Concordia University

SOEN 6461 Software Design Methodologies

Assignment 1

Group members: Duy Thanh Phan – 40269832
Omnia Alam – 40261762

1. Table of Contents

2. Introduction

The Intelligent Hospital Management System (IHMS) is a complex and modern solution designed to streamline healthcare operations, enhance patient care, and optimize resource management within healthcare facilities. This system integrates diverse healthcare processes, including user registrations, appointment scheduling, medical history management, and billing system, into a unified and efficient platform. IHMS leverages intelligent solutions to ensure seamless interactions among patients, doctors, administrators, and pharmacists, ultimately improving the healthcare experience for all stakeholders involved.

This design document serves as a roadmap for developing IHMS, addressing the assignment tasks while meeting the technical constraints. The tasks detailed in this document encompass the creation and implementation of crucial components, design patterns, and structured approaches necessary for achieving a robust, adaptable Intelligent Hospital Management System.

Each assignment task requested in the original assignment will be addressed by the sections below, ensuring that the resulting system aligns with the defined goals, fulfills the needs of various user roles, adheres to technical constraints, and maintains scalability and security. Through these assignments, we aim to create a well-structured, efficient software system that will be an invaluable experience in building large-scale enterprise systems in industry work.

3. Object-Oriented Design Proposal

The IHMS follows an object-oriented design approach that is modular and scalable. Object-oriented programming allows us to model real-world entities as objects, each encapsulating its state and behavior, creating a clear representation of the IHMS components and their interactions. Below is a summary of key design principles that we adopted:

Encapsulation: Class attributes are private, and access to them will be controlled through public methods, enhancing security and preventing unauthorized access.

Inheritance: Utilized to create a hierarchy of classes, providing a clear structure for related entities.

Polymorphism: This allows objects to be treated as instances of their parent class, enhancing flexibility and enabling dynamic method invocation based on the actual type.

DRY (Don't Repeat Yourself): We reduce code repetition and promote using reusable functions and classes.

SOLID Principles: Using these principles helps us to achieve the requested design details in the assignment such as:

- Maintainability: By adhering to the Single Responsibility Principle (SRP), each IHMS component only serves a specific purpose, making the system easier to understand and maintain. In the design, we created classes with clear roles and responsibilities.
- Flexibility and Extensibility: Following the Open/Closed Principle (OCP) allows new features or functionalities to be added to IHMS without modifying existing code, ensuring system adaptability to changing healthcare requirements.
- Interchangeability and Reusability: Liskov Substitution Principle (LSP) promotes the seamless substitution of IHMS components, allowing for interchangeable usage of different modules. This is expressed via the adoption of the Stable Mediatory Structures with the design pattern Adapter.

- Interface Clarity and Focus: Implementing the Interface Segregation Principle (ISP) results in interfaces customized to specific requirements within IHMS, ensuring clarity, focused functionality.
- Reduced Coupling and Enhanced Maintainability: Adopting the Dependency Inversion Principle (DIP) reduces direct dependencies between IHMS components, promoting modularity, scalability of the system. This is expressed via various abstractions implemented in our design.

The system will be structured into various classes and interfaces, representing different entities such as Patient, Doctor, Appointment, MedicalHistory, Notification, and more. Each class will encapsulate relevant attributes and methods, promoting code reusability, and adhering to the principles of abstraction to hide complex implementation details.

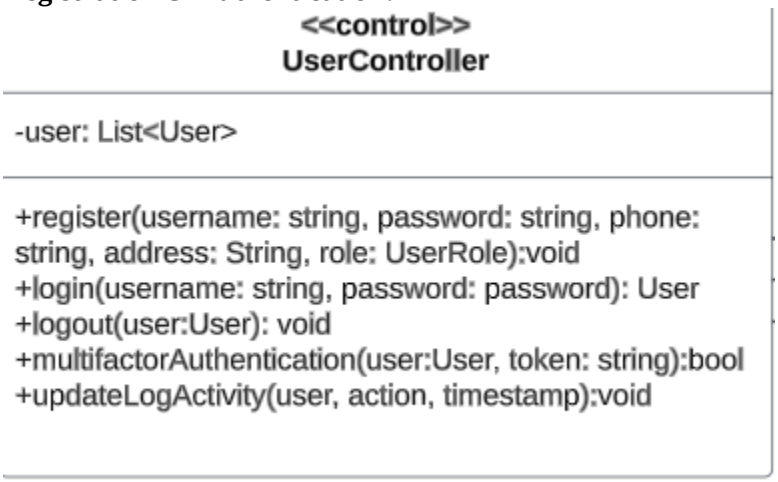
4. Class Diagram

Please see the page below for the detailed class diagram.

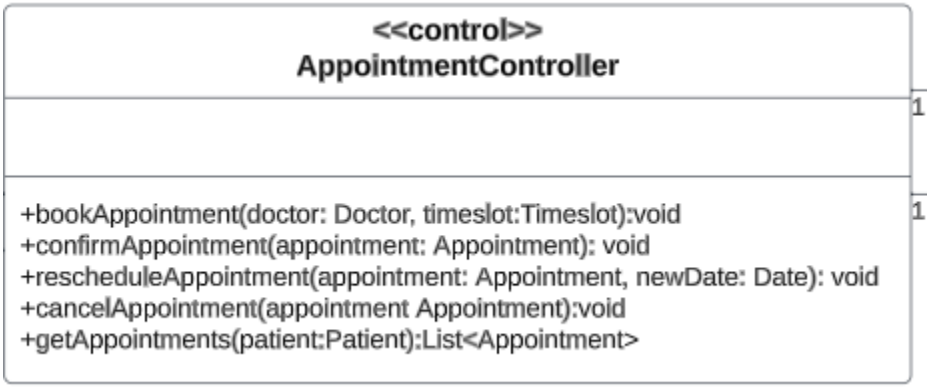
5. Controller Classes Design

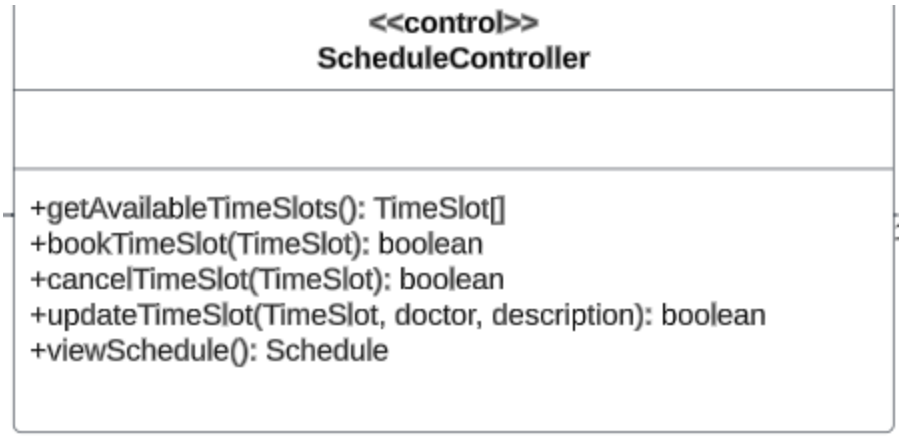
As shown in the diagram each entity class has a specific controller to reach with the UI (User Interface) and data classes. By separating the logic that handles user input and coordinates actions from the logic that deals with data and business rules, controllers make the codebase more modular, maintainable, and testable. The diagram in designed to o handle new features or modify existing controllers to adapt to changing requirements without affecting other parts of the application.

Registration & Authentication:

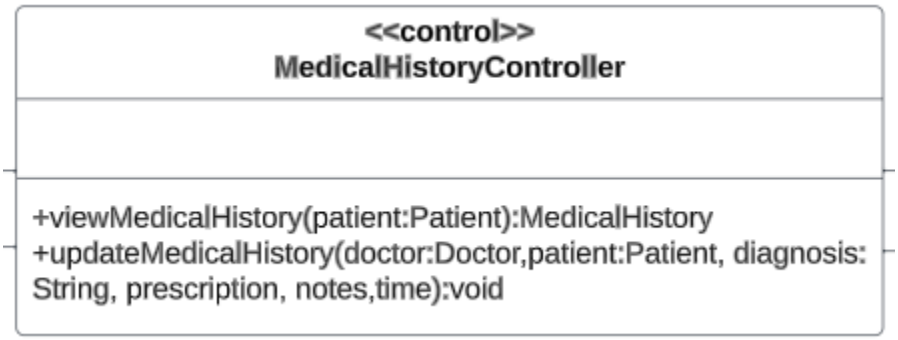


Appointment Booking & Schedule Management:

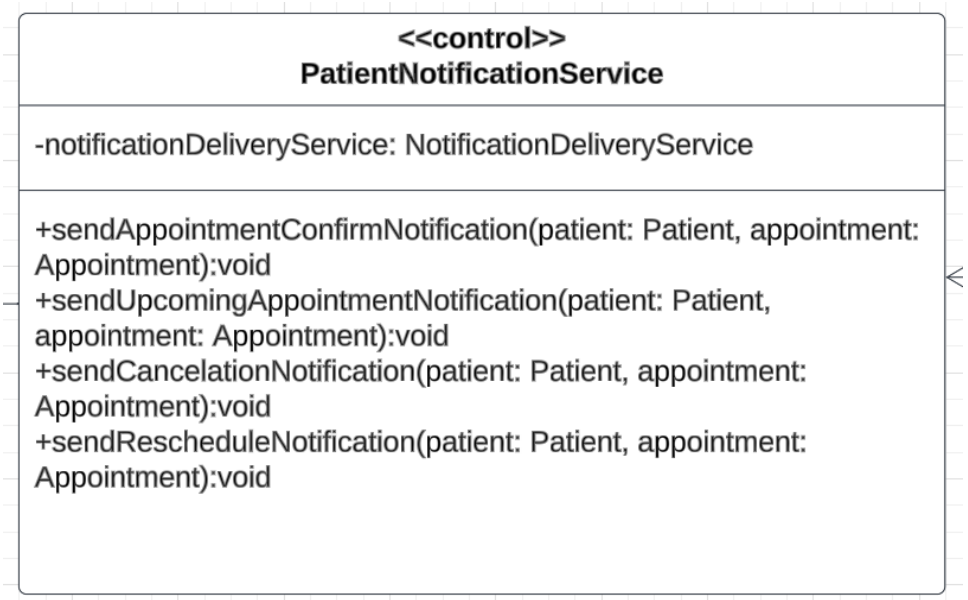




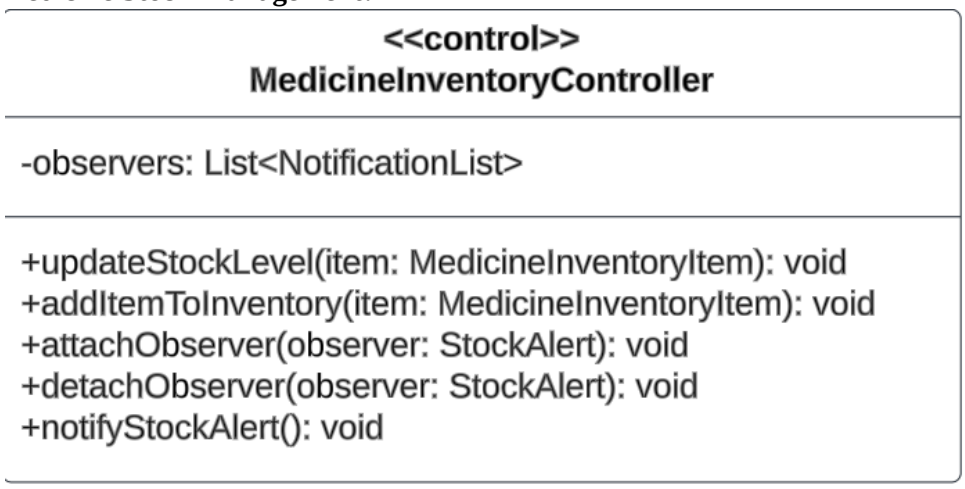
Medical History Maintenance:



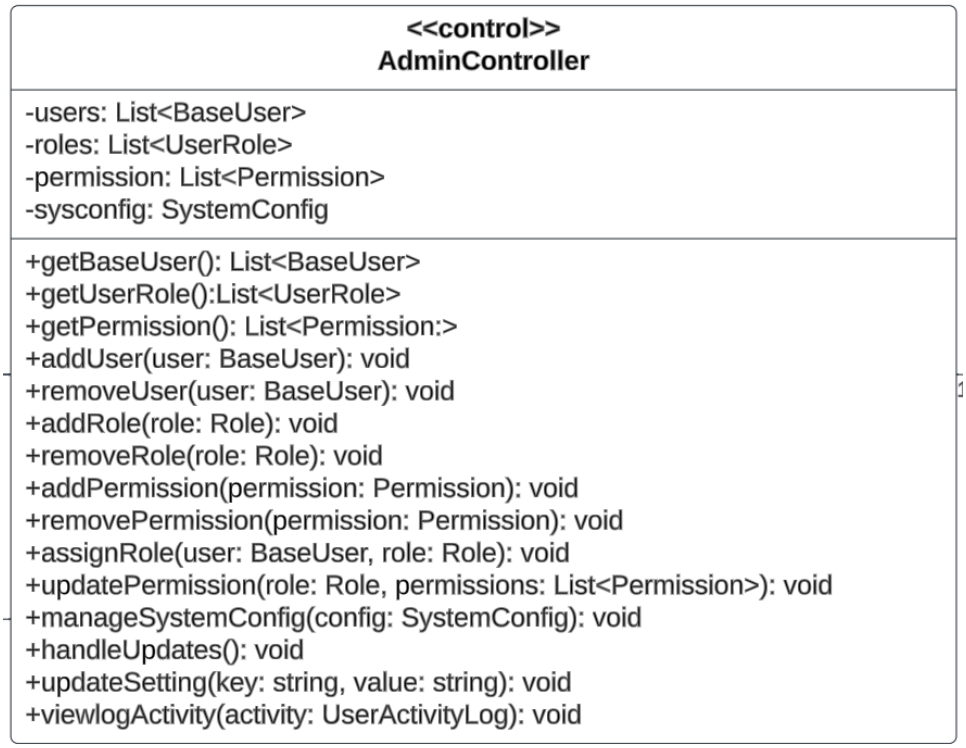
Notifications & Alerts:



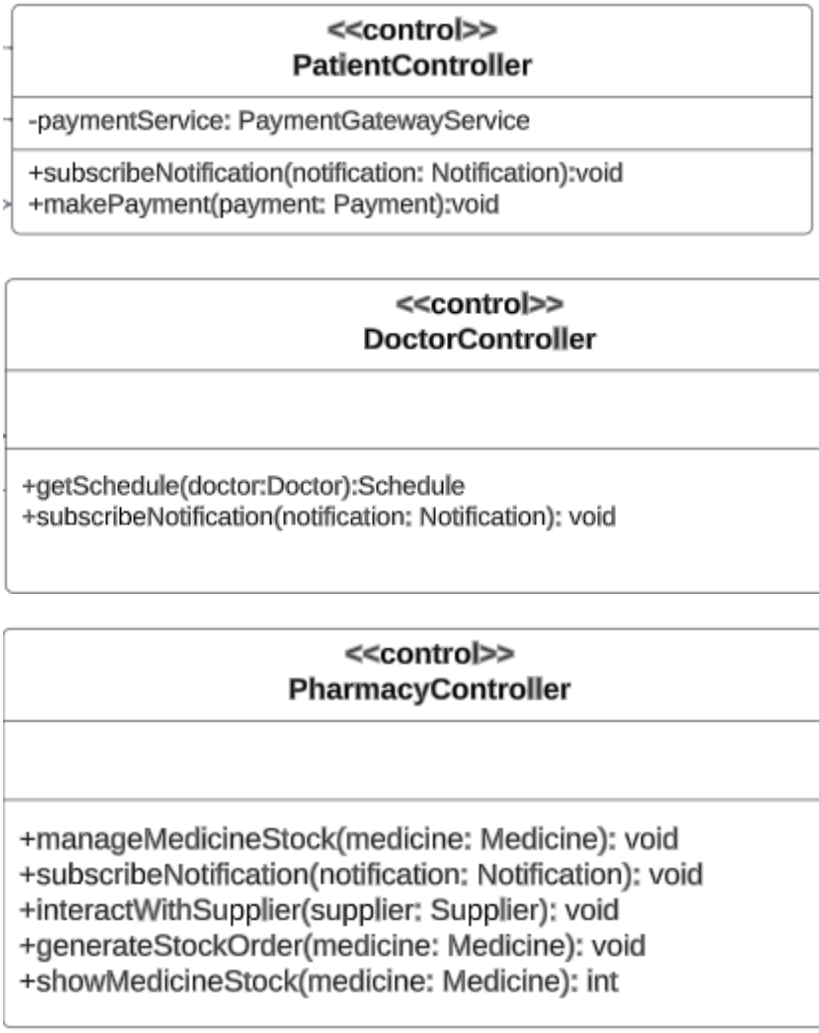
Medicine Stock Management:



System Configuration & Management:



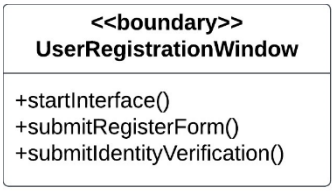
Different user Controllers:

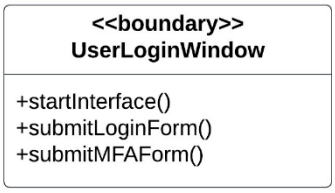


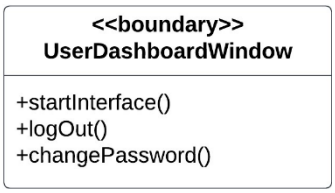
6. Boundary Classes Design

The boundary classes for each detailed workflow will be listed below:

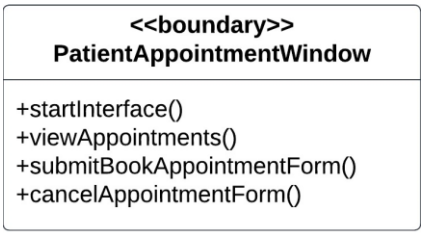
Registration & Authentication:

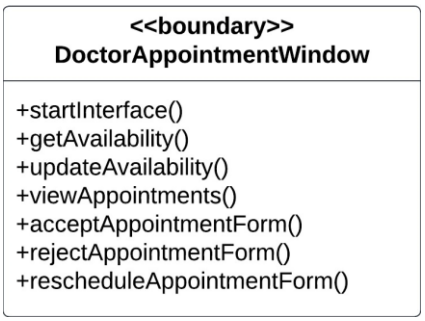


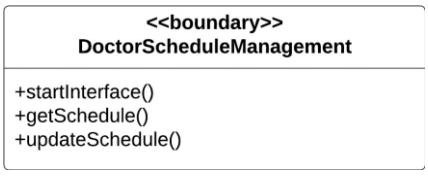




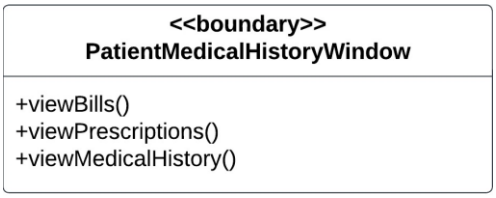
Appointment Booking & Management:

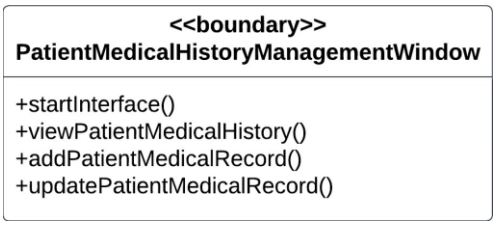




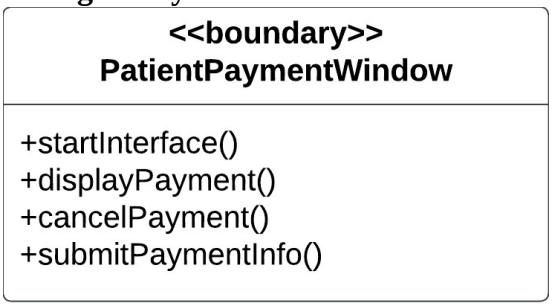


Medical History Maintenance:

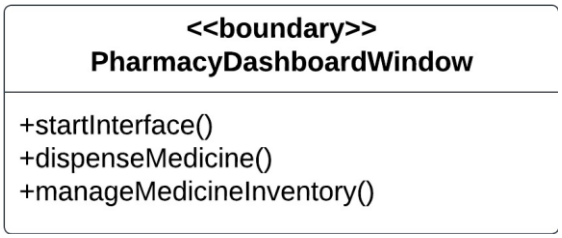




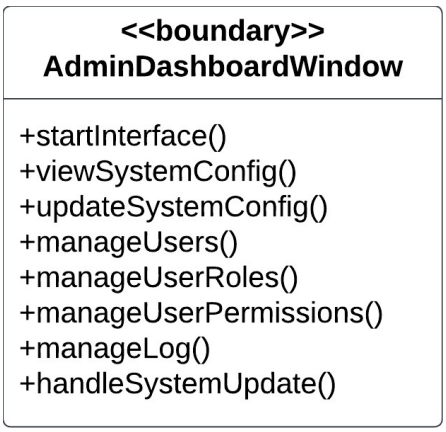
Billing & Payments:



Medicine Stock Management:



System Configuration & Management:



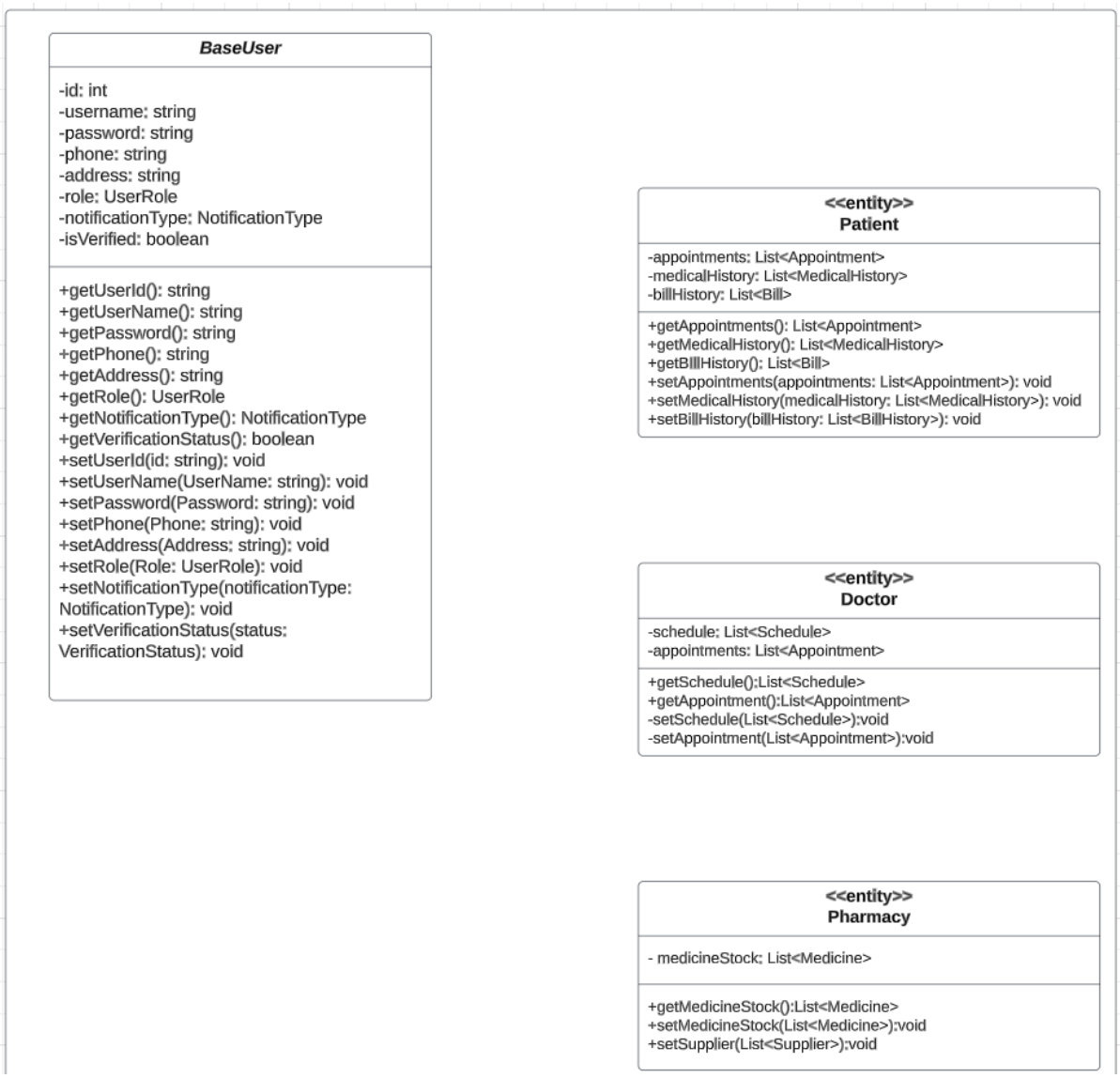
7. Entity Classes Design

We have designed our class diagram using MVC pattern where we separated all our data class from the business function. The entity class only contains the attributes and functions related to the attributes.

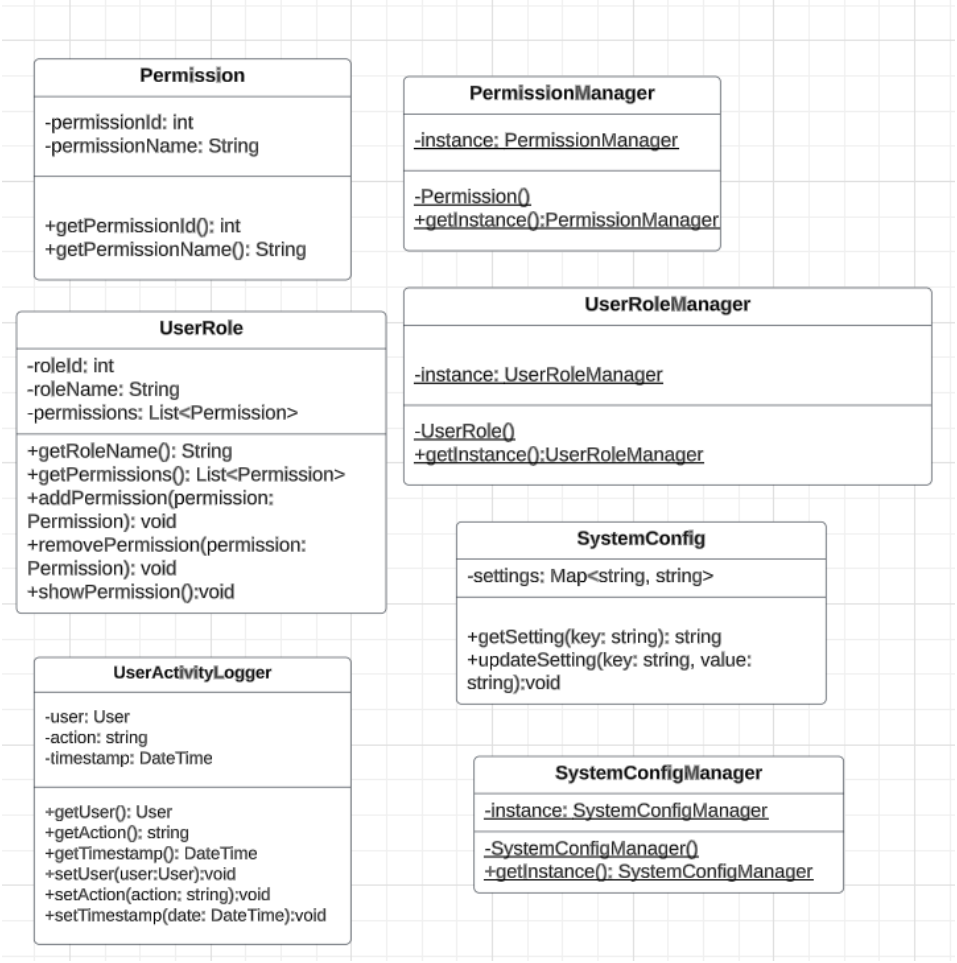
All the attributes are created as private and to access the data we have added getter, setters so that other classes can use the use information but without manipulating the entity structure.

Each of the entities has their own controller class to assign business logic by accessing it's or other entities attributes.

User: Patient, Doctor, Pharmacy Entities

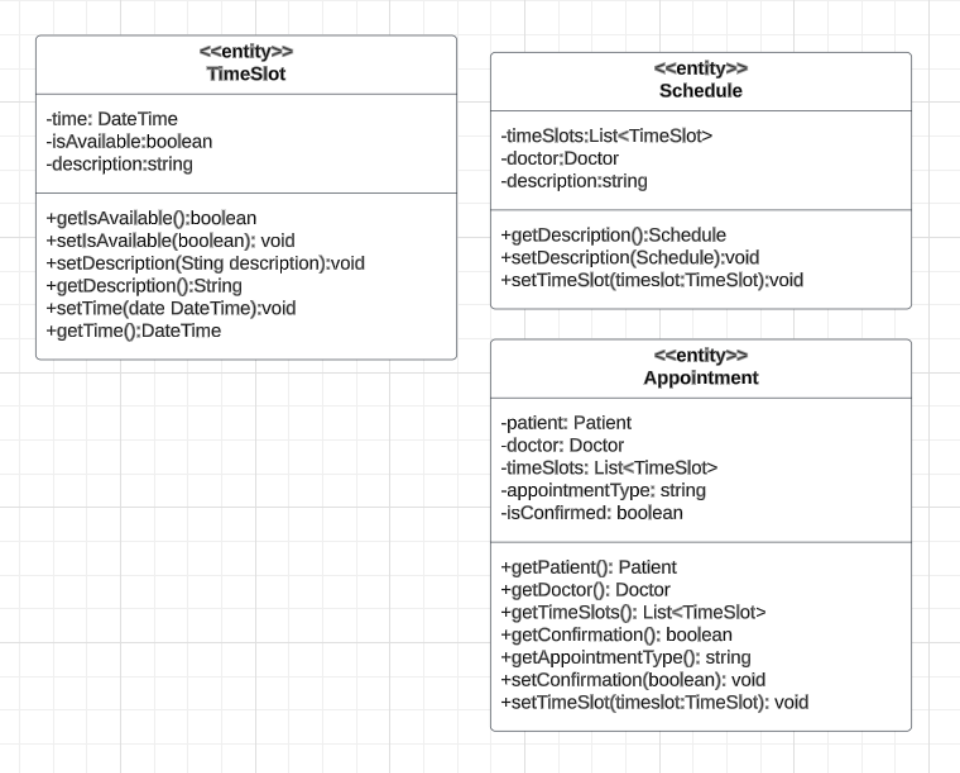


System Configuration & Management:



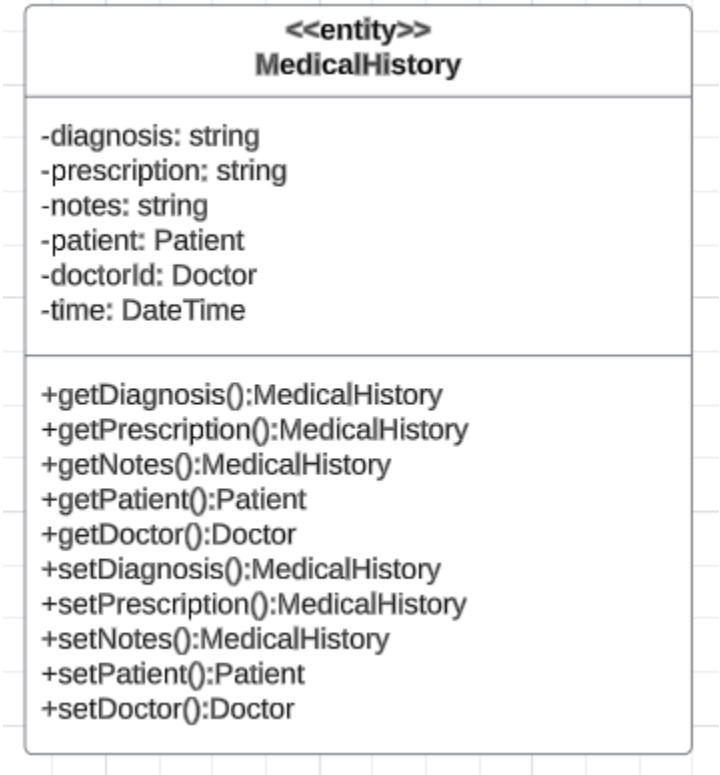
We wanted to ensure that there is only one instance of the system configuration and user roles data in our application. This is useful for maintaining consistency and avoiding conflicts when multiple parts of the application need to read or modify this data.

Appointment & Doctor Schedule:

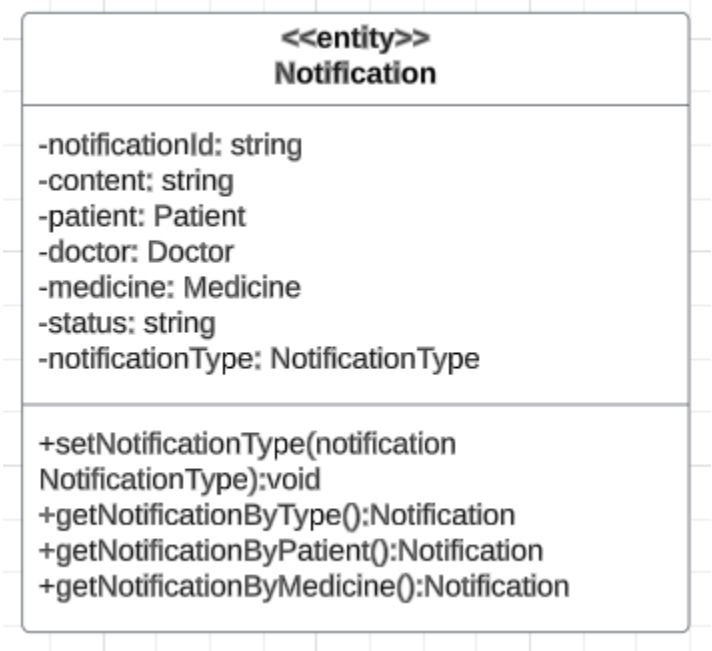


We used one timeslot data class so that the doctors cannot overbook themselves. So, the timeslot as shown in the diagram is used by both appointment class and as schedule class of the doctor. Timeslot class has the attributes that checks if that specific timeslot is available or not.

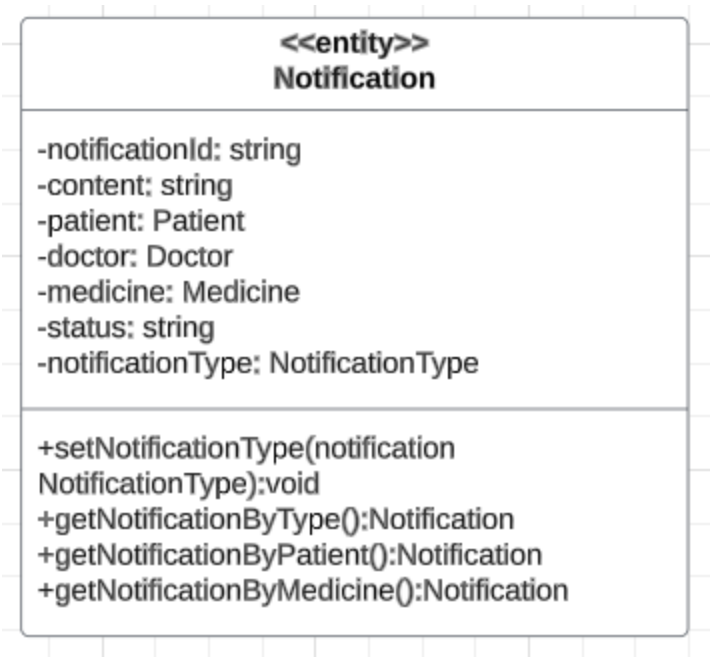
Medical History:



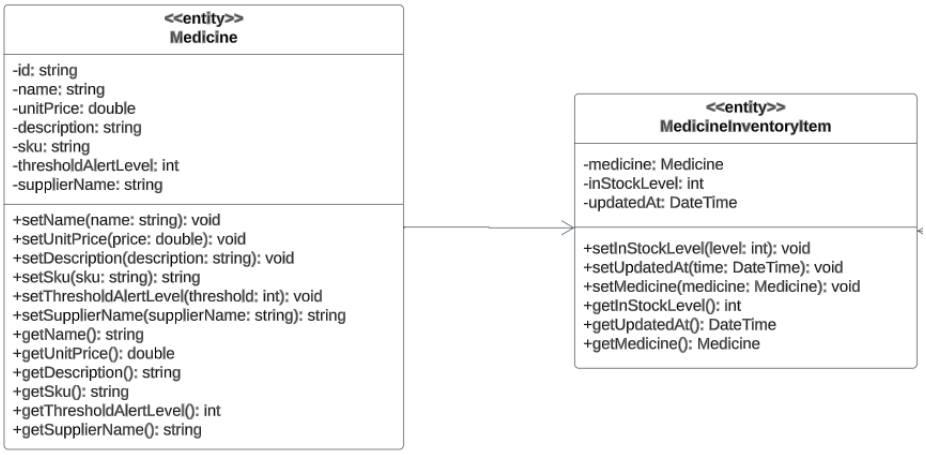
Notifications & Alerts:



We are keeping a history of the notification. So, the we can check the notification status or different types of notification by users.

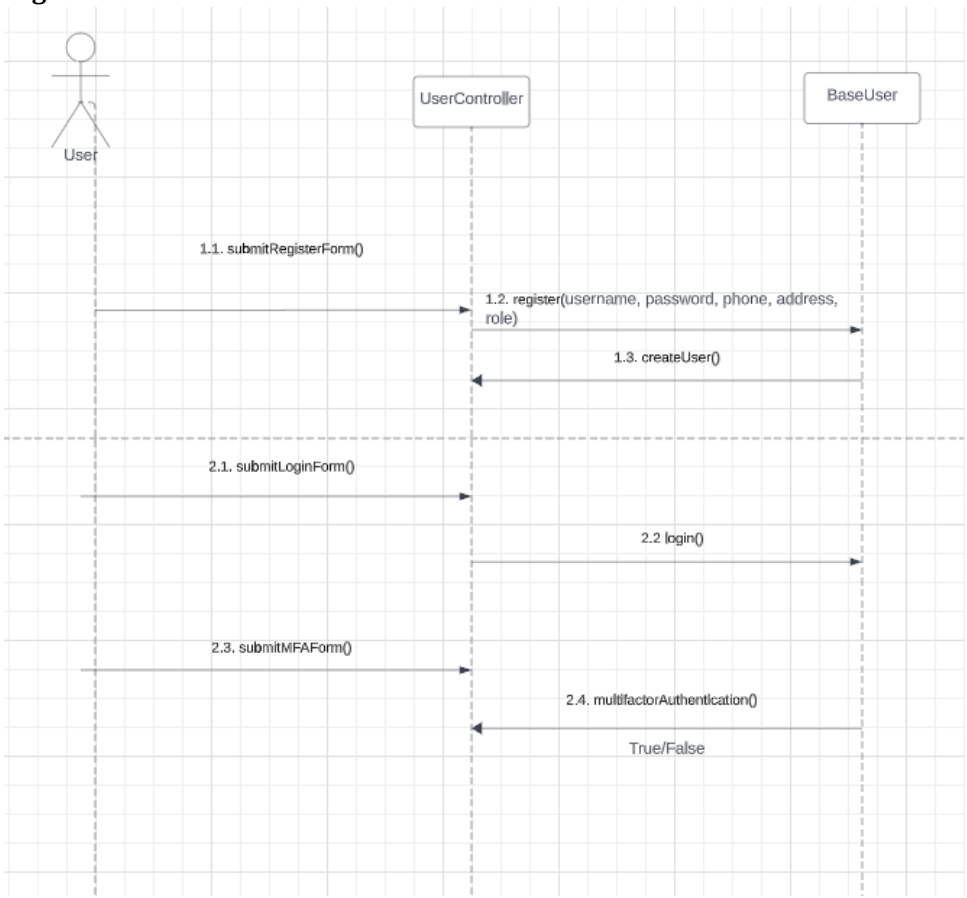


Medicine Stock Management:

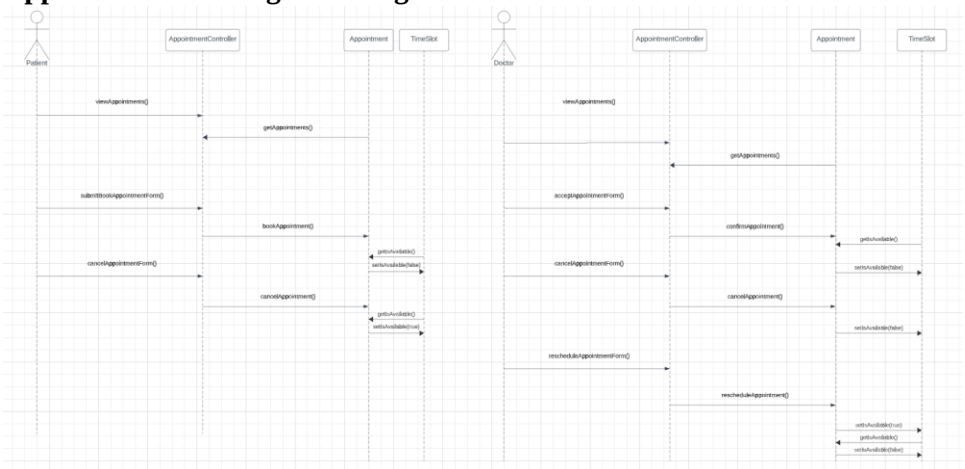


8. Responsibility Definitions

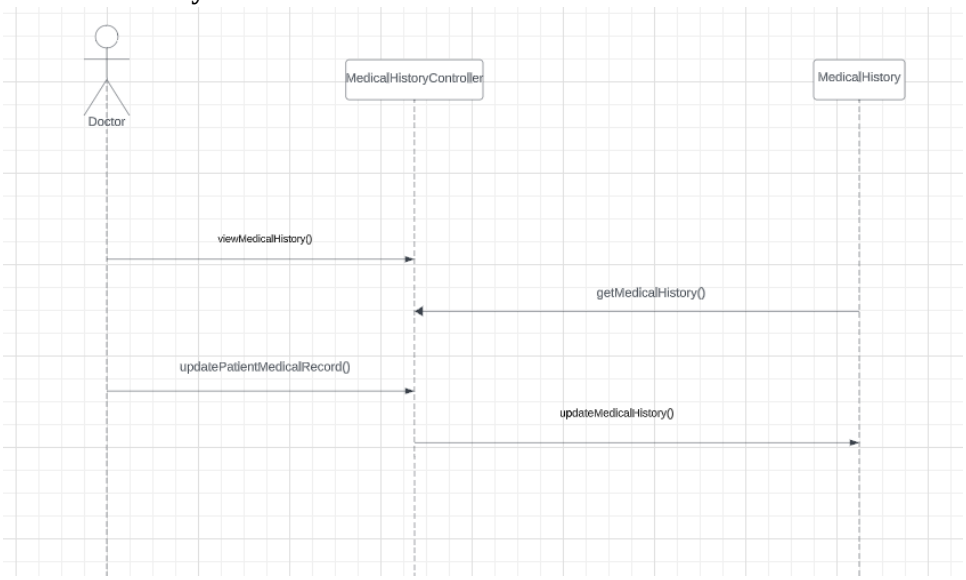
Registration & Authentication:



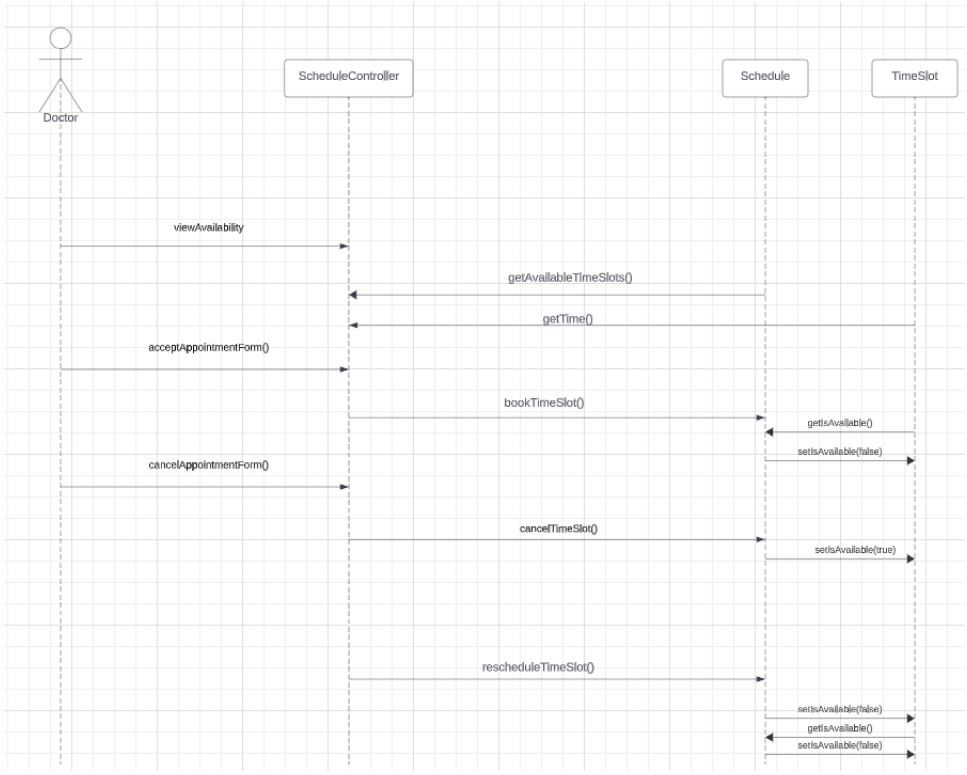
Appointment Booking & Management:



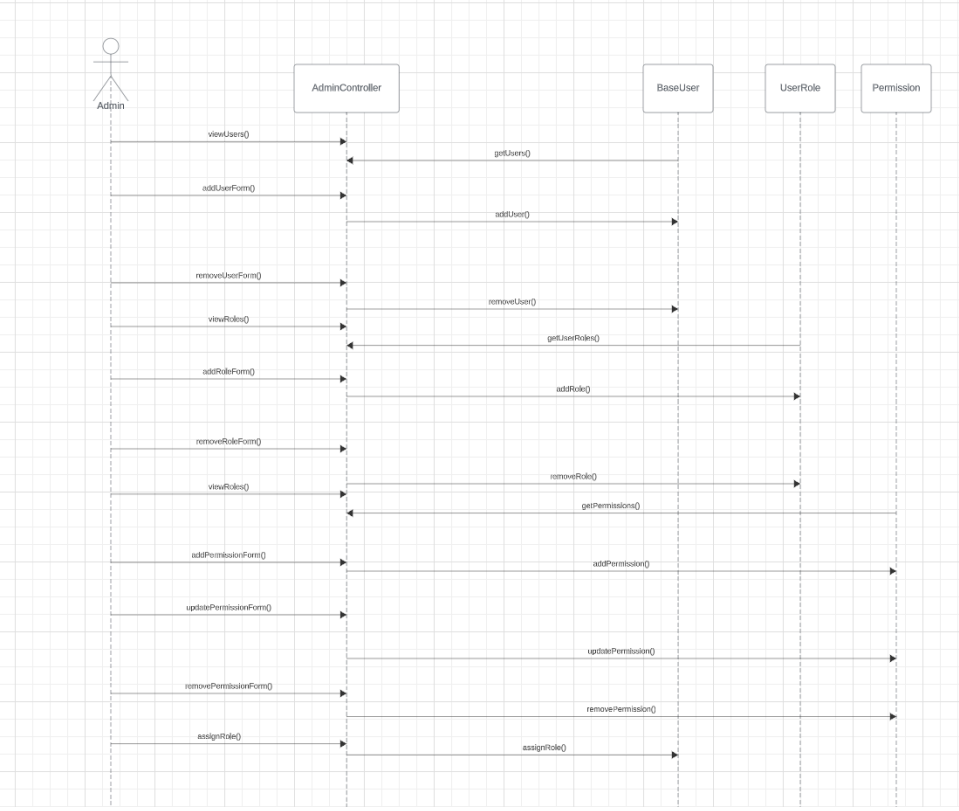
Medical History Maintenance:



Schedule:



System Configuration & Management:



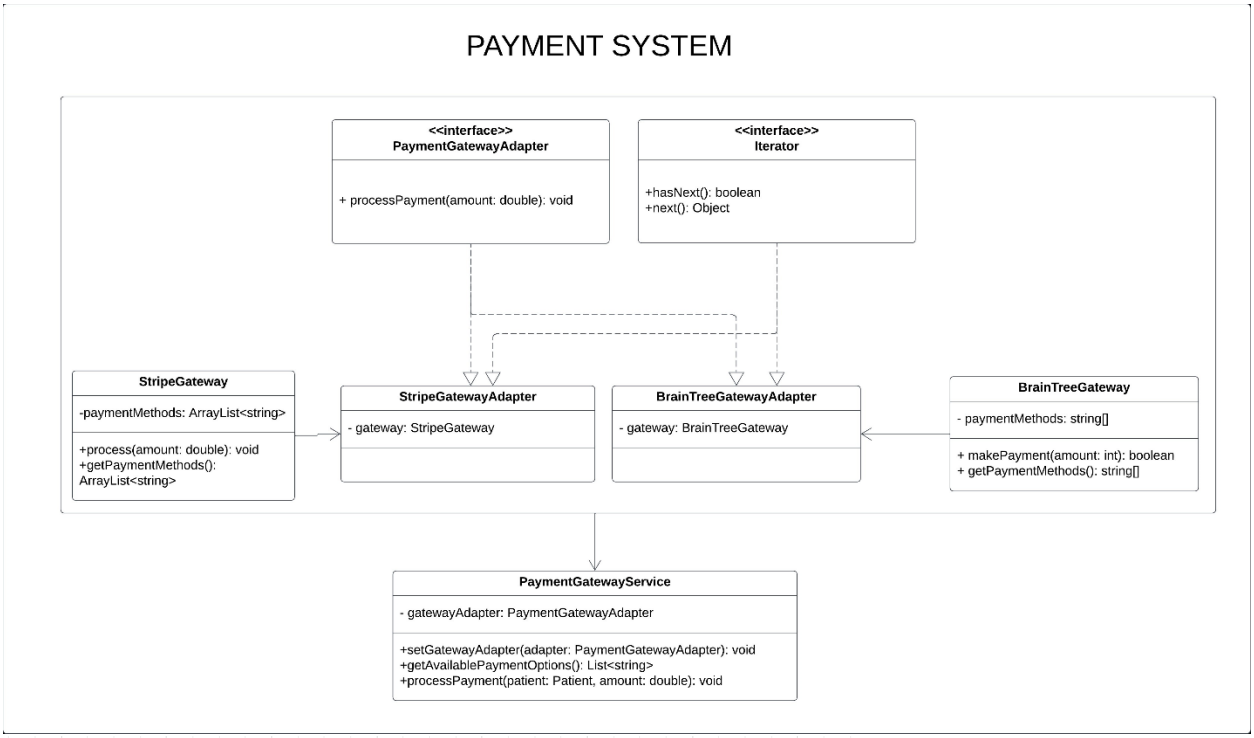
9. Single Responsibility Principle Implementation

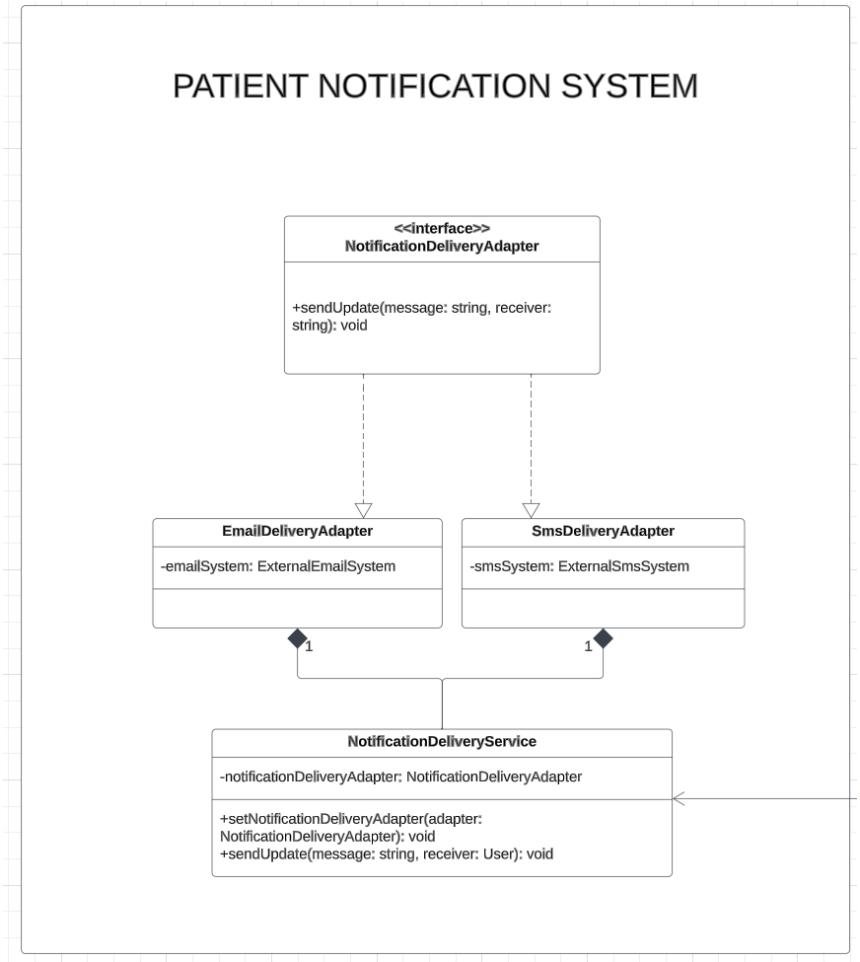
- Each class hiding its internal details, exposing only the necessary methods and attributes to achieve its single responsibility.
- Our class diagram follows Open-Closed Principle (OCP) principles as we used a factory pattern so that we can add more types of users like Nurses...etc. We used adapter pattern so that our application is closed for modifications but open for extensions for external systems.

10. Stable Intermediary Structures Implementation

As the IHMS interacts with various external systems, IHMS must include components or patterns that act as intermediaries between different parts of the system. They should be designed to absorb changes in the system without causing disruptions. The intermediaries should also be mature regarding API surface, or in other words, remain stable despite modifications or variations in the system.

In our work, we designed the system to have several stable intermediary structures, such as the payment system and the notification system. Take the payment system, for example:


PAYMENT SYSTEM
<<interface>>
PaymentGatewayAdapter
+ processPayment(amount: double): void
<<interface>>
Iterator
+hasNext(): boolean
+next(): Object
StripeGateway
-paymentMethods: ArrayList<string>
+process(amount: double): void
+getPaymentMethods():
ArrayList<string>
StripeGatewayAdapter
- gateway: StripeGateway
BrainTreeGatewayAdapter
- gateway: BrainTreeGateway
BrainTreeGateway
- paymentMethods: string[]
+ makePayment(amount: int): boolean
+ getPaymentMethods(): string[]
PaymentGatewayService
- gatewayAdapter: PaymentGatewayAdapter
+setGatewayAdapter(adapter: PaymentGatewayAdapter): void
+getAvailablePaymentOptions(): List<string>
+processPayment(patient: Patient, amount: double): void


PATIENT NOTIFICATION SYSTEM
<<interface>>
NotificationDeliveryAdapter
+sendUpdate(message: string, receiver:
string): void
EmailDeliveryAdapter
-emailSystem: ExternalEmailSystem
SmsDeliveryAdapter
-smsSystem: ExternalSmsSystem
1
1
NotificationDeliveryService
-notificationDeliveryAdapter: NotificationDeliveryAdapter
+setNotificationDeliveryAdapter(adapter:
NotificationDeliveryAdapter): void
+sendUpdate(message: string, receiver: User): void

We applied the principles of stable intermediary structures as follows:

- **Abstraction and encapsulation:** We hide the complexities of concrete payment gateways such as Stripe or BrainTree by providing a common interface shared with all payment solutions.
- **Extensibility:** The intermediary structure allows IHMS to integrate with new external systems without having to modify the existing implementation.
- **Configuration-centric:** The intermediary structures receive a configuration object to control the behavior. Changes can be absorbed by modifying the configuration rather than the internal structure.
- **Separation of concerns:** Each intermediary structure has a clear separation of concerns, with well-defined boundaries and responsibilities. For instance, each adapter is only concerned with the logic of a specific payment gateway. The implementation details of the payment gateway is only shared with the adapter and not anything else.
- **Design pattern utilization:** We implement the Adapter pattern, a tried-and-true solution, to help encapsulate variations and provide stable interfaces for interaction.

11. Intermediary Objects for Mediation

We designed our class diagram using the layering concept. We have data, controller, view and external system in different layers.

As the system grows, this will help us to manage the system more efficiently. Also testing and debugging the system will be easier.

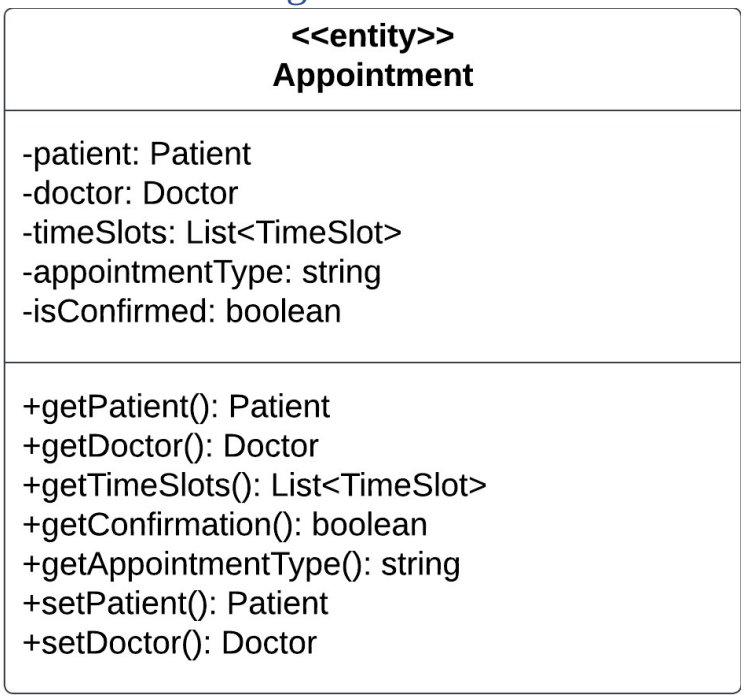
This way we can add new objects or change existing ones without affecting the rest of the system thus it makes the system more flexible and scalable.

12. Minimizing Dependency Between Classes

To minimize dependency between classes, we employed the following methods:

- We used the MVC pattern to separate user interface (UI) and business logic.
- We used the Factory pattern to define different types of users with the common attributes of the BaseUsers.
- We used the Adaptor pattern to integrate with different subsystems like payment gateways. This will help the developer to expand the application in the future.
- We used Observer pattern to send the notifications to all subscribers when an inventory alert is triggered.

13. Maximizing Relatedness of Methods within a Class

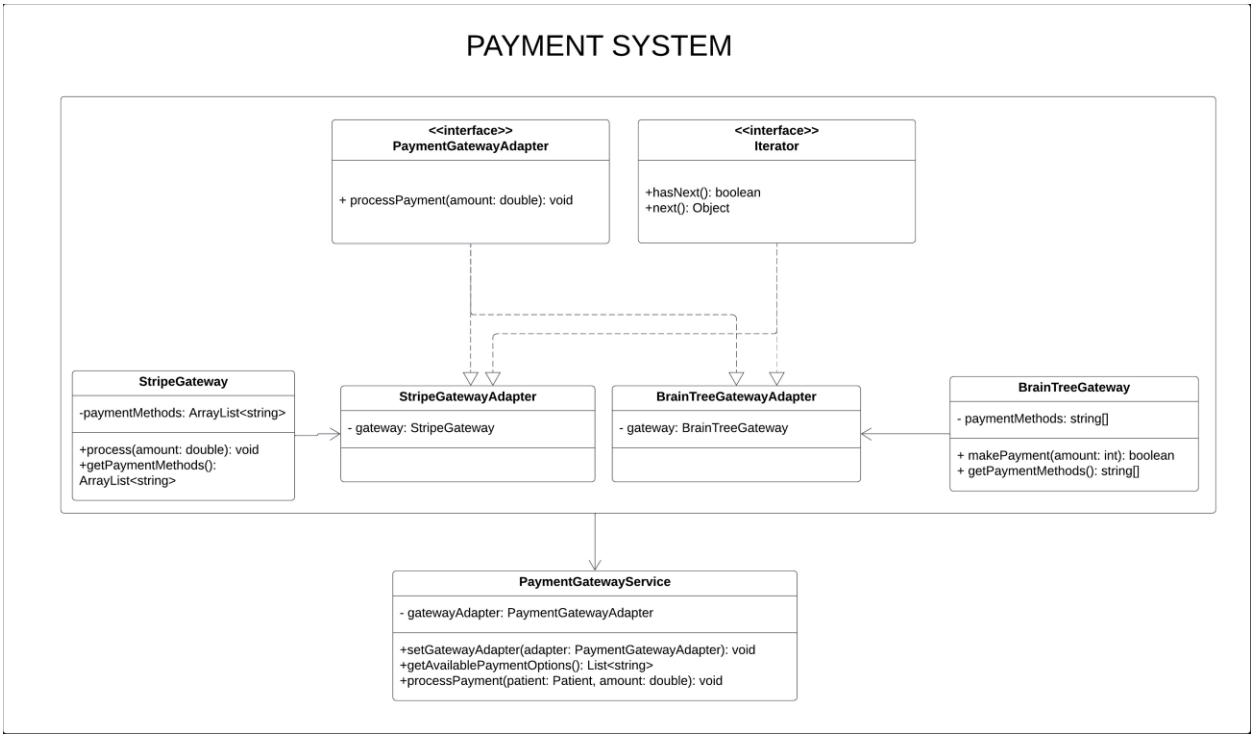


As you can see, the Appointment entity class's methods only work on updating or getting the information within the Appointment class only. As the attributes are created and the getter setter of the class can only update the data.

All the business functions are separated from the data class, and we put them in the controllers, so the entity classes are kept separate.

14. Adapter Design Pattern Integration

In our proposal, the Adapter design pattern is implemented in the Payment System and Notification System.



In the example of a payment system, we define the interface PaymentGatewayAdapter and implement that interface in classes BrainTreeGatewayAdapter and StripeGatewayAdapter that act as adapters accordingly for payment gateways BrainTreeGateway and StripeGateway. The PaymentGatewayService uses the adapter to process payments.

There are many reasons why we should implement this design pattern:

Reusability: By adopting the adapter pattern, the existing code can be easily reused. If the IHMS is integrating with an existing healthcare system, we just need to create new adapters to bridge the gap and allow seamless integration.

Flexibility and extensibility: The pattern allows IHMS to work with different implementations through a common interface. For instance, in the future, developers can integrate other external payment gateways with ease by creating new adapters that conform with the PaymentGatewayAdapter interface without modifying the existing code.

Well-defined responsibilities and separation of concerns: The conversion logic is encapsulated within the adapter. This keeps the rest of the system clean and focuses on its own functionality. For instance, since the payment adapters implement the same interface and act as a layer of abstraction, the PaymentGatewayService is not concerned with the implementation detail of each payment gateway.

15. Factory Design Pattern Implementation

We used Factory Design Pattern to create different types of Users. The reasons behind using the Factory Pattern:

Encapsulation: The Factory Pattern encapsulates the object creation process. Instead of creating BaseUser objects directly in the client code, you use the IBaseUserFactory to create instances. This encapsulation provides a central place to manage object creation logic.

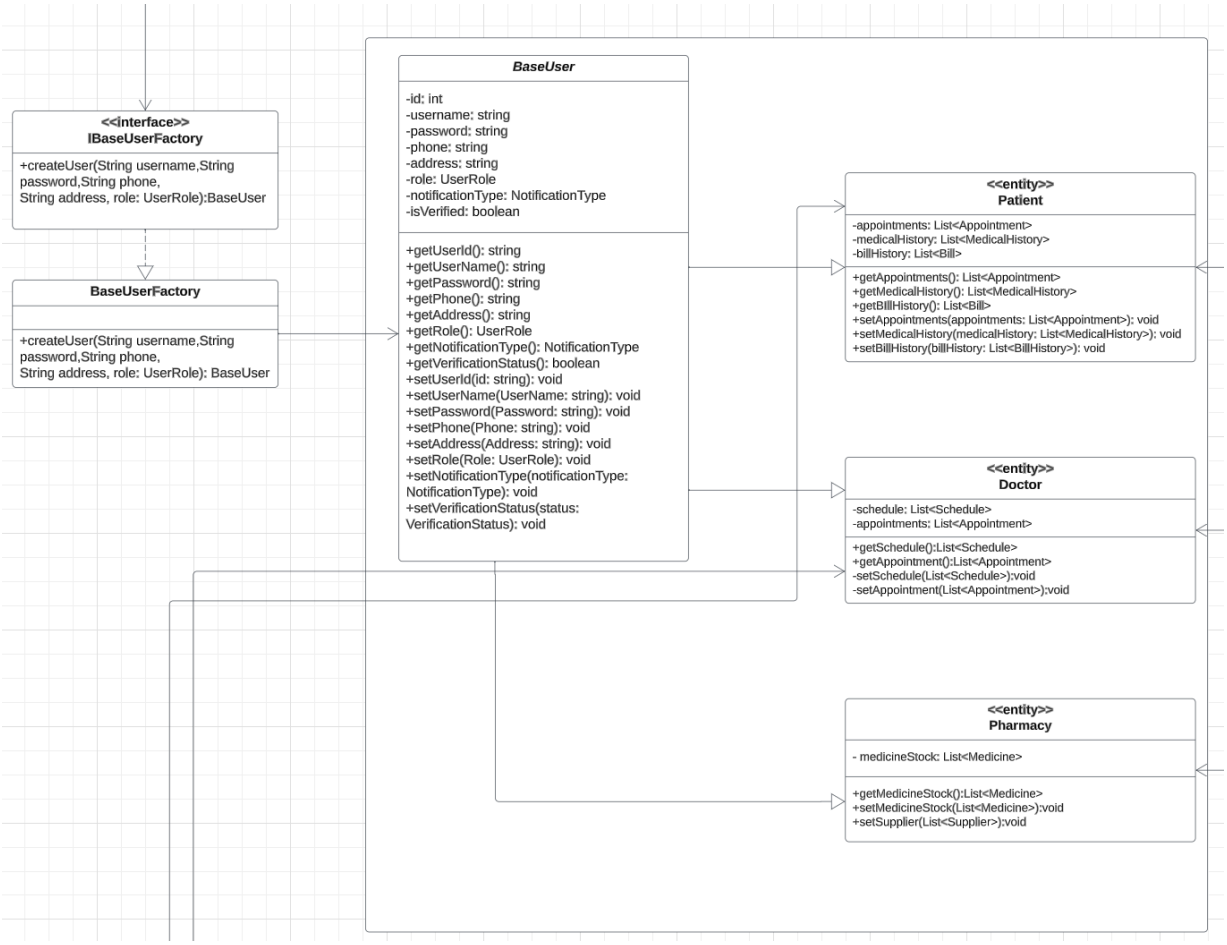
Abstraction: It abstracts the object creation details from the client code. The client code doesn't need to know how the BaseUser object is created; it simply requests a user with specific parameters, such as username, email, and role.

Consistency: It ensures consistent object creation. If we need to change the way BaseUser objects are created (e.g., add validation or logging), we can make those changes in the IBaseUserFactory without impacting the client code.

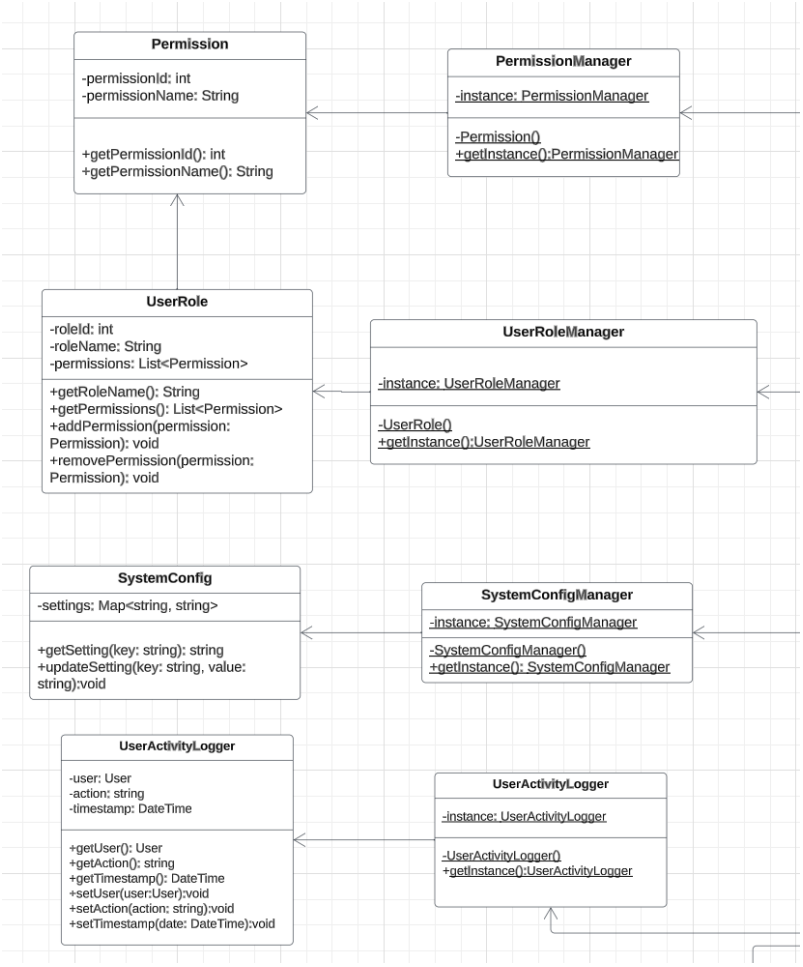
Testing: It simplifies unit testing. We can easily mock or replace the IBaseUserFactory during testing to control the creation of BaseUser objects, making it easier to write unit tests.

Decoupling: It promotes decoupling between the client code and the concrete BaseUser class. The client code doesn't depend on the specific implementation of User. Instead, it relies on the factory to provide instances of User.

Extension: It allows for the easy extension of the object creation process. For example, you can create different types of users with varying attributes and behaviors by extending or modifying the IBaseUserFactory without changing the client code. So in future we can separate the different types of user like external user and medical users.



16. Singleton Design Pattern Application



S**ingle Point of Access:** we wanted to ensure that there is only one instance of the system configuration and user roles data in our application. This is useful for maintaining consistency and avoiding conflicts when multiple parts of the application need to read or modify this data.

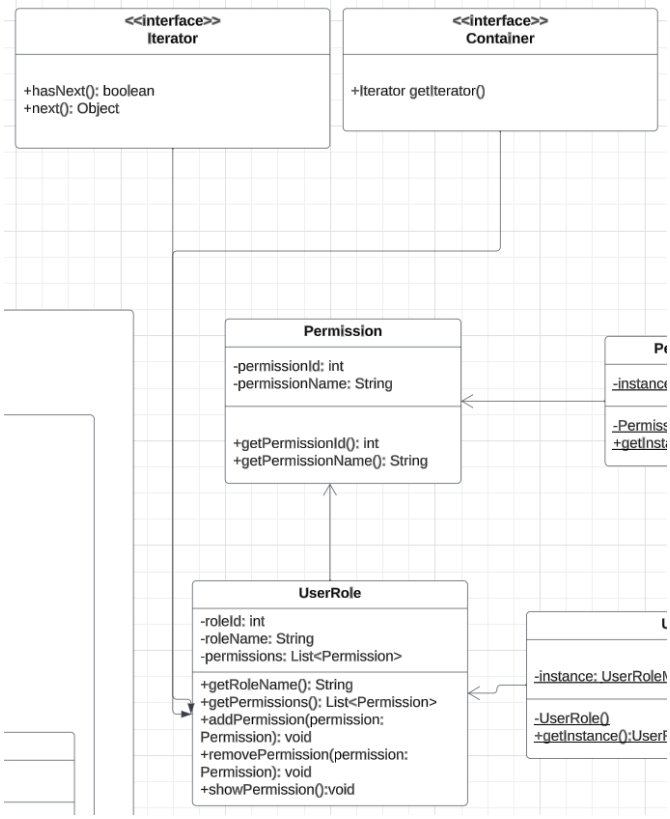
Resource Efficiency: Using a Singleton ensures that us to create and manage the resource efficiently by having just one instance.

Global State Management: Singleton allows the admin to maintain global state in a controlled and organized way. System configuration and user roles are typically application-wide concerns, and the

Consistency and Synchronization: The admin can manage data consistency and synchronization more effectively.

Lazy Initialization: The instances are created only when it's requested for the first time, which can be helpful in scenarios where the data initialization might be costly.

17. Iterator Design Pattern Usage

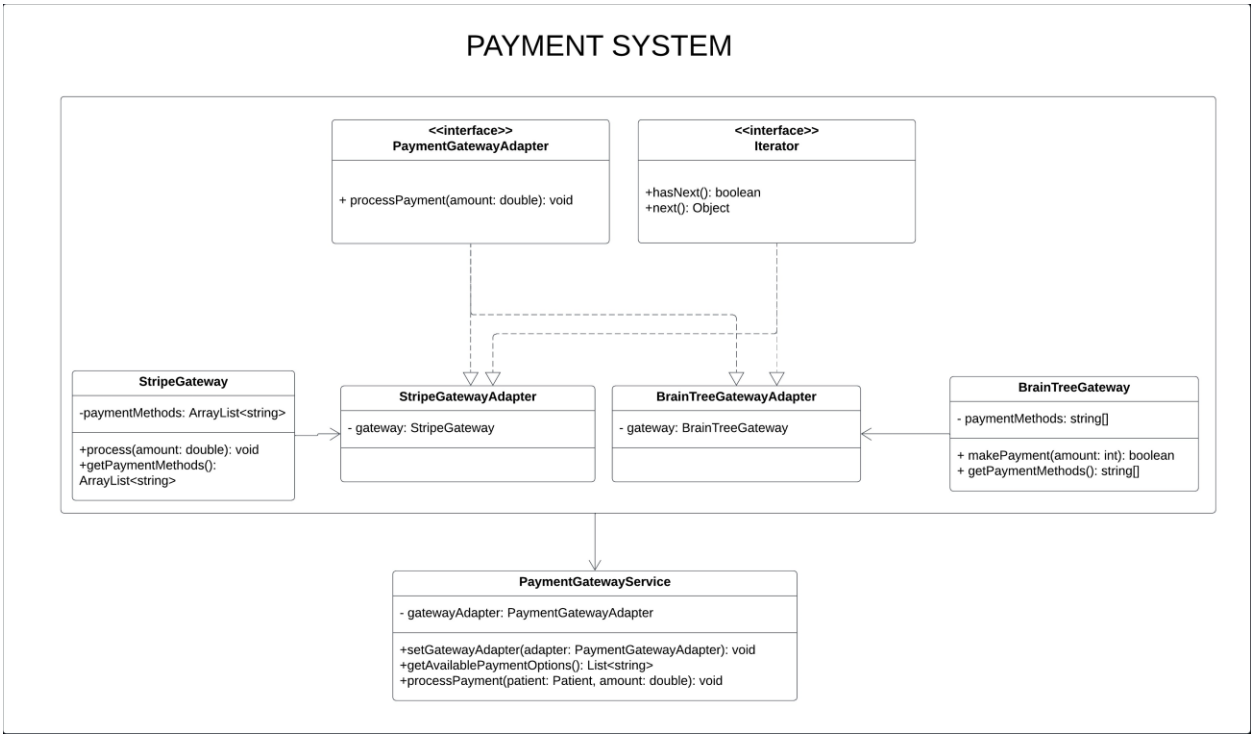


Here is the demonstration of how to access the elements of a collection permission objects from the UserRole class to access the permissions for a user:

```
// Iterator.java
public interface Iterator {
    public boolean hasNext();
    public Object next();
}

// Container.java
public interface Container {
    public Iterator getIterator();
}

// PermissionRepository.java
public class PermissionRepository implements Container{
    public String permissionList[] ={"perimission1","permission2","permission3"};
    @Override
    public Iterator getIterator(){
        return new PermisiionIterator();
    }
    private class PermisiionIterator implements Iterator{
```

```
        int index;
        @Override
        public boolean hasNext(){
            if(index<permissionList.length){
                return true;
            }
            return false;
        }
       @Override
       public Object next(){
        if(this.hasNext()){
            return permissionList[index++];
        }
        return null;
       }
    }
}

// UserRole.java
public class UserRole {
   public void showPermission(){
    PermissionRepository newPermissionRepository = new PermissionRepository();


    for(Iterator iterator= newPermissionRepository.getIterator();
iterator.hasNext();){
        String permission =(String)iterator.next();
        System.out.println("User Permission: "+ permission);
    }
   }
}
```

Here is another example of the Iterator pattern that we used in payment gateway.



We also implemented the Iterator design pattern in the payment system. Different payment gateways could maintain different types of data structures for the payment methods. In our case, the StripeGateway uses an ArrayList of string, whereas the BrainTreeGateway uses an array to represent the list of payment methods. By implementing the Iterator interface, both StripeGatewayAdapter and BrainTreeGatewayAdapter provide a common way for the consumer to iterate over the available payment methods.

The reasons we used the Iterator design pattern are as follows:

Abstraction: The pattern offers a level of abstraction, allowing the consumer to iterate over the collection via a common interface. For instance, the consumer PaymentGatewayService will use the hasNext() and next() methods from the Iterator interface.

Encapsulation: The pattern encapsulates the internal structure of a collection, making a clear separation between the traversal mechanism and the collection data structure. In the payment example, the consumer is also not concerned with the implementation details of the payment methods in StripeGateway and BrainTreeGateway.

Concurrent iterations: Since each iterator maintains its own index position, concurrent traversal is made possible without interfering with each other.

18. Implementing technical constraints

+ Encryption: We would apply database encryption for Users and MedicalHistory following HIPAA compliance. The sensitive information, such as user information and medical history, cannot be readable even with direct database access.

+ Data scrubbing: If there is any case of copying the data to different environments for debugging and development purposes, we need to scrub all the data to remove sensitive information.

+ Strong authentication and authorization mechanism: We use MFA authentication to prevent the possibility of unauthorized access. Only authorized users can access the data and functions assigned to their roles.

+ Auditing: The system maintains a log history of all user activities to detect anomalies and notify the administrators to act accordingly.