Addis Ababa Institute of Technology
School of Information Technology and Engineering
Department of SiTE.

Efoyta Doctor's Appointment Booking System
Software Design Specification

Section 2 - Group 4

Team Members

1. Biniyam Assefa Mekonnen····· UGR/8320/14
2. Fikreyohanes Abera Shibru····· UGR/8889/14
3. Isam Ahmed Hussen····· UGR/7643/14
4. Kaleb Asratemedhin Bekele····· UGR/9104/14
5. Meklit Asrat Tefera····· UGR/5387/14
6. Daniel Demerew Endale····· UGR/9530/14
7. Feysel Hussien Seid····· UGR/5898/14

Advisor: Nuniyat Kifle

Date: December 16, 2023

Table of Content

List of Tables

List of figures

Definitions, Acronyms, and Abbreviations

CBD····· Component Based Development

CSS····· Cascading Style Sheets

MVC·····Model-View- Controller

MySQL····· My Structured Query Language

RDBMS····· Relational Database Management System

SDS·····Software Design Specification

UI····· User Interface

UML····· Unified Modeling Language

1. Introduction

1.1. Purpose

The software design specification document serves as a comprehensive blueprint for the software project, detailing the architecture, functionality, and behavior of the system to be developed. It outlines the technical specifications, design principles, and implementation details that guide the development team throughout the project life cycle. It acts as a reference for developers, providing a clear and structured representation of the system's components, their interactions, and the overall system architecture.

Additionally it translates the business requirements and processes into a technical design that will be used to develop the application. It intends to capture and convey the significant architectural and design decisions that have been made for the system.

1.2. General Overview

The doctor appointment booking system is a web-based application that interacts with external entities such as a database server, an email server, a payment gateway, and a notification service, and allows users and doctors to book and manage their appointments online.

The system will use a three-tier architecture. Such an architecture is well-suited for an appointment booking system due to its modular and scalable design, providing a clear separation of concerns. In this architecture, the system is divided into three main layers: the presentation layer, the business logic layer, and the data storage layer. The presentation layer, also known as the user interface, handles the interaction with end-users. In the context of an appointment booking system, this layer would manage the interface through which users, such as patients or medical staff, interact with the system to schedule or manage appointments. Separating the presentation layer from the other components enhances maintainability and allows for flexibility in adapting to various user interfaces, such as web applications or mobile apps.

The business logic layer serves as the brain of the application, managing the core functionality and processing of data. For an appointment booking system, this layer

would handle tasks like validating appointment requests, checking availability, and enforcing business rules. The modular design of the business logic layer ensures that changes to the logic do not directly impact the user interface or data storage, promoting a more straightforward and organized development process.

The final layer, the data storage layer, is responsible for managing the system's data and ensuring its persistence. In an appointment booking system, this layer would store information about users, appointments, and other relevant data. Utilizing a relational database facilitates efficient data retrieval and manipulation.

1.3. Development Methods & Contingencies

The system and software design for the doctor appointment booking app is based on the object-oriented approach, using UML diagrams to model the data and processes. The design follows the waterfall methodology.

In the implementation of a three-tier architecture for an appointment booking system, the presentation layer would be developed using a component-based approach. The basic tools such as HTML, CSS, and JavaScript will be used together with React, a JavaScript library, that employs CBD(component-based approach), allowing to create reusable UI components. This modular approach facilitates the independent development and testing of components, leading to a more maintainable and scalable frontend.

In the business logic layer, node.js and express.js will be used to handle tasks such as validating appointment requests, managing availability, and enforcing business rules. The layer adopts the MVC architecture and handles business logic and server-side processing. Node.js serves as the runtime environment, and Express.js, a web application framework for Node.js, implements the MVC pattern. They will be well-suited for defining routes, handling HTTP requests, and implementing the core functionality.

The data storage layer involves the use of MySQL, a relational database management system (RDBMS) to store and retrieve appointment data. Through this modular and organized approach, a three-tier architecture not only ensures better maintainability

and scalability but also facilitates easier collaboration among development teams working on different aspects of the system.

The design goals of the system are to provide a user-friendly, secure, reliable, and scalable solution for booking and managing doctor appointments online.

The main challenges and contingencies that the design faces are the availability and reliability of the doctors' data, and the user experience and feedback. The design addresses these issues by adopting best practices and standards, such as encryption, authentication, and authorization, as well as conducting regular reviews and evaluations with the stakeholders and the end-users. The design also incorporates flexibility and scalability features, such as modular components, reusable code, and cloud-based hosting, to accommodate future changes and enhancements.

2. System Architecture

2.1. Subsystem decomposition

Below we present the subsystems that make up the online doctor's appointment booking system in three levels of decomposition. First the architecture to be used will be a three tier architecture with three layers. The three layers are the presentation layer, business logic layer and the data layer. The figures below do not represent the the three tiers separately but rather the whole system decomposed into three levels. Therefore, each level gives varying detail about all the three tiers in the architecture.

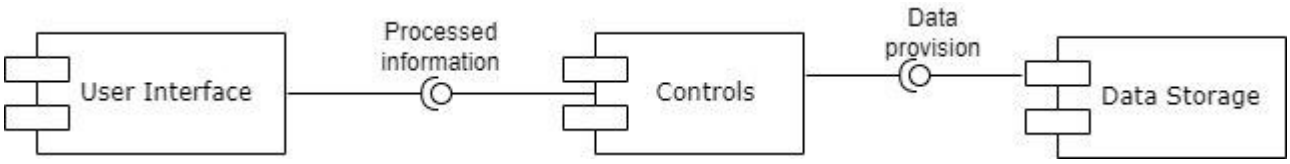

Figure 1: Level 1 System Decomposition

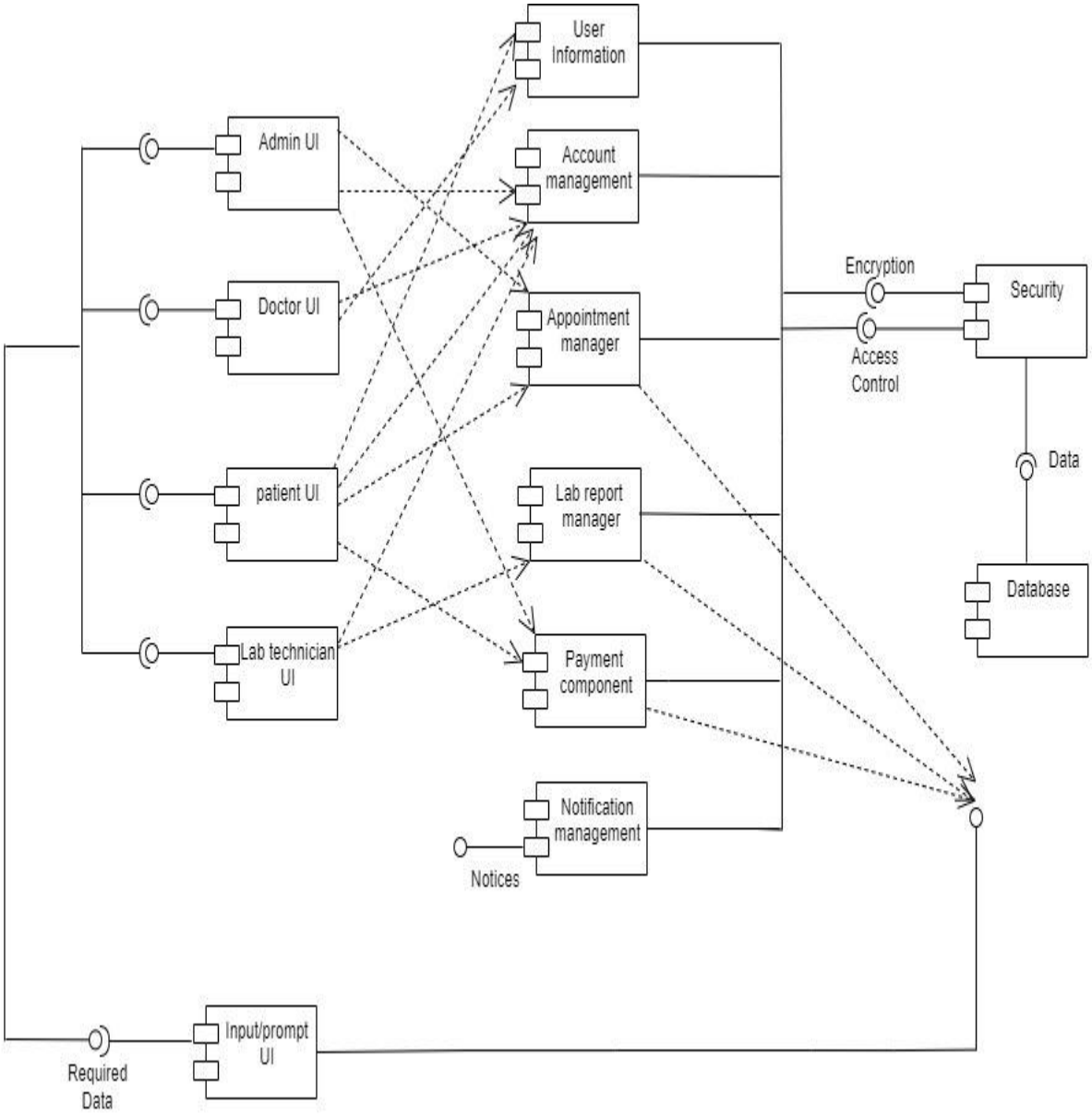

Figure 2: Level 2 System Decomposition

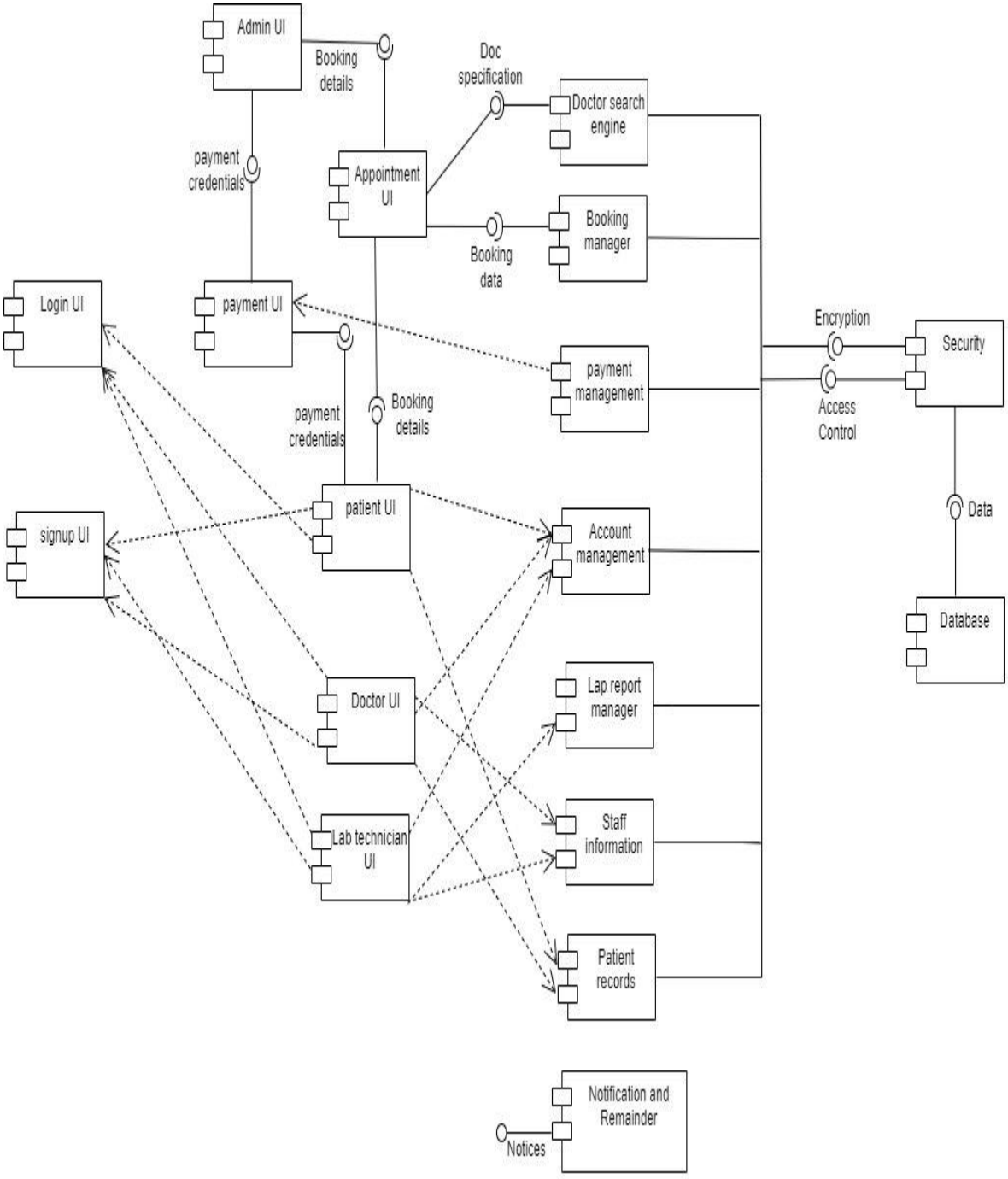

Figure 3: Level 3 System Decomposition

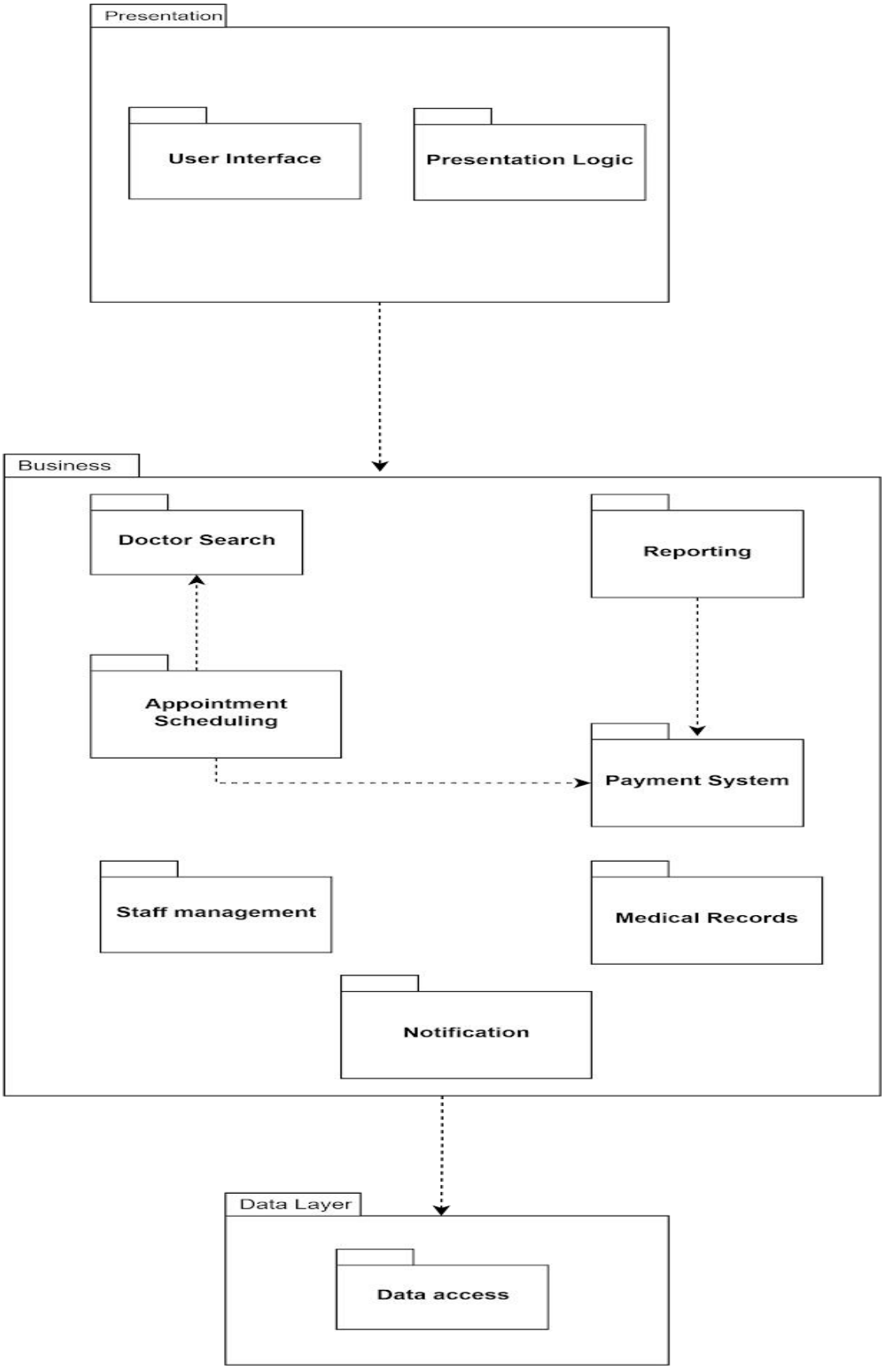

Figure 4: UML Package Diagram

2.2. Hardware/software mapping

UML Deployment diagram

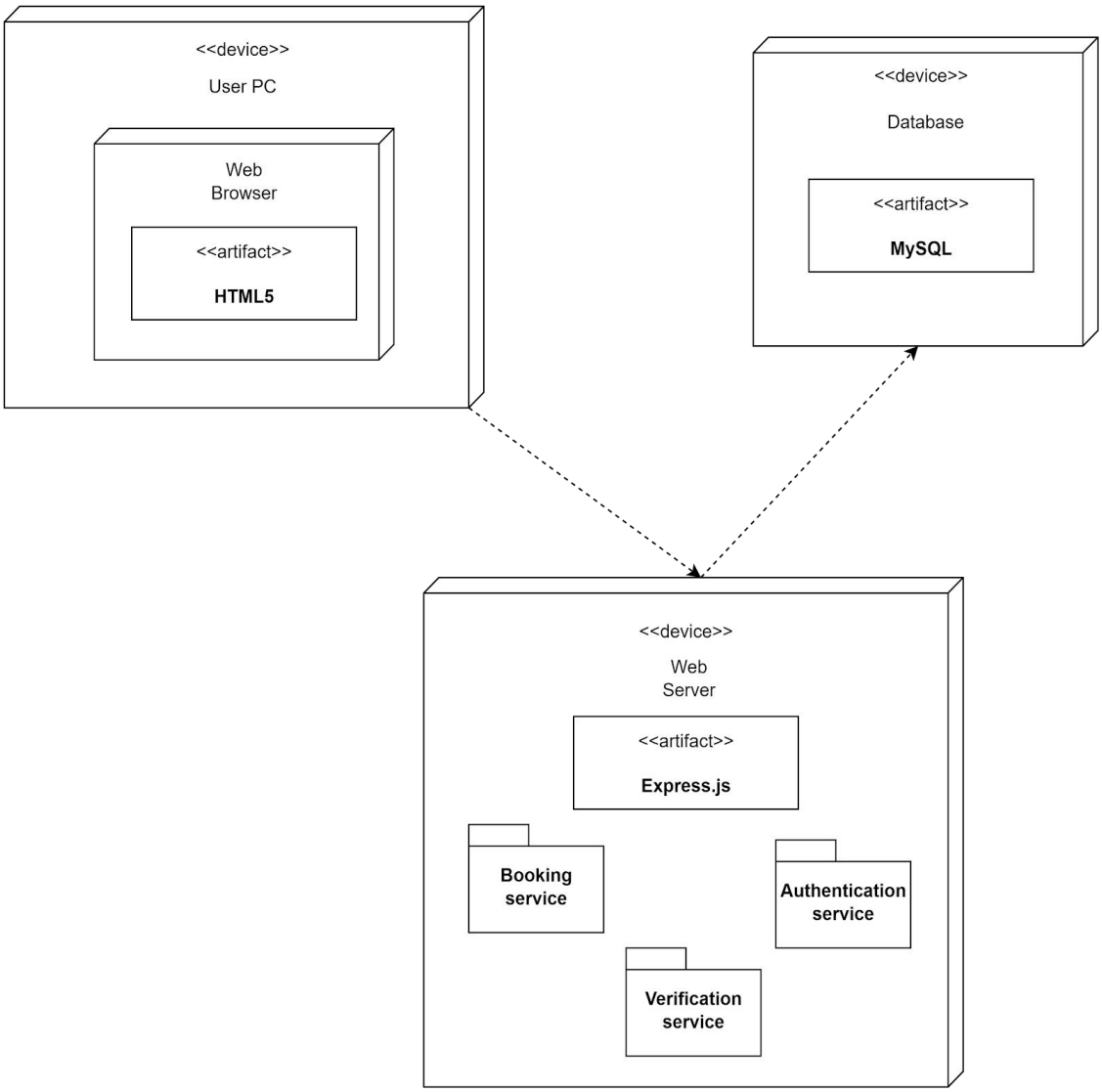

Figure 5: UML Deployment Diagram

3. Object Model

3.1. Class Diagram

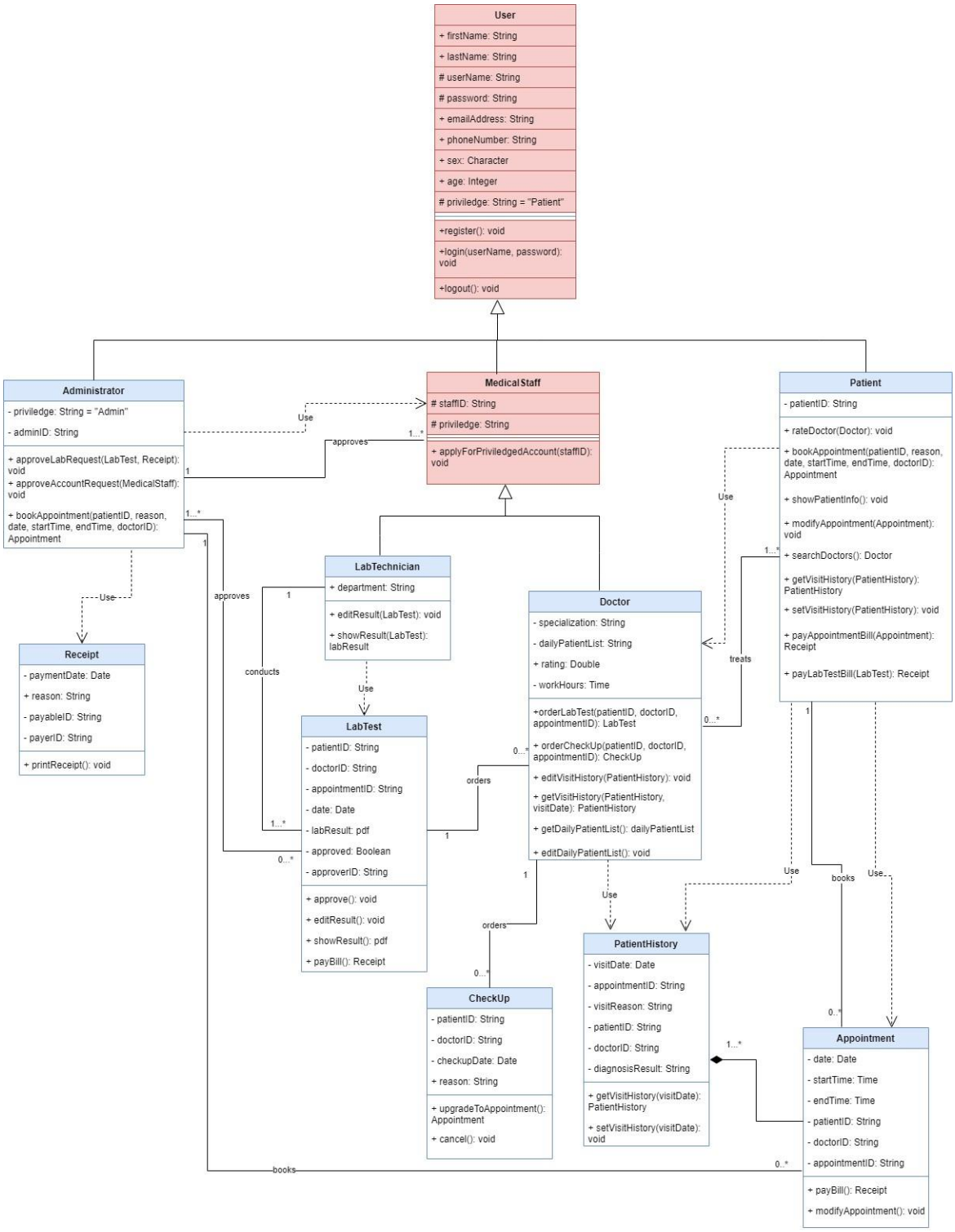

Figure 6: UML Class Diagram

You can get a better view of the diagram here.

3.2. Sequence Diagram

3.2.1. Create account

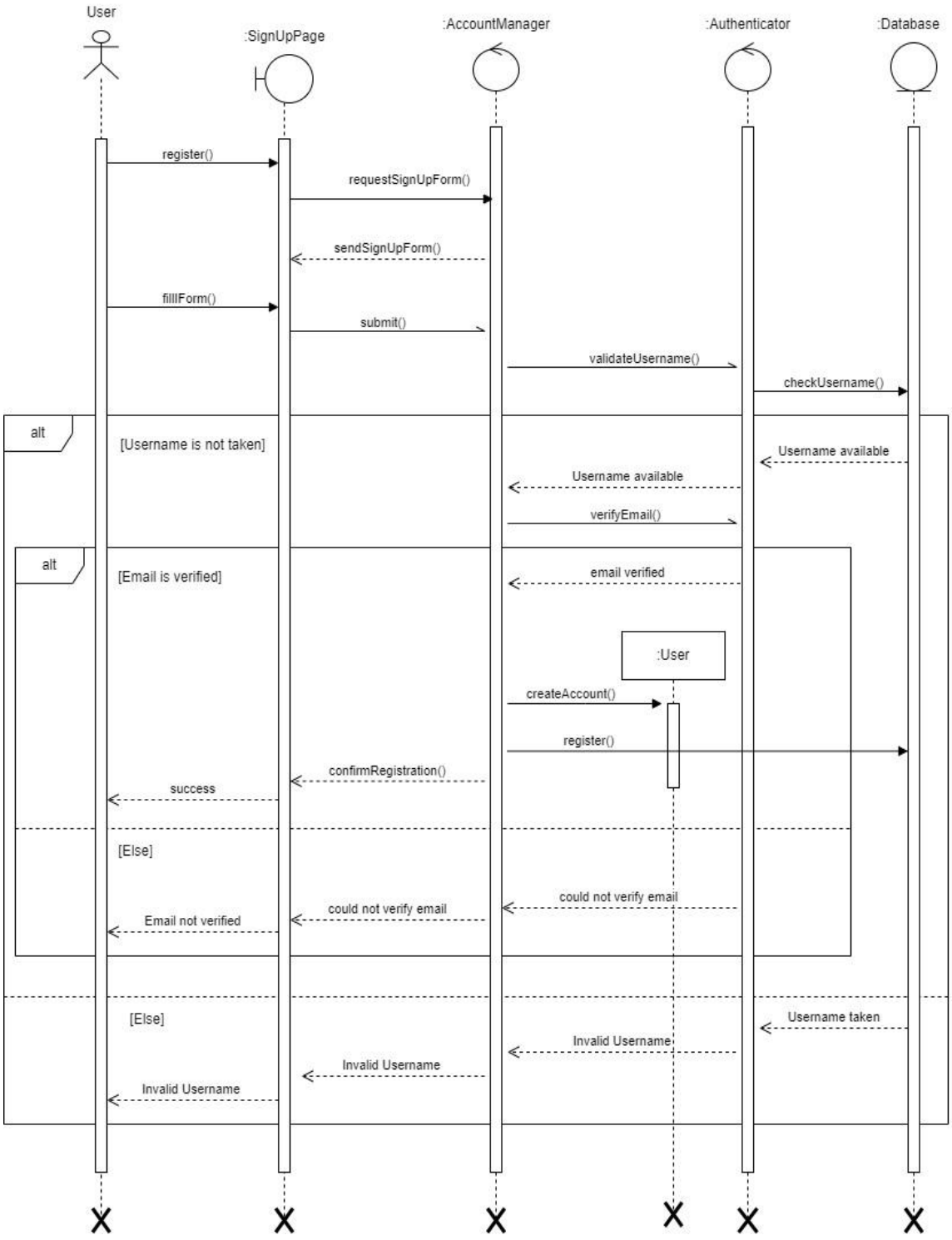

Figure 7: Create Account Sequence Diagram

3.2.2. Login

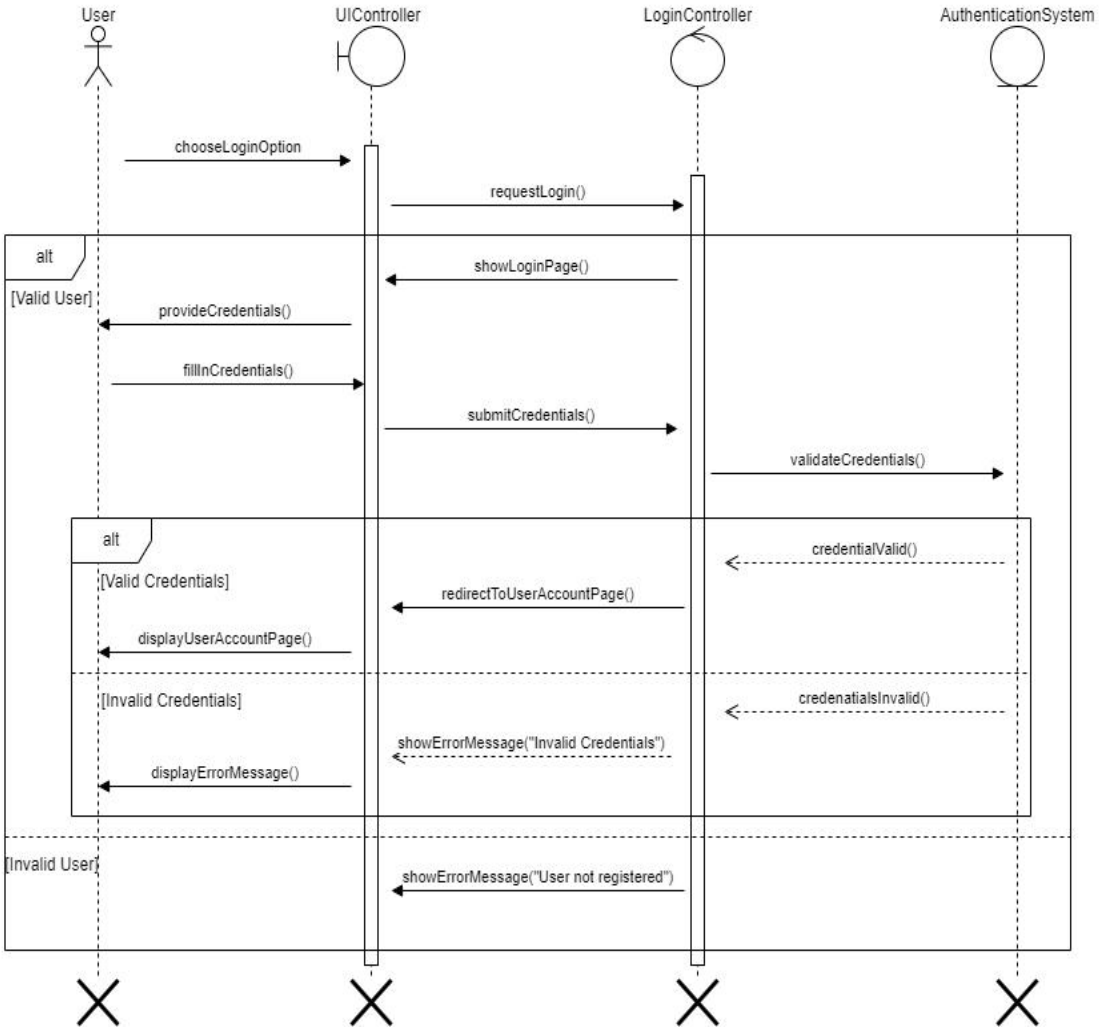

Figure 8: Login Sequence Diagram

3.2.3. Privileged Account

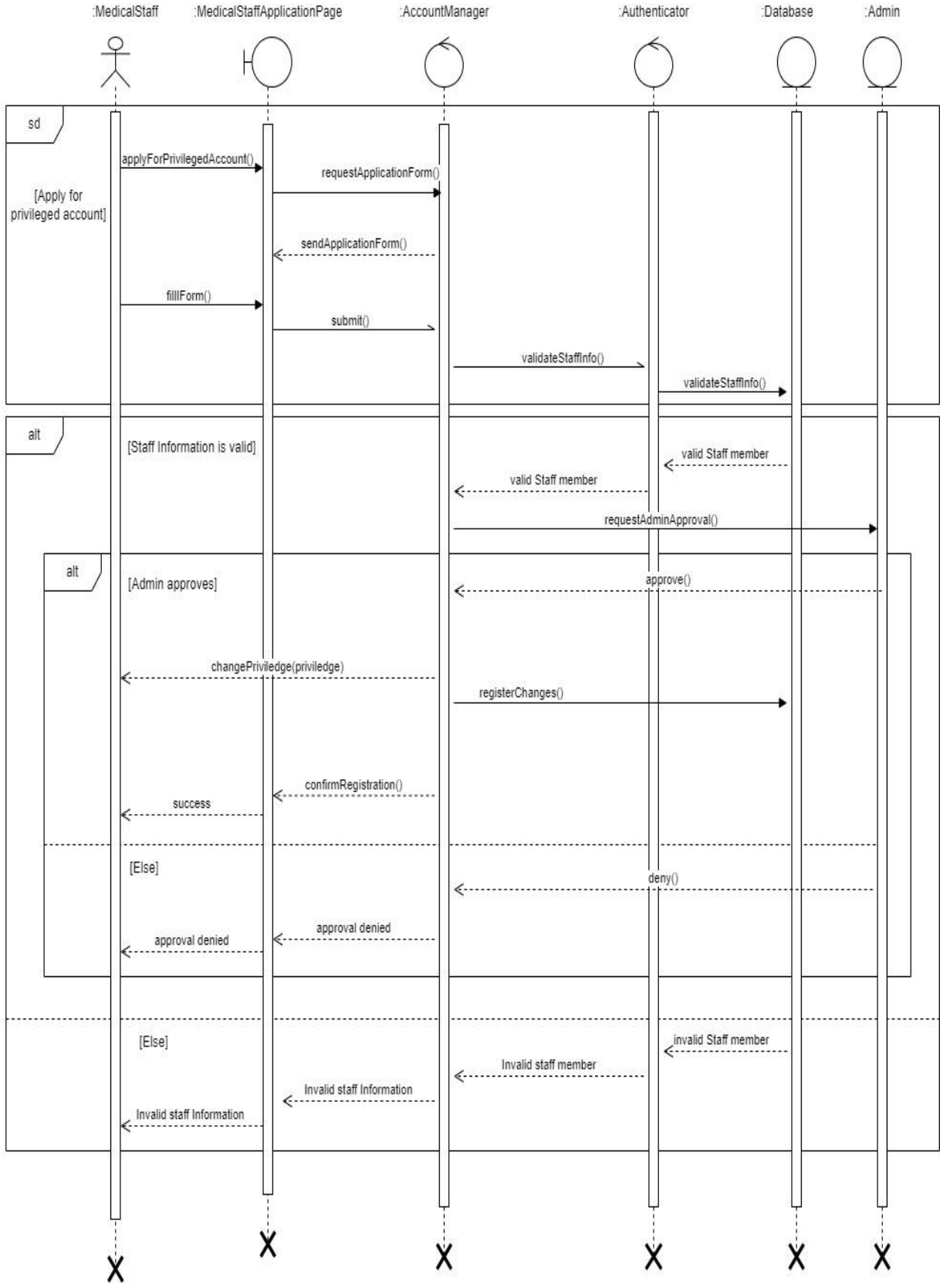

Figure 9: Privileged Account Sequence Diagram

3.2.4. Search Doctor

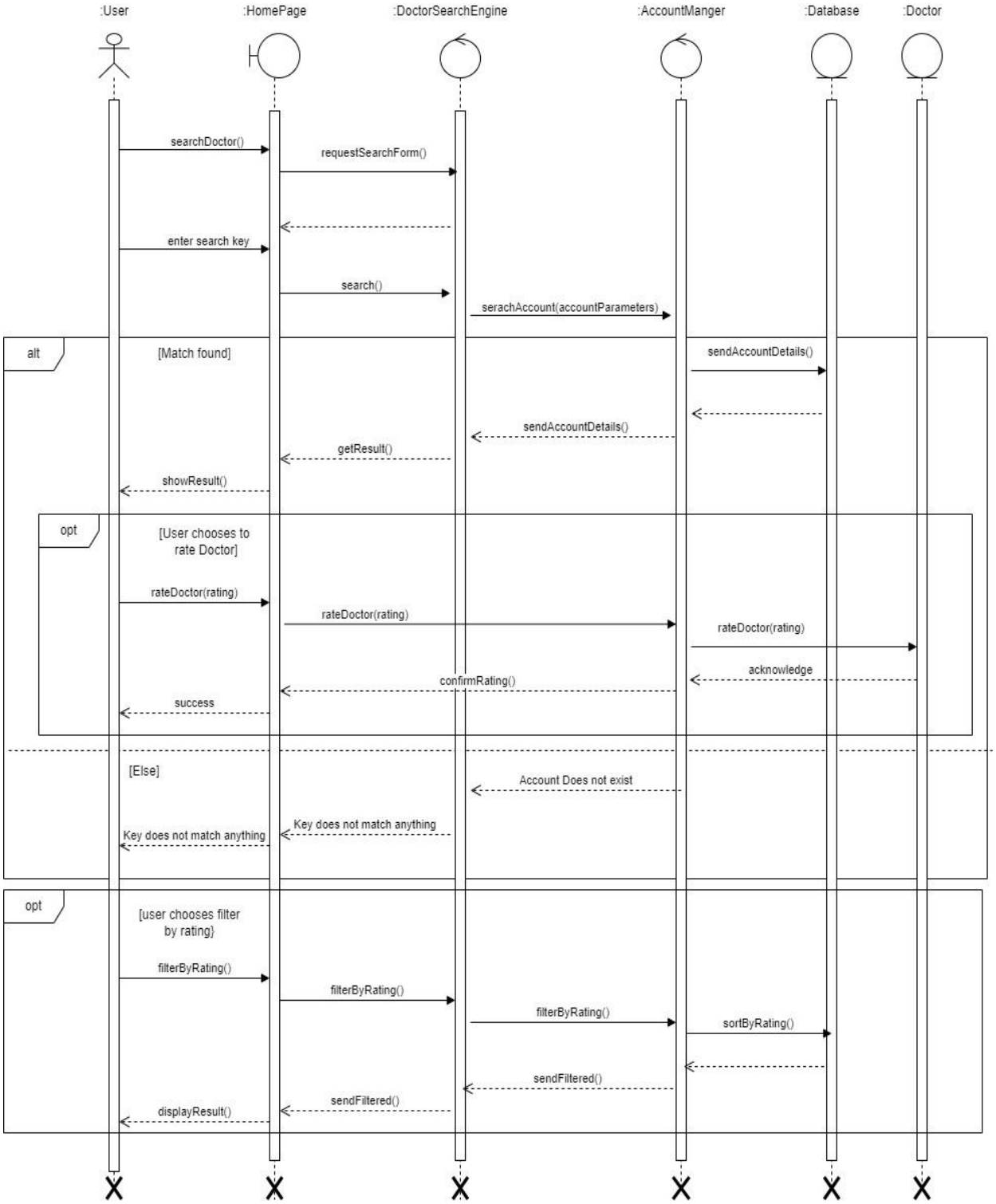

Figure 10: Search Doctor Sequence Diagram

3.2.5. Book Appointment

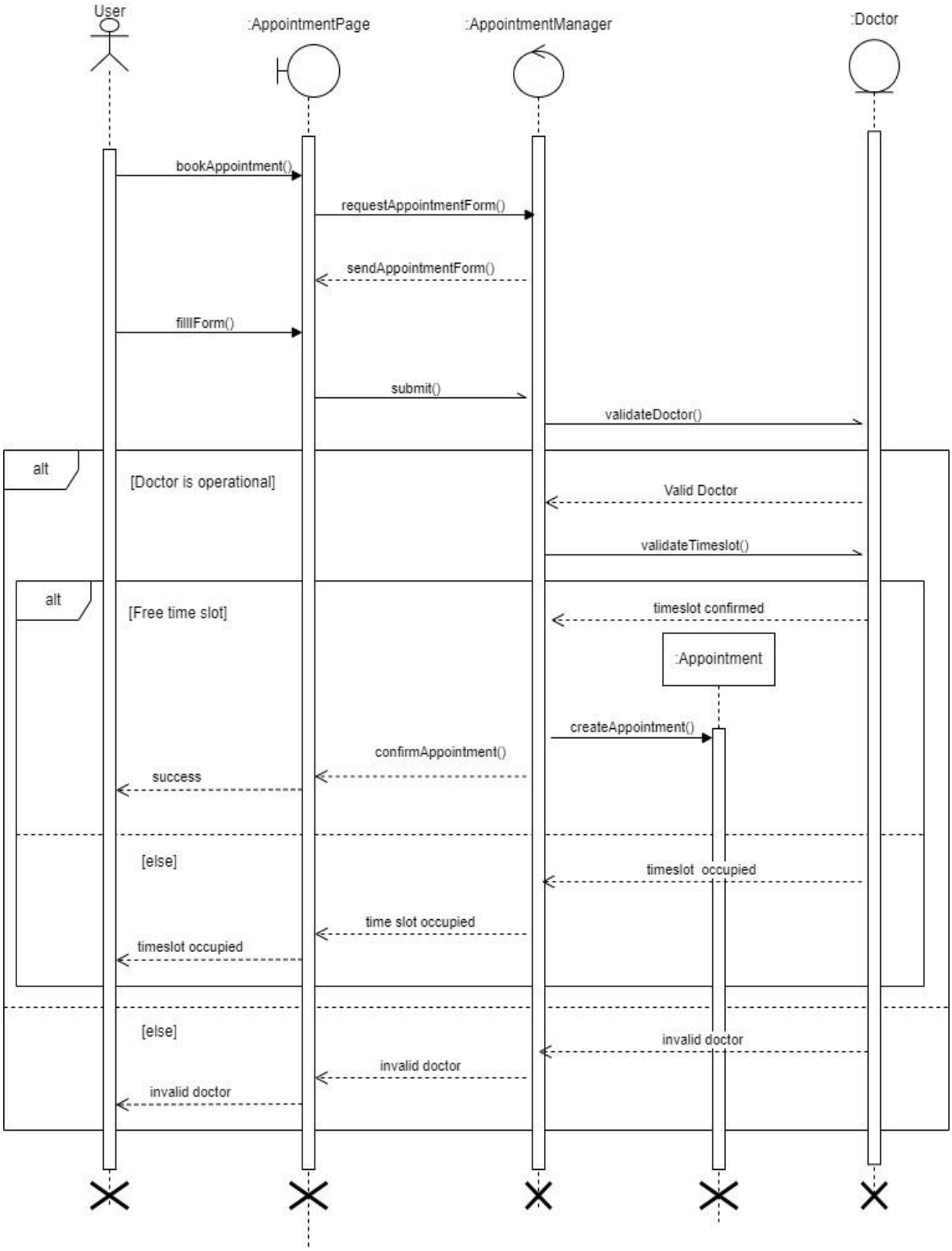

Figure 11: Book Appointment Sequence Diagram

3.2.6. Pay Appointment Bills Online

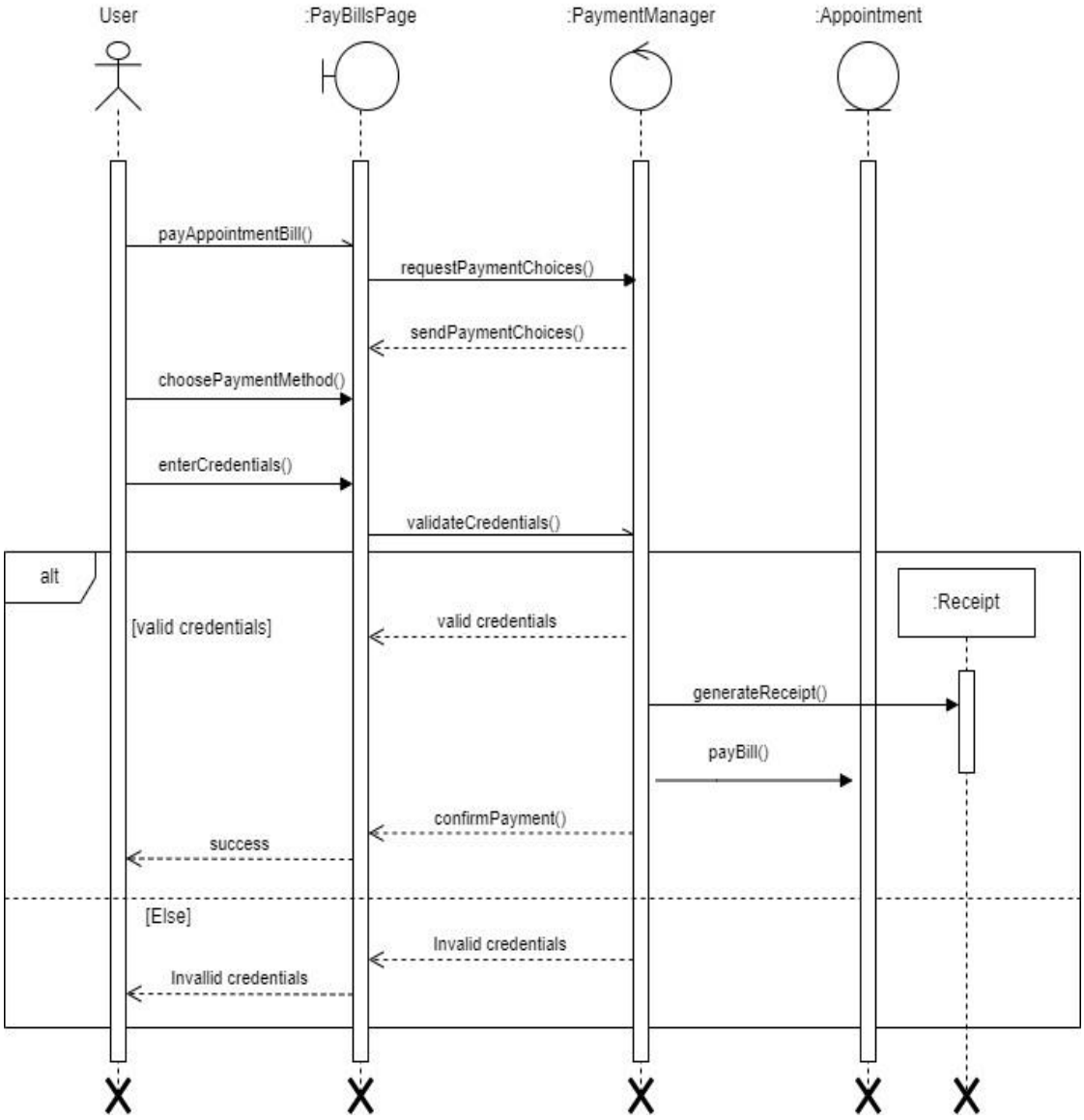

Figure 12: Pay Appointment Bills Online Sequence Diagram

3.2.7. Modify Appointment

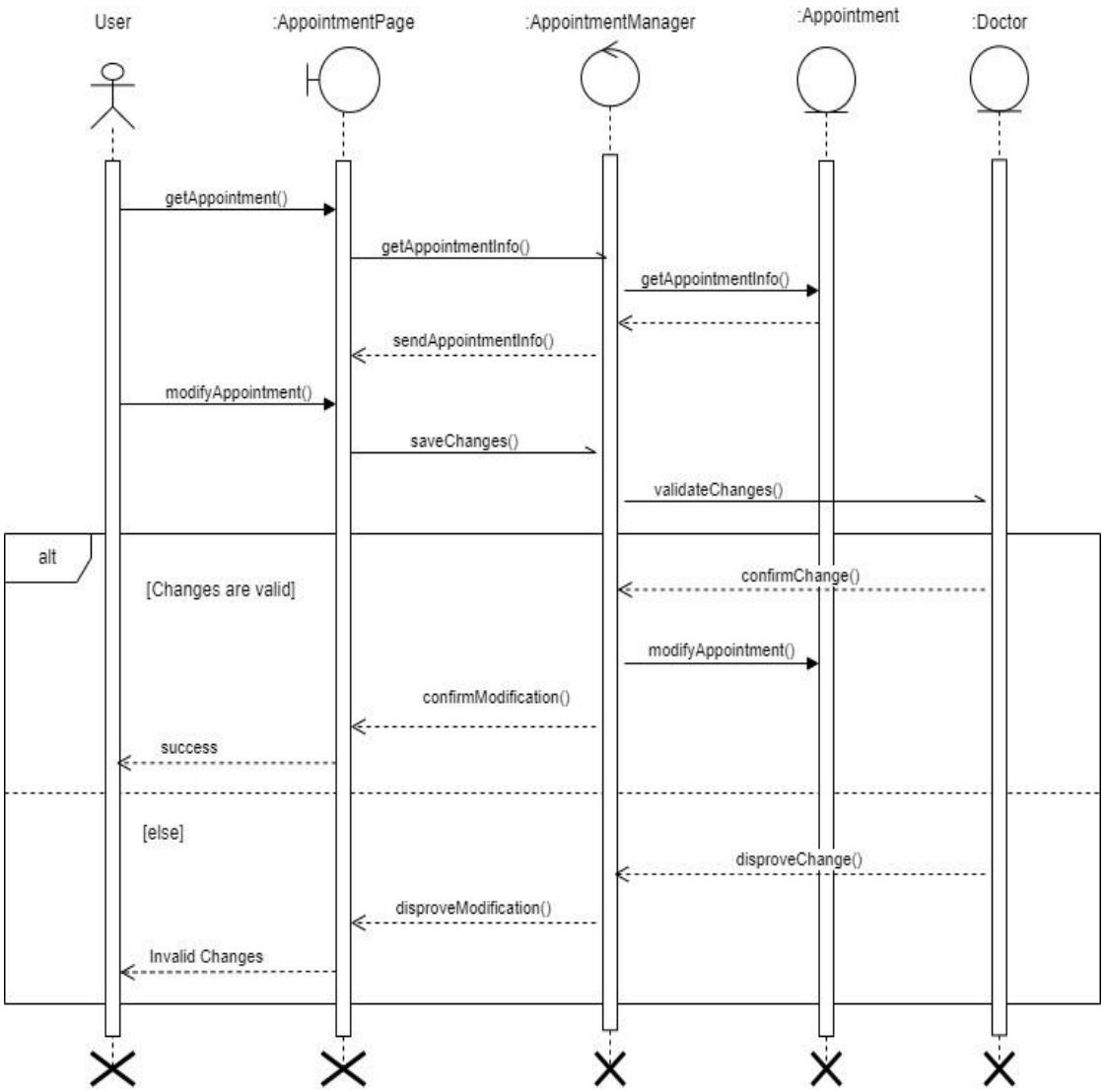

Figure 13: Modify Appointment Sequence Diagram

3.2.8. Request Lab Test

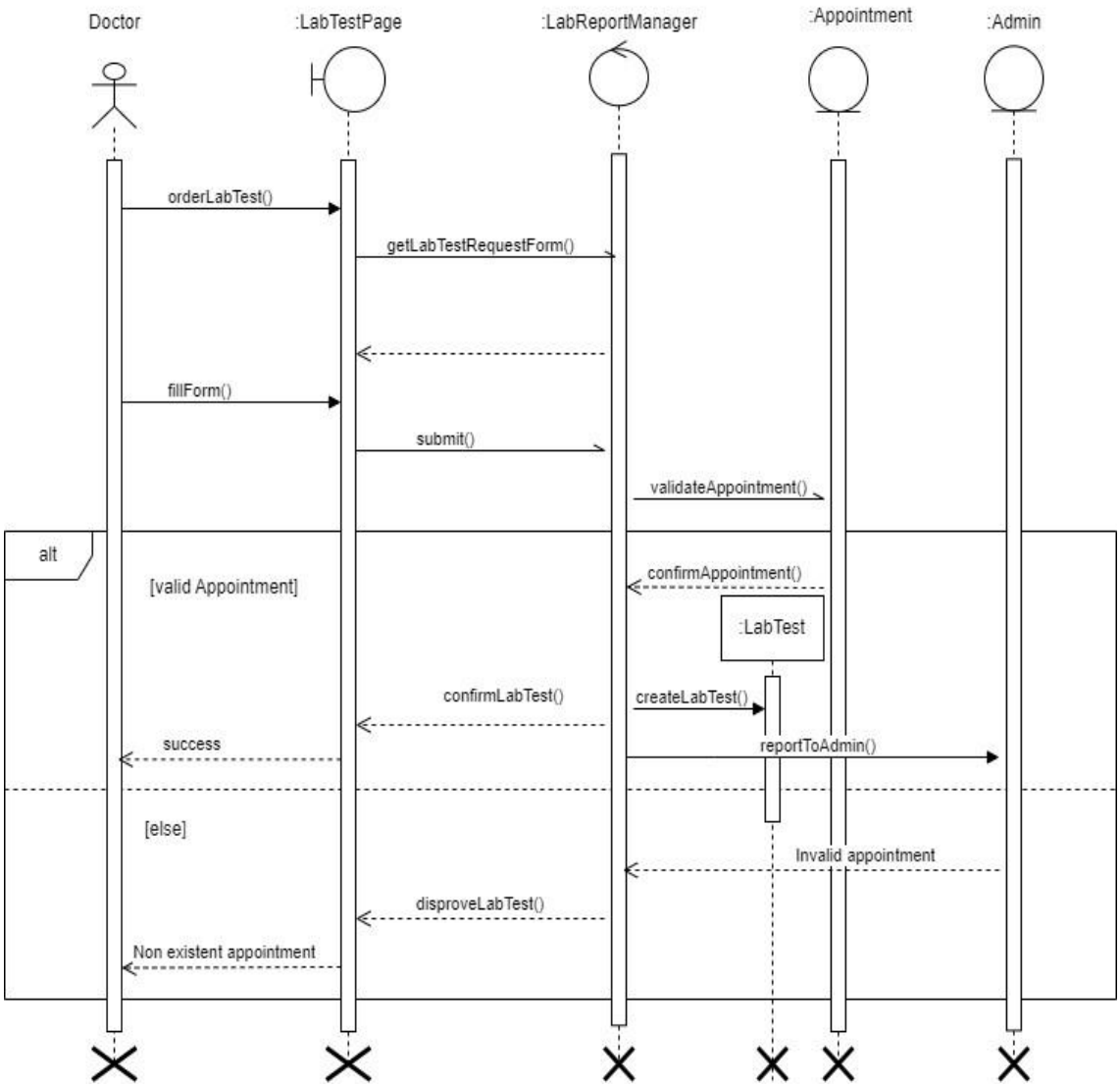

Figure 14: Request Lab Test Sequence Diagram

3.2.9. Pay Lab Bill

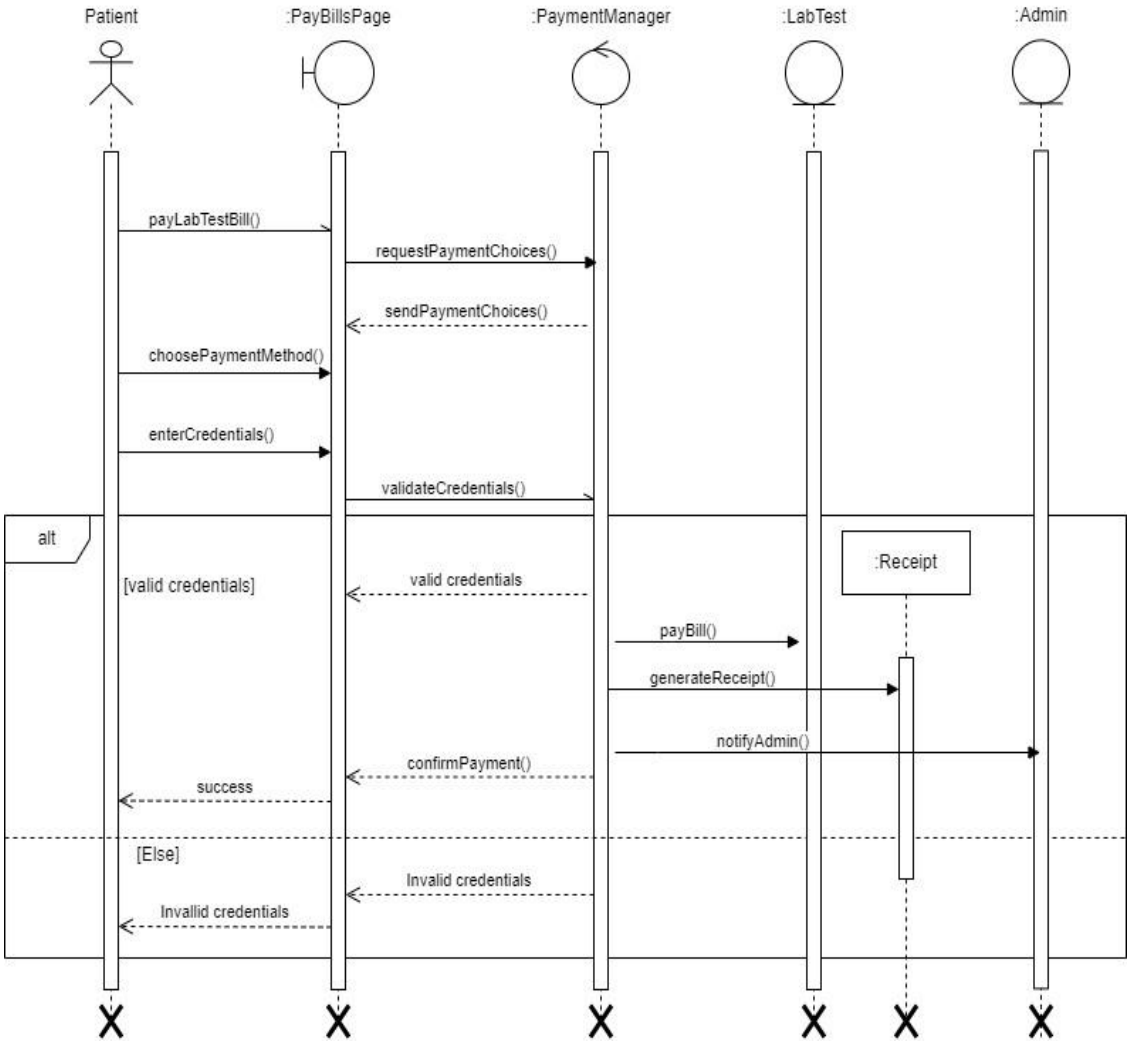

Figure 15: Pay Lab Bill Sequence Diagram

3.2.10. Approve Lab Test

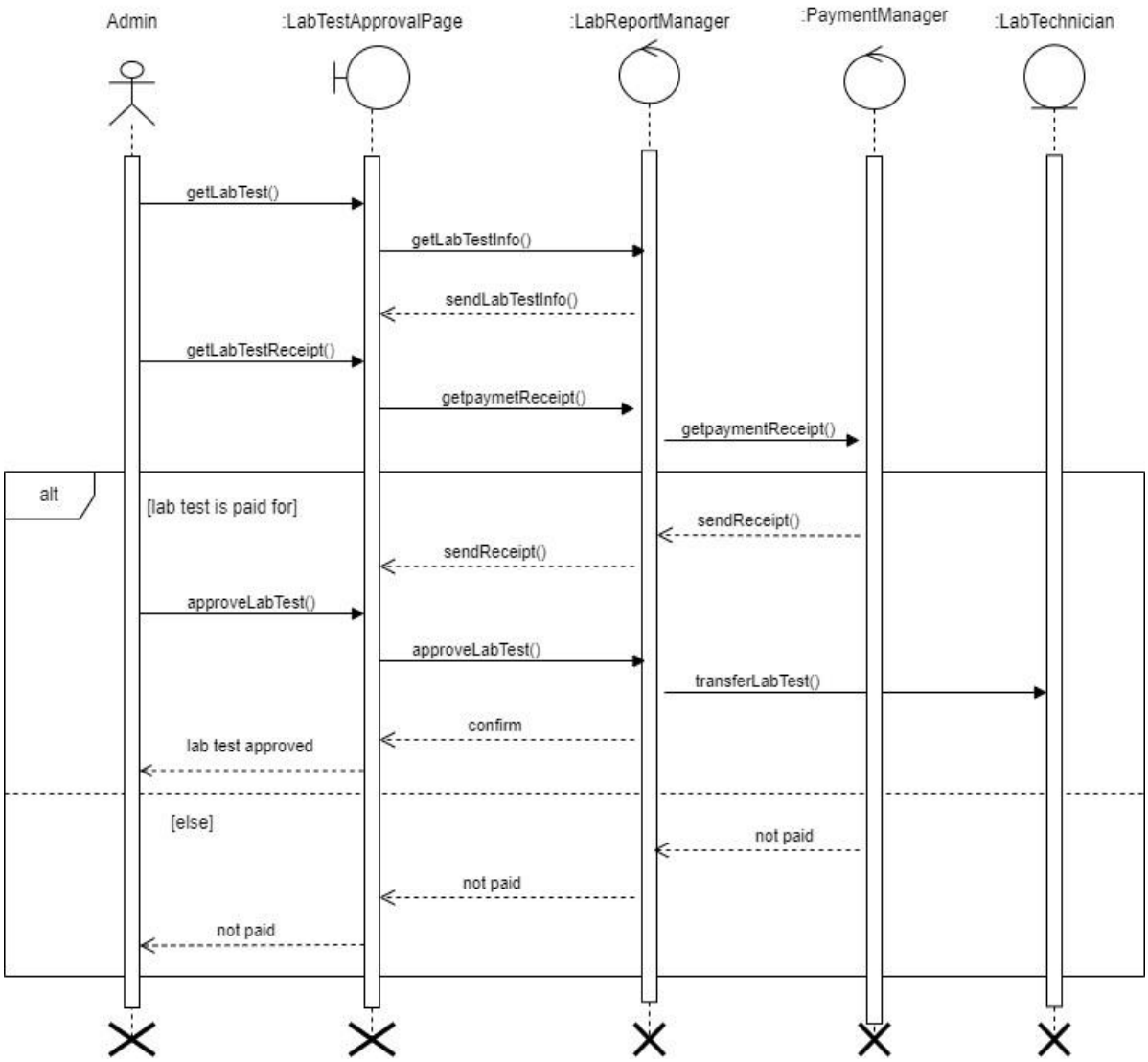

Figure 16: Approve Lab Test Sequence Diagram

3.2.11. Send Lab Result

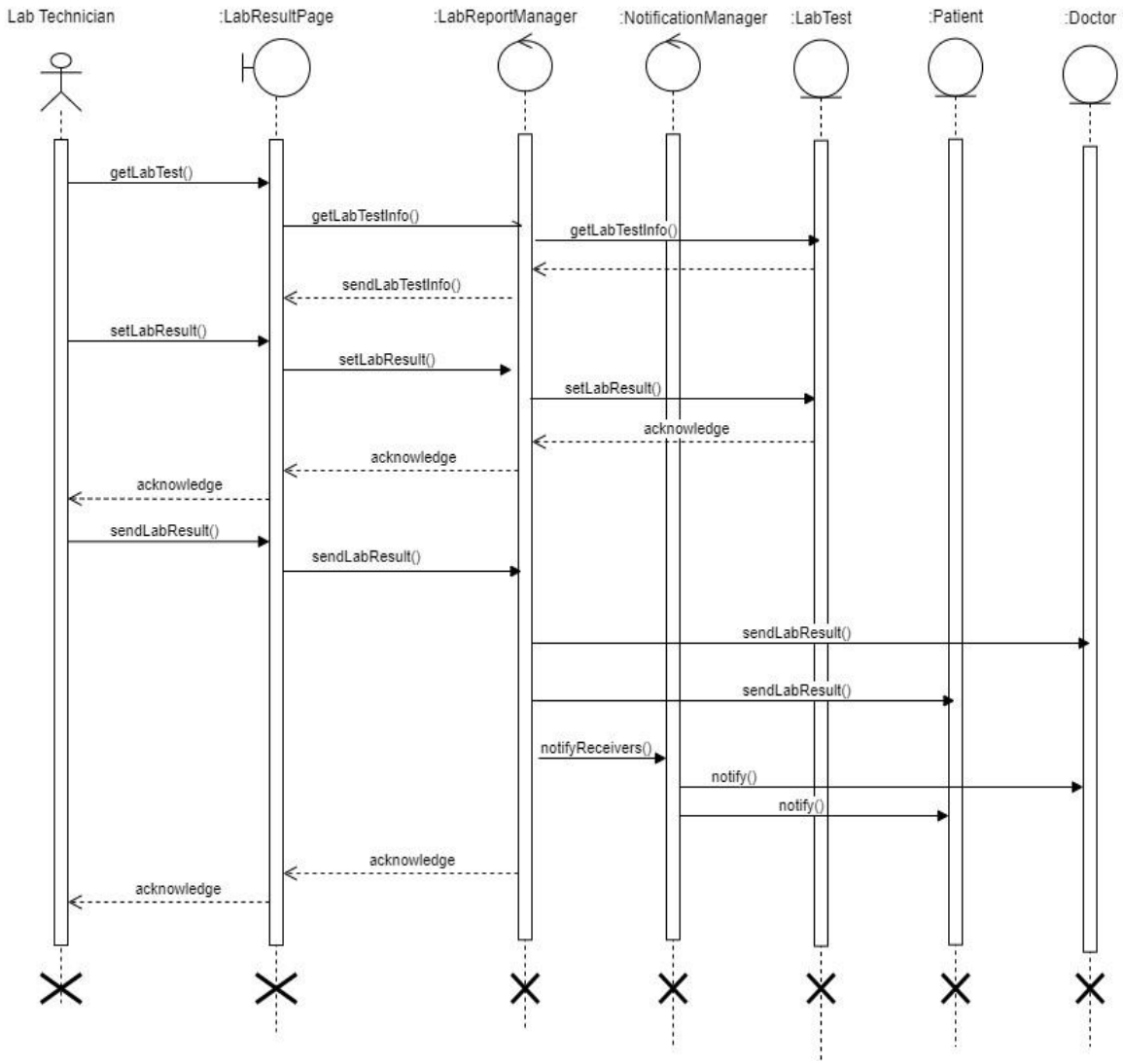

Figure 17: Send Lab Result Sequence Diagram

3.2.12. Checkout Lab Result

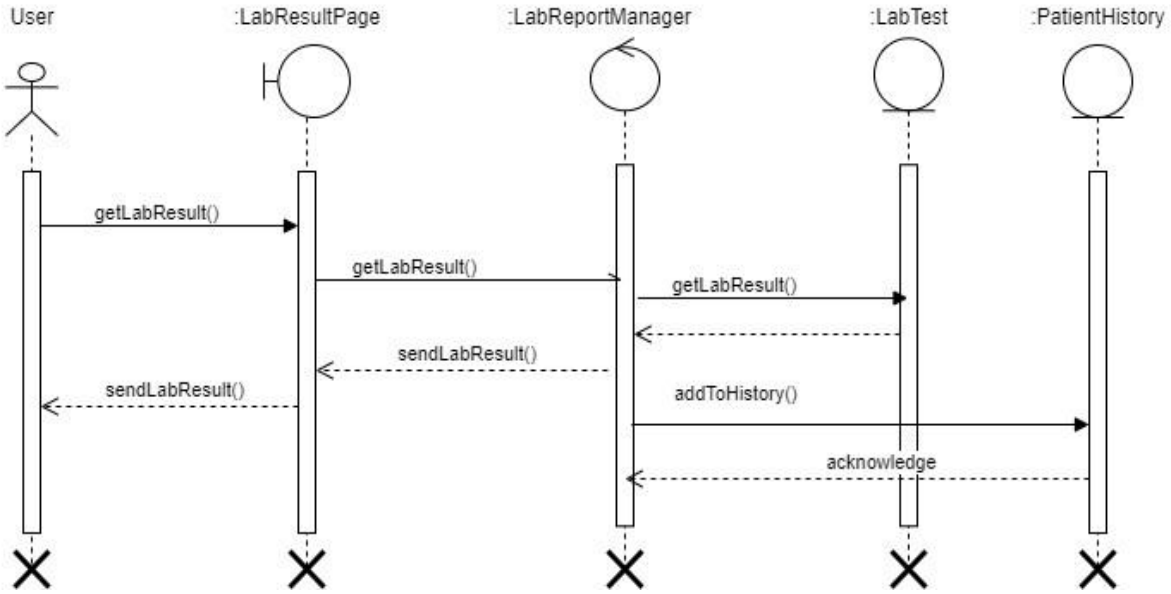

Figure 18: Check Lab result Sequence Diagram

3.2.13. Order Checkup

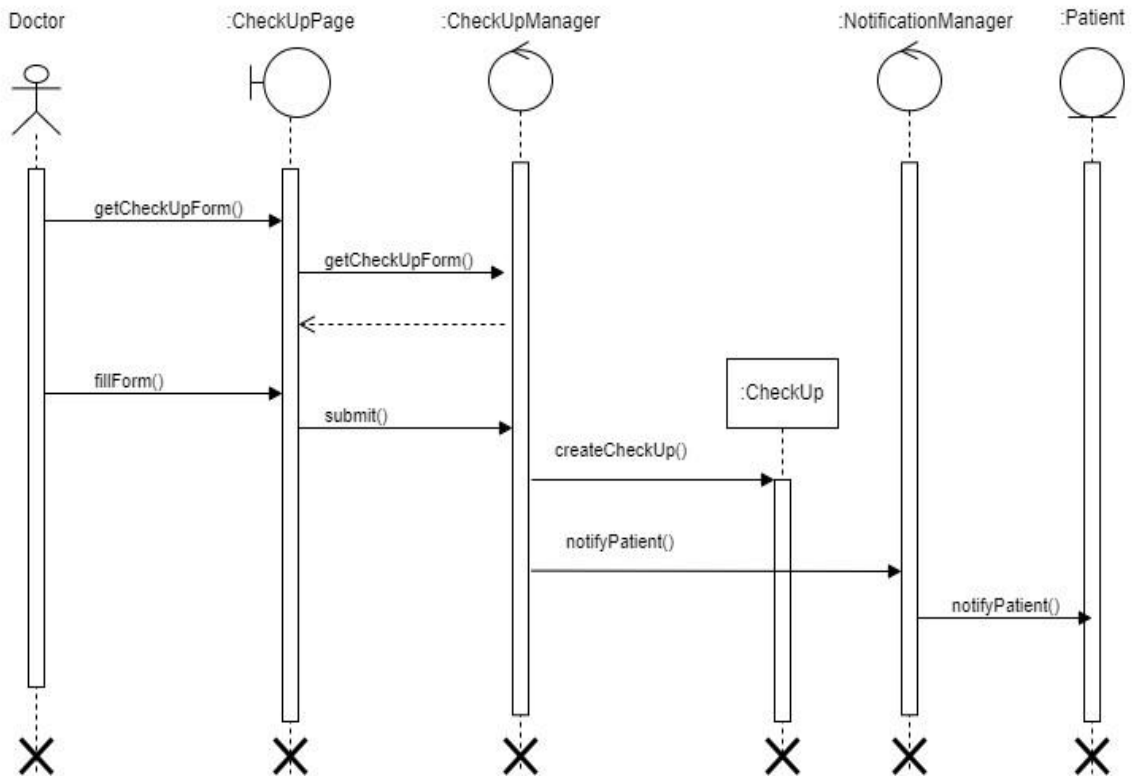

Figure 19: Order Checkup Sequence Diagram

3.3. State chart Diagram

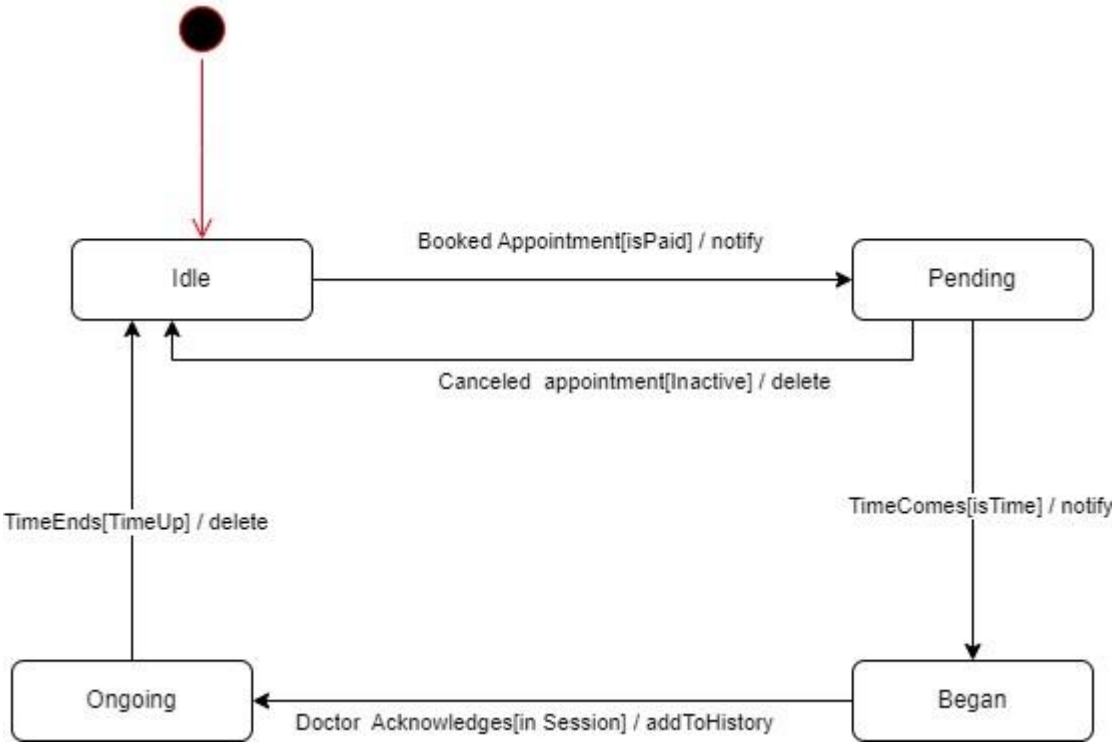

Figure 20: State Chart Diagram for Appointment Object

4. Detailed Design

Detailed design is an essential step in software development that focuses on providing a comprehensive and precise description of the structure and behavior of individual classes in a system. This design phase delves into the attributes and operations of a class, offering a thorough understanding of how it should function and interact with other components.

In a detailed design, attributes represent the state or data elements associated with a class. These attributes capture the characteristics or properties of an object instantiated from the class. They can be defined using appropriate data types, visibility modifiers (such as private or public), and accessors (such as getters and setters). The detailed design should specify the purpose, constraints, and relationships of each attribute within the class.

Operations, on the other hand, define the behavior or functionality of a class. They represent the actions or methods that can be performed on an object instantiated from the class. Operations encapsulate the algorithms, computations, or transformations that the class can perform. The detailed design should outline the input parameters, return types, preconditions, post conditions, and any exceptions associated with each operation.

4.1. User Class

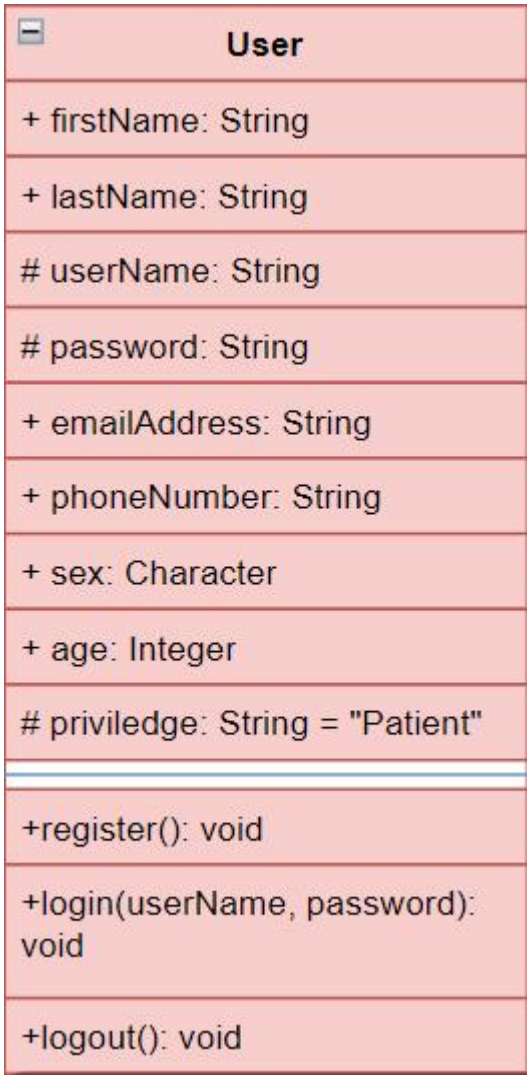

Figure 21: User Class

4.1.1. Operation description for User class

Attribute	Type	Visibility	Invariant
firstName	String	Public	firstName <> NULL and shouldn't contain special characters and integers.
lastName	String	Public	lastName <> NULL and shouldn't contain special characters and integers.
userName	String	Protected	userName <> NULL and shouldn't contain special characters with the exception of '_' .
password	String	Protected	password <>NULL and it must be more than 7 characters with a minimum of one capital letter and one number.
emailAddress	String	Public	emailAddress <> NULL ✓Must contain @ ✓Must contain. (dot) ✓Position of @ >1 ✓Position of (dot)>position of @ + 2 ✓Position of (dot)+3<= total length of email address and the total character of the Email is at least 5 characters
phoneNumber:	String	Public	phoneNumber<> NULL and must be 10 digits and must start by +251/09
sex	Character	Private	sex <> NULL and must be one character either 'M' or 'F'
age	Integer	Public	age <> NULL and a positive value
privilege	String	Protected	privilege <>Null and the user must have registered.

Table 1: Attribute description for User class

4.1.2. Operation description for User class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
login()	Public	void	-username -email -password	User must have first registered.	The user is logged into the system
logout()	Public	void		User must have first logged in.	User no longer exists in the system.

register()	public	void		User should provide valid arguments.	New user account is added to the system.

Table 2: Operation description for User class

4.2. MedicalStaff Class

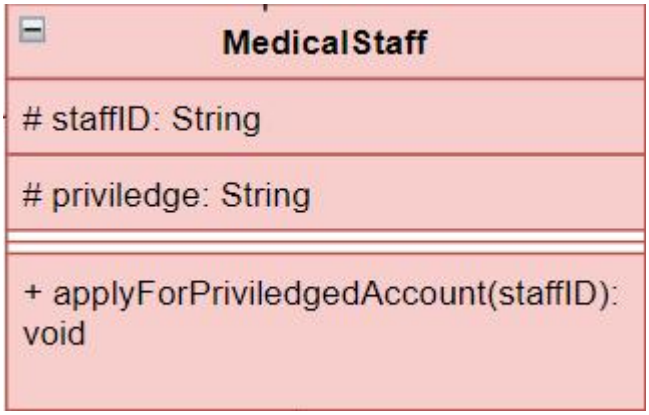

Figure 22: MedicalStaff class

4.2.1. Attribute description for MedicalStaff class

Attribute	Type	Visibility	Invariant
staffID	String	Protected	StaffID <> Null and must conform to the company ID standards
privilege	String	Protected	Privilege <>Null and the user must have registered.

Table 3: Attributes description for MedicalStaff class

4.2.2. Operation description for MediclaStaff class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition

applyForPrivilegedAccount(StaffID)	Public	void	-staffID	The account should first be registered into the system.	The request is successfully sent to admin for review .

Table 4: Operation description for MediclaStaff class

4.3. Doctor Class

Figure 23: Doctor class

4.3.1. Attributes description for Doctor class

Attribute	Type	Visibility	Invariant
specialization	String	Private	specialization <> NULL and valid specialization
dailyPateintList:	String	Private	dailyPateintList <> NULL
rating	Double	Public	rating <> NULL and a valid rating value
workHours	Time	Private	workHours <> NLL

Table 5: Attributes description for Doctor class

4.3.2. Operation description for Doctor class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
orderLabTest()	Public	:Lab Test	-patient ID -doctorID -appointmentID	The patient is diagnosed.	Lab test request is successfully sent to the admin.
orderCheckup()	public	:Checkup	-patient ID -doctorID -appointmentID		
editVisitHistory()	public	void	:PatientHistory	An existence of a visit History	Visit history is updated.
getVisitHistory()	Public	:Patient History	- :PatientHistory -visitDate	An existence of a visit History	A visit history is displayed.
getDailyPateintHistory()	Public	String		An existence of a daily patient History.	A daily patient history is displayed.
editDailyPateintHistory()	Public	Void		An existence of a daily patient History.	Daily patient history is updated.

Table 6: Operation description for Doctor class

4.4. Class LabTechnician

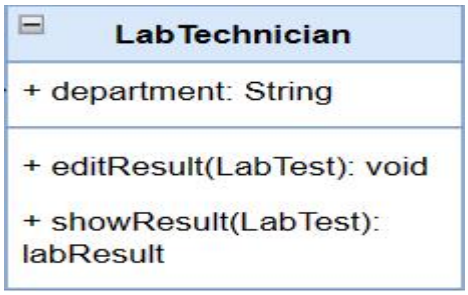

Figure 24: LabTechnician class

4.4.1. Attributes description for LabTechnician class

Attribute	Type	Visibility	Invariant
department	string	Public	Department<>NULL and the request for a privileged account should be accepted.

Table 7: Attributes description for LabTechnician class

4.4.2. Operation description for LabTechnician class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
editLabTest()	Public	void	:LabTest	Users must have first registered.	The user is logged into the system
showResult()	public	pdf	:LabTest	The test should first exist.	A valid test result.

Table 8: Operation description for Lab technicians class

4.5. Class Patient

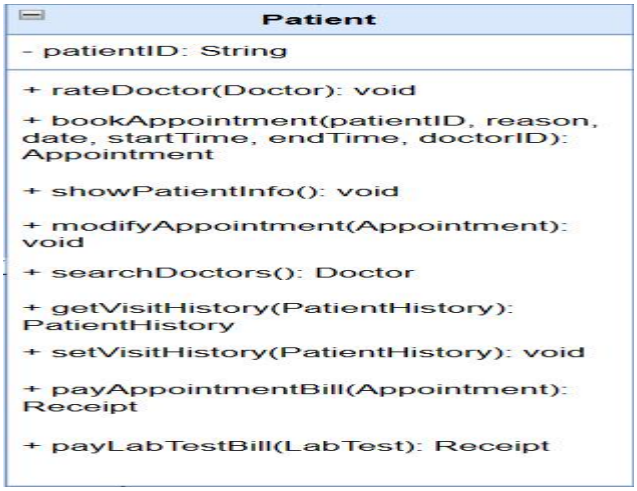

Figure 25: Patient

4.5.1. Attributes description for Patient class

Attribute	Type	Visibility	Invariant
PatientID	String	Private	

Table 9: Attributes description for Patient class

4.5.2. Operation description for Patient class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
rateDoctor()	Public	void	:Doctor		
bookAppointment()	public	Appointment	-patientID -reason -startTime -endTime -doctorID	Payed Bill	An appointment is booked.
showpatientInfo()	public	void	-patientID	The patient should exist in the	A valid patient information is

				system.	displayed.
modifyAppointment()	Public	void	- appointmentID	An appointment should exist.	The appointment is successfully updated.
searchDoctors()	Public	Doctor	- firstName	The user should be in the search doctor interface.	The Doctor is either displayed or not based on its existence in the system.
getVisitHistory()	Public	:Patient History	:Patient History	The Patient's existence in the system.	Visit History is successfully displayed.
setVisitHistory()	Public	void	:Patient History	The Patient's existence in the system.	The visit History is updated.
payappointmentBill()	Public	:Receipt	:Appointment	The Patient's existence in the system.	An appointment is booked.
payLabTestBill()	Public	:Receipt	:LabTest	The Patient's existence in the system	An available receipt

Table 10: Operation description for Patient class

4.6. Class Administrator

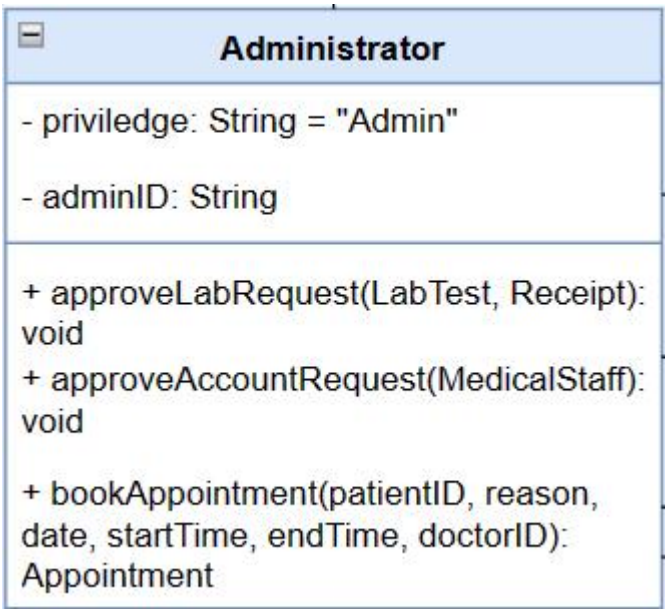

Figure 26: Administrator

4.6.1. Attributes description for Administrator class

Attribute	Type	Visibility	Invariant
Privilege	string	private	Privilege <>Null and the user must have registered.
adminID	String	Private	

Table 11: Attributes description for Administrator class

4.6.2. Operation description for Administrator class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
approveLabrequest()	Public	void	:LabTest :Receipt	A lab request.	The lab request is successfully sent to lab

					technician.
approveAccountRequest()	public	Void	:MedicalStaff	A request should be sent.	The account is either approved or rejected.
bookAppointment()	public	:Appointment	-patientID -reason -startTime -endTime -doctorID	Payed Bill	An appointment is booked.

Table 12: Operation description for Administrator class

4.7. Class Lab Test

Figure 27:LabTest

4.7.1. Attributes description for LabTest class

Table 13: Attributes description for LabTest class

Attribute	Type	Visibility	Invariant
patientID	String	Private	patientID<> NULL
doctorID	String	Private	doctorID<> NULL
appointmentID	String	Private	appointmentID<> NULL
labresult	pdf	Private	
date	Date	Private	Date<> NULL
approved	Boolean	Private	Approved <> NULL and a lab test request must exist.
approverID	String	Private	approverID <>NULL

4.7.2. Operation description for LabTest class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
editLabTest()	Public	void	- change the lab result	Users must have first registered.	The user is logged into the system
approve()	public	void		-a request must be sent - the bill must be payed	-the request is successfully accepted.
showResult()	public	Lab result	-patientID	The lab result should exist.	A valid test result.
payBill()	Public	:Receipt	-payer ID -reason	-valid bill status -sufficient fund	An available receipt

Table 14: Operation description for LabTest class

4.8. Class Receipt

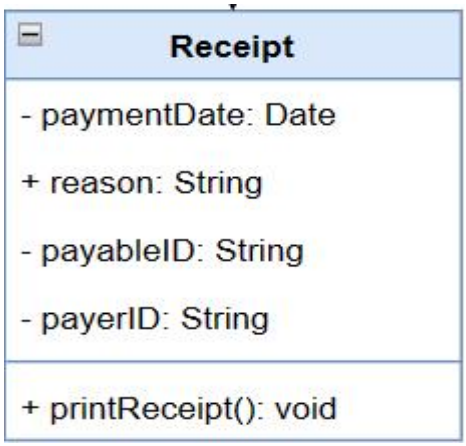

Figure 28: Receipt

4.8.1. Attributes description for Receipt class

Attribute	Type	Visibility	Invariant
paymentDate;	Date	Private	Date <> NULL
reason :	String	Public	Reason <> NULL
payableID:	String	Private	payableID <> NULL
payerID:	String	Private	payerID <> NULL

Table 15: Attributes description for Receipt class

4.8.2. Operation description for Receipt class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
printReceipt()	Public	void		The payment should be performed.	An available receipt.

Table 16: Operation description for Receipt class

4.9. Class Checkup

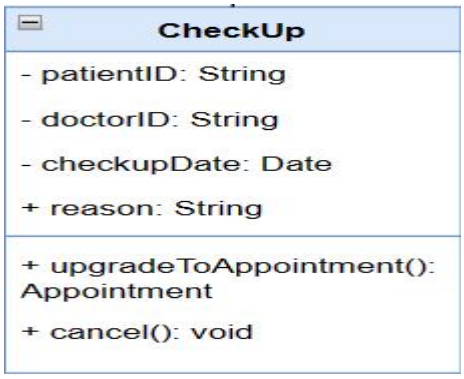

Figure 29: Checkup

4.9.1. Attributes description for Checkup class

Attribute	Type	Visibility	Invariant
patientID:	string	Private	patientID <> NULL
doctorID:	String	Private	doctorID <> NULL
checkupDate:	Date	Private	Date <> NULL
reason:	string	public	Reason <> NULL

Table 17: Attributes description for Checkup class

4.9.2. Operation description for Checkup class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
upgradeToAppointment()	Public	Appointment		An existence of a checkup.	Checkup updated
cancel()	Public	void		An existence of an appointment.	The appointment is canceled.

Table 18: Operation description for Checkup class

4.10. Class PatientHistory

Figure 30: PatientHistory

4.10.1. Attributes description for PatientHistory class

Attribute	Type	Visibility	Invariant
visitDate:	Date	Private	Date <> NULL
appointmentID:	String	Private	appointmentID <> NULL
visitReason:	String	Private	visitReason <> NULL
patientID:	String	Private	patientID <>NULL
doctorID:	String	Private	doctorID <> NULL
diagnosisResult:	String	Private	diagnosisResult <> NULL

Table 19: Attributes description for PatientHistory class

4.10.2. Operation description for PatientHistory class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
getVisitHistory()	public	Patient History	-patientID - appointmentID	An existence of a visit History	A visit history is displayed.

setVisitHistory()	Public	void	-patientID - appointmentID	An existence of a visit History	Visit history is update.

Table 20: Operation description for PatientHistory class

4.11. Appointment Class

Figure 31: Appointment

4.11.1. Attributes description for Appointment class

Attribute	Type	Visibility	Invariant
Date:	Date	Private	Date <> NULL
startTime:	Time	Private	startTime <> NULL
endTime:	Time	Private	endTime <> NULL

patientID:	String	Private	patientID <> NULL
doctorID:	String	Private	doctorID <> NULL
appointmentID	String	Private	appointmentID <> NULL

Table 21: Attributes description for Appointment class

4.11.2. Operation description for Appointment class

Operation	Visibility	Return type	Argument	Pre-Condition	Post Condition
payBill()	public	Receipt	-patientID	The Patient's existence in the system	An available receipt
modifyAppointment()	Public	void	-appointmentID	An appointment should exist.	The appointment is successfully updated.

Table 22: Operation description for Appointment class

References

Bibliography

1. Cayirli, T, E. Veral, and H. Rosen. (2006). *"Designing appointment scheduling systems for ambulatory care services."* Health Care Management Science 9, 47–58.

2. Clue, R. (2020). *"Design And Implementation Of Appointment Management System"* (A CASE STUDY OF UNITHECH HOSPITAL). From Research-Clue: researchClue.com

3. MLA. Fowler, Martin, 1963-. *UML Distilled : a Brief Guide to the Standard Object Modeling Language*. Reading, Mass. :Addison-Wesley, 2000.

Web Resource

1. https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/ at Nov 1, 2023.

2. http://www.tutorialspoint.com/uml/index.htm at May 5,2020.