User>**, **List<Appointment>, or List<InventoryItem>**), which can be easily extended or replaced with different implementations in the future without altering this class.

# UML Class Diagram

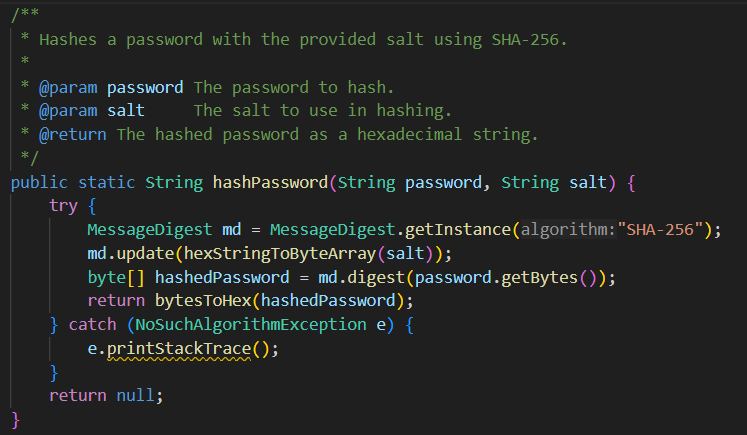
[Link to Class Diagram](https://github.com/jwistired/SC2002_Hospital_Managament_System/blob/main_complete/Report/Class%20Diagram.png)



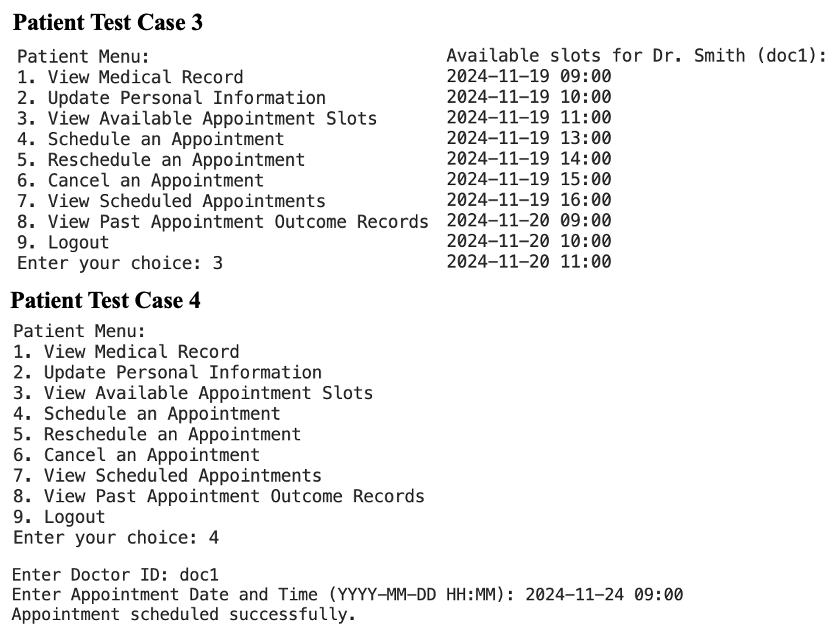
Additional Features/Functionality

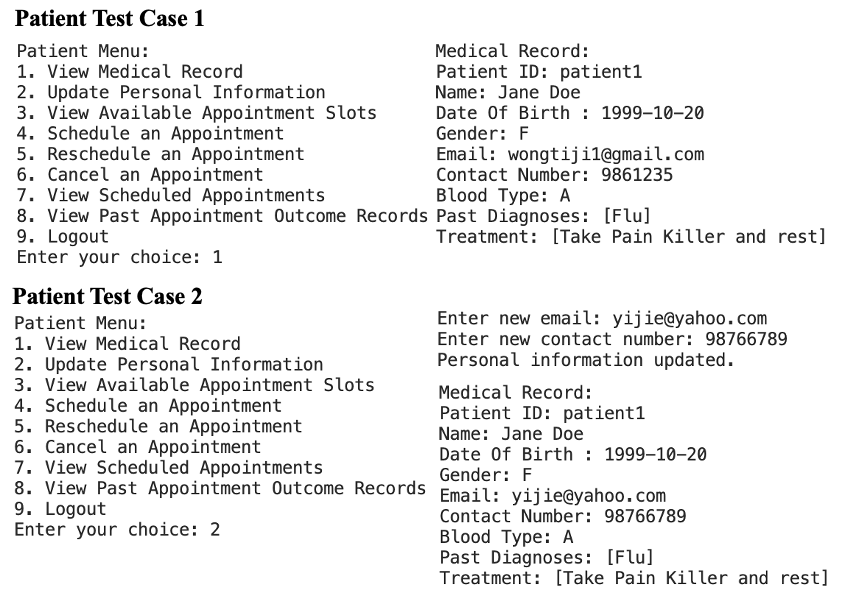
1. **Password Hashing** - For security purposes, we implemented password hashing using SHA-256. In the event of a data breach, having hashed passwords helps prevent hackers from getting their hands on sensitive data such as personal details.

**Example**

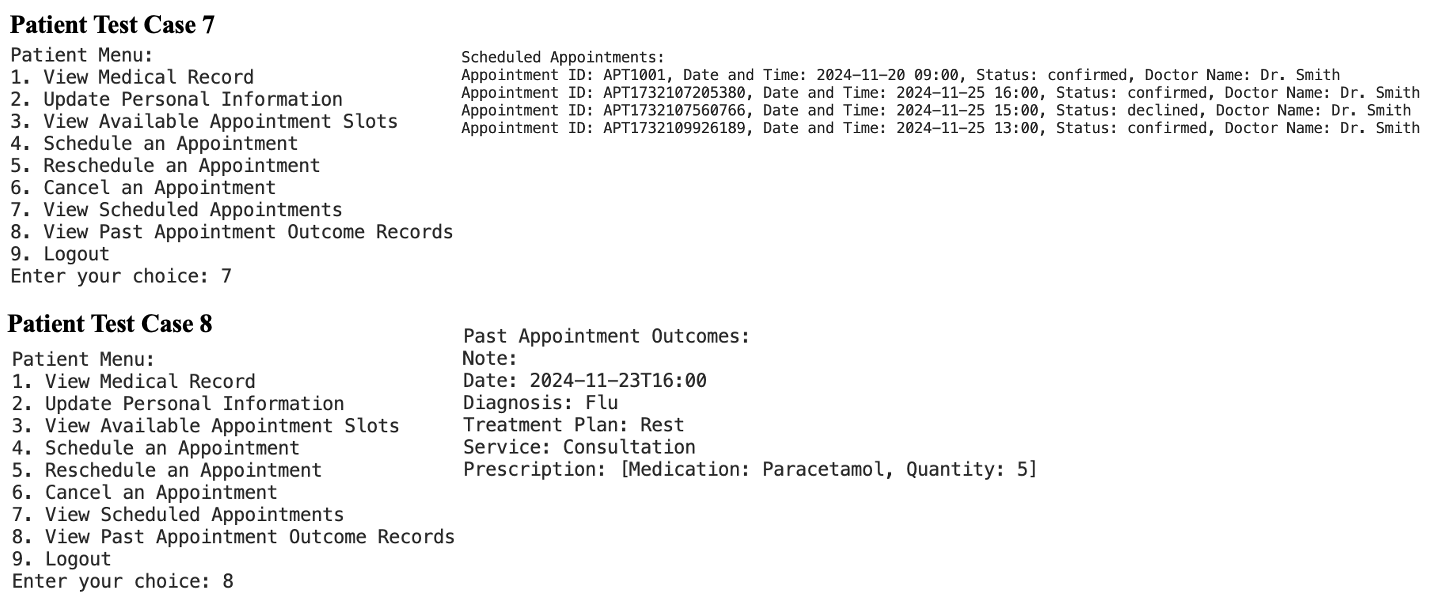


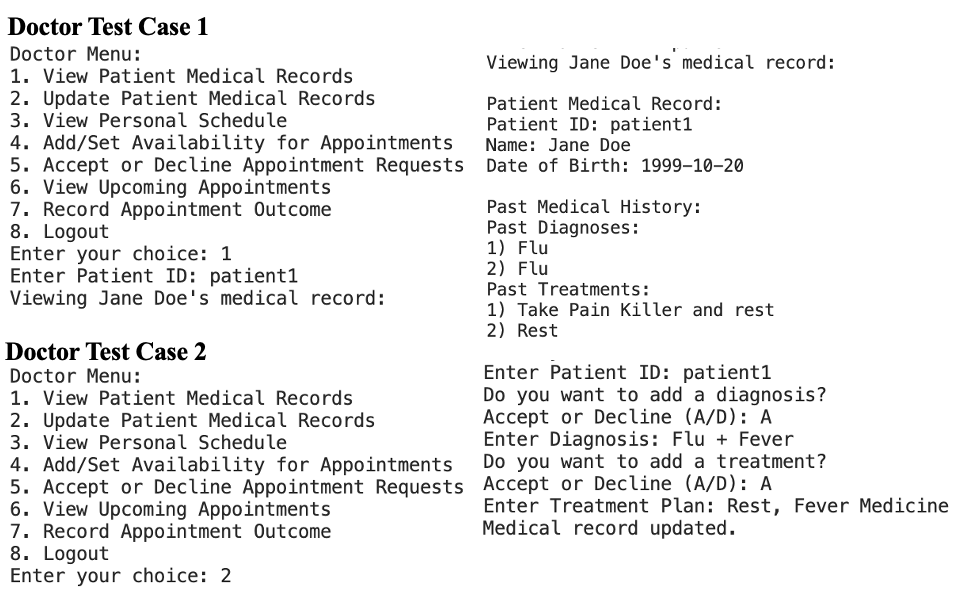
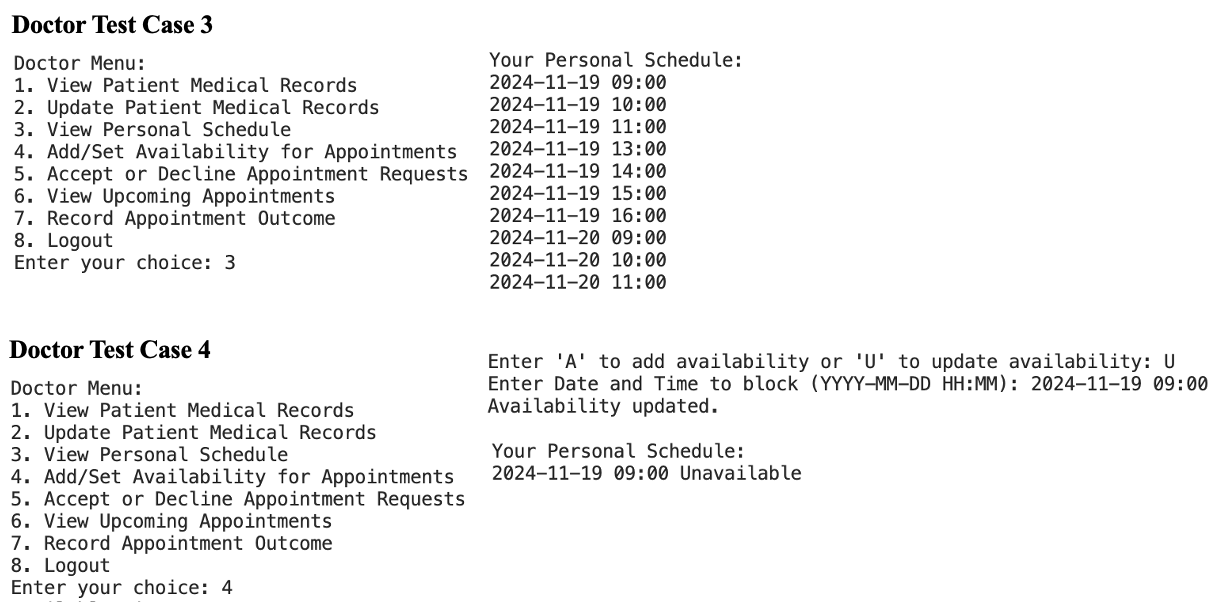
1. **Serialization/Deserialization**: The **SerializationUtil** class provides methods to save and load objects to/from files. This ensures that system data (e.g., medical records, appointment schedules) persists across application restarts. It uses Java's **ObjectOutputStream** and **ObjectInputStream** to serialize and deserialize data, facilitating object storage in the specified database director
2. Our classes are more adaptable, allowing it to **store any type of list data** **rather than being restricted to specific types**. This would enable the class to support a wide range of data types in different scenarios.
   1. For example, the **prescriptions** field in the class could be generalized to store **List<T>,** where **T** could be **Prescription, User, Appointment, InventoryItem**, or any other relevant class.

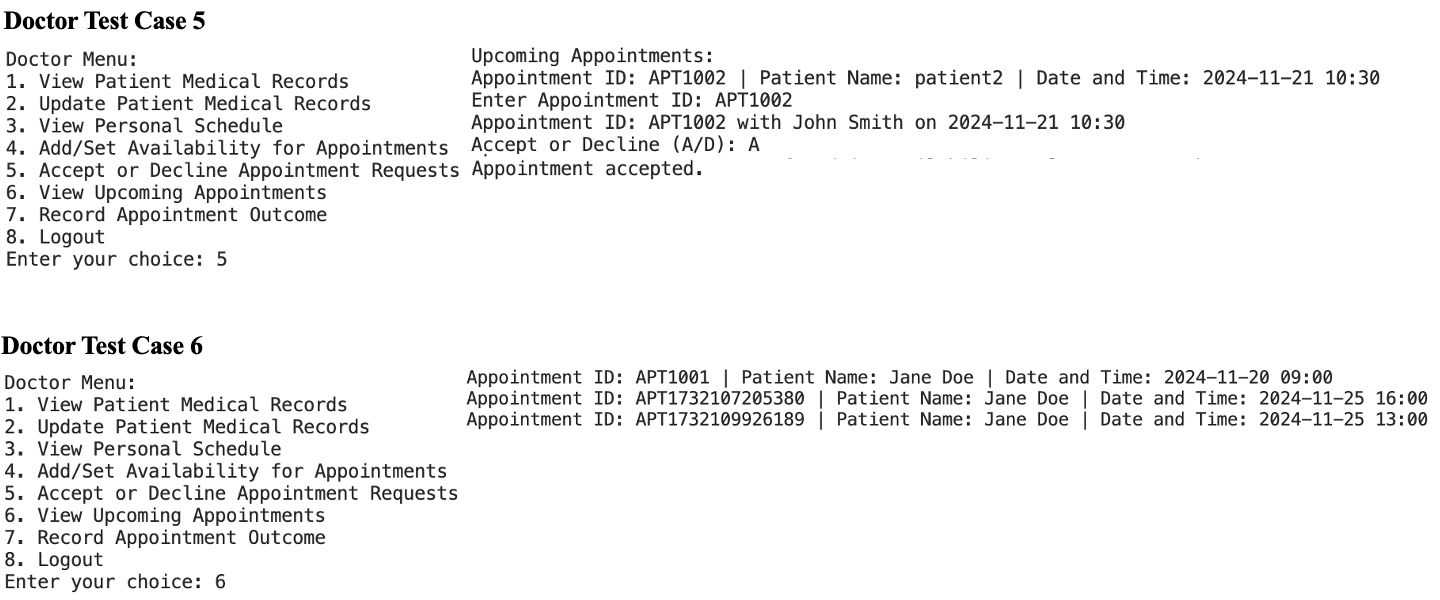
**Test Cases**

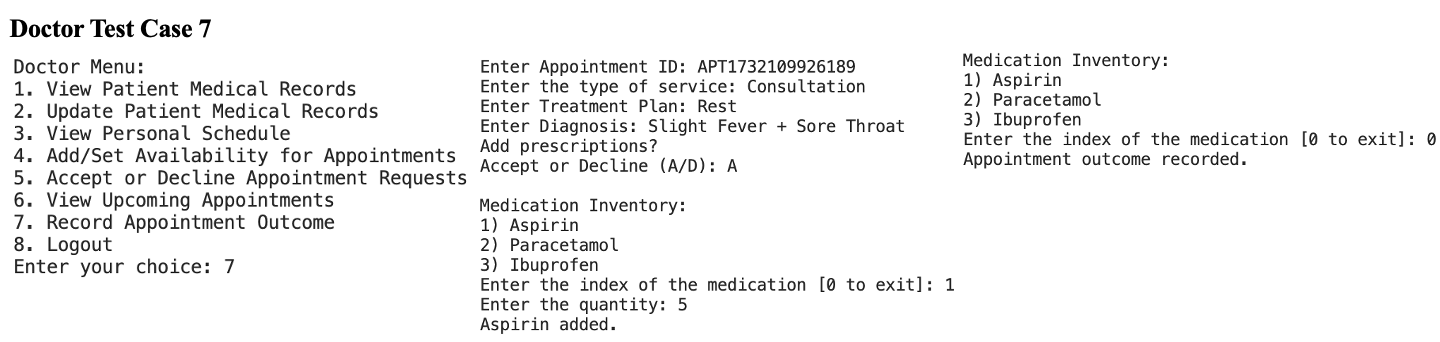
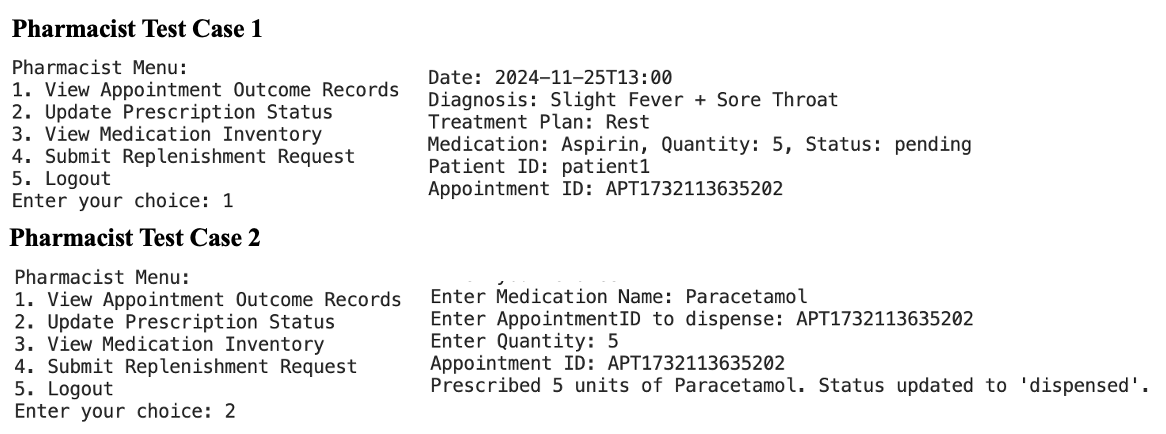
****

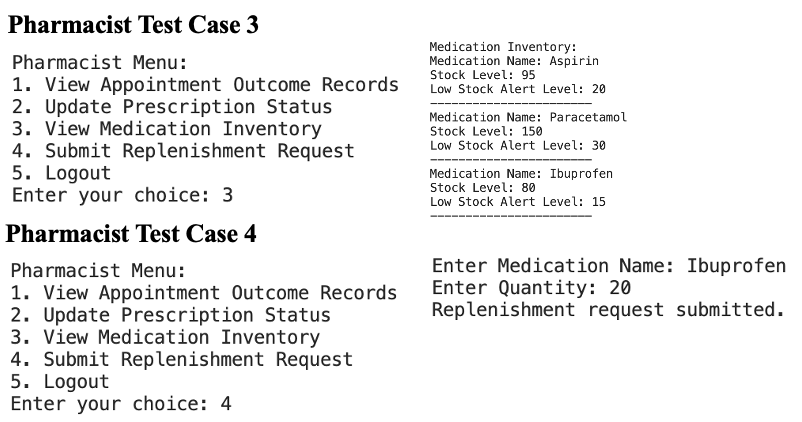
****

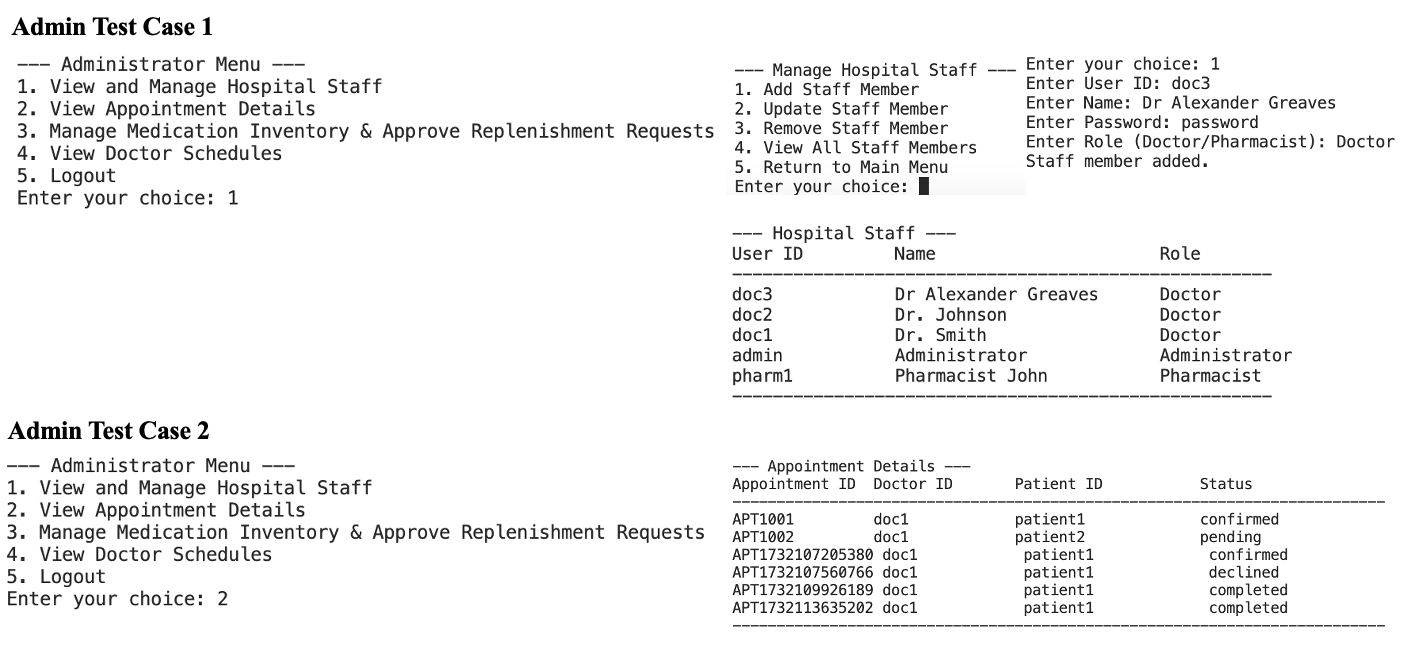


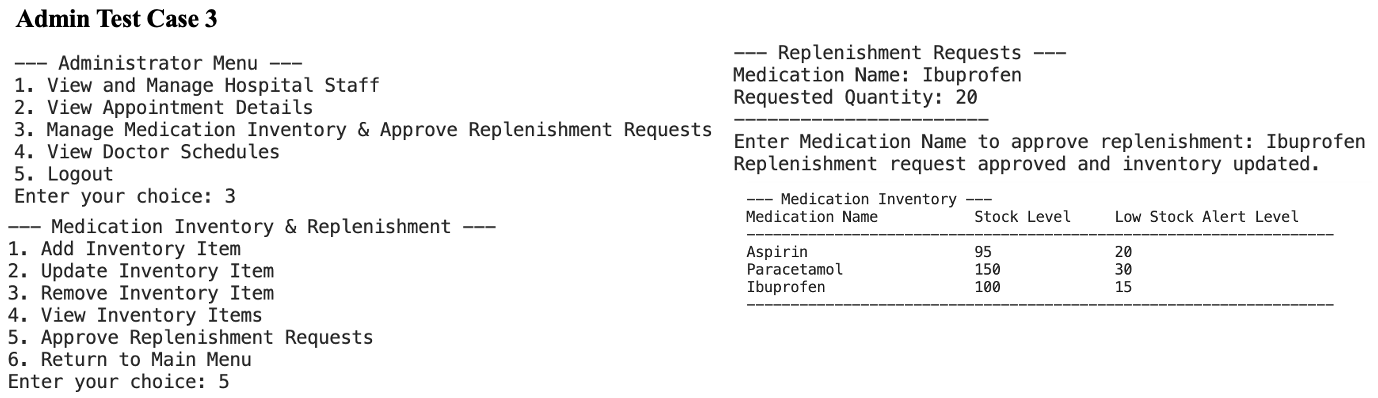


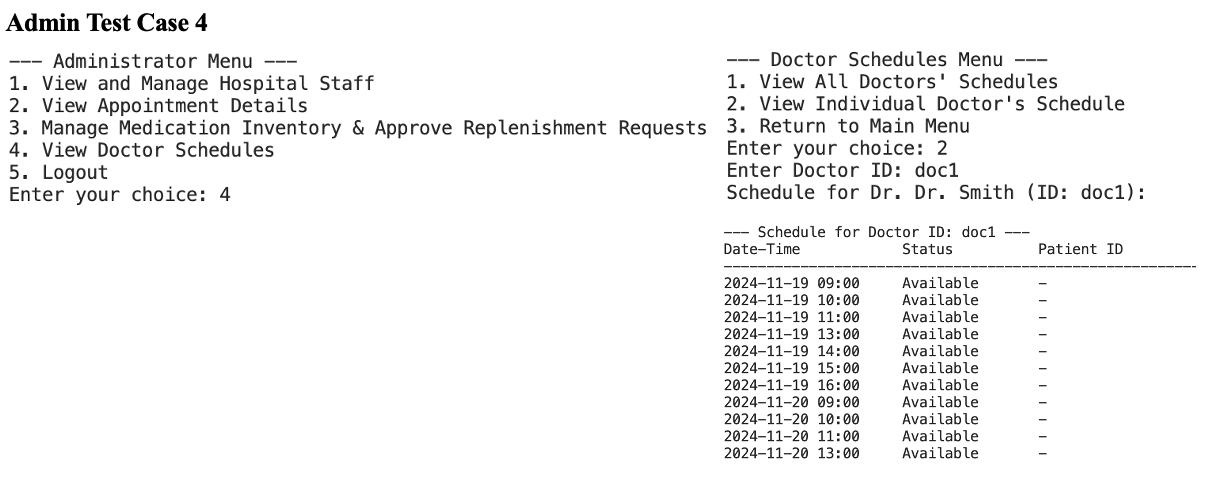


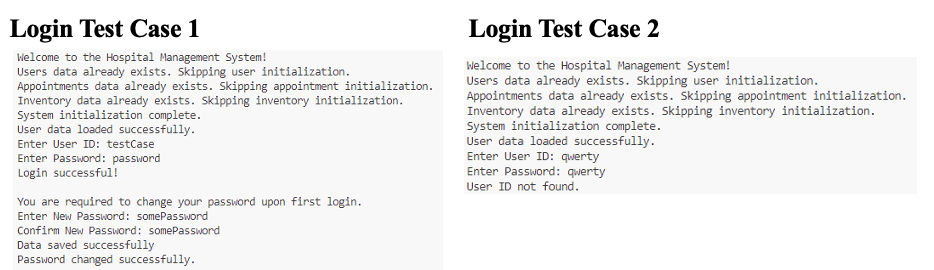












**Team Reflection:**

At the start of the project, we decided that each team member would focus on coding their assigned role and then merge everything later. We began by designing a UML diagram, but soon realized it was all over the place and lacked a clear structure. After reaching out to Prof. Marcus, he suggested a few software architecture options for us to explore. We ultimately decided to go with MVC because it seemed to fit our needs perfectly.

As we began coding, we made sure to follow both the MVC and SOLID principles. This added a level of complexity we weren’t used to. Instead of writing large blocks of code in a single function, we worked hard to minimize dependencies between functions, aiming to keep the software flexible and extensible.

The challenges we faced during the project were tough, but they taught us a lot and helped us grow as a team. We listed some of the biggest challenges we have faced:

1. **Availability:**

Coordinating our schedules was a big struggle. With everyone working on different parts of the project, we often found ourselves falling behind because we couldn’t meet up as often as needed. There were times when not having real-time communication led to confusion about how certain features should work, which slowed us down. We learnt the hard way that consistent meetings were the key to ensure that the project stayed on track.

1. **Merging Process:**

We had agreed to code our individual roles and planned to merge our work later. However the actual merging process turned out to be much more complicated than we had expected. We ran into a number of issues, like mismatched data between views and null objects, which led to functionality errors. Fixing these problems required a lot of patience and communication, as we had to understand each other’s code to pinpoint the issues. This experience really taught us the importance of planning for integration earlier and ensuring our work meshes well from the start.

1. **Knowledge Beyond the Course:**A lot of the project required knowledge that went beyond what we had learned in the course. While we were familiar with the basics of programming, applying the MVC framework and SOLID principles in a real-world setting was a challenge. We had to dig deeper into these advanced concepts and spent time researching to fully understand how to use them. Additionally, structuring our code for future flexibility and reducing dependencies was harder than we had anticipated, pushing us to learn new things on the fly. This experience helped us grow and pushed us to go beyond the course material.

All in all, these challenges made the project more complicated, but they also provided invaluable experience. Working through them as a team taught us a lot about problem-solving, collaboration, and applying what we’ve learned in real-world situations. It’s a lesson we will take with us into future projects.

*Link to Github :* <https://github.com/jwistired/SC2002_Hospital_Managament_System>

**References**

1. Jain, S. (2024, October 15). *Software Design Patterns Tutorial*. GeeksforGeeks. Retrieved November 17, 2024, from <https://www.geeksforgeeks.org/software-design-patterns/?ref=shm>
2. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns*. Pearson India.
3. Martin, R. (2016). *Clean code*. Addison-Wesley Professional.