Hospital Management System

A Report Submitted
In Partial Fulfillment
for award of Bachelor of Technology/Master of Integrated Technology

In
COMPUTER SCIENCE AND ENGINEERING
DATA SCIENCE

By

KASHIF HUSSAIN (Roll No. 2401331540136)
GAURAV GUPTA (Roll No. 2401331540115)
DEENDAYAL (Roll No. 2401331540097)

Under the Supervision of
Mr. Chandrapal Singh Arya
Assistant Professor, Department of Computer Science & Engineering
Data Science

Noida Institute of Engineering Technology
19, Knowledge Park- II, Institutional Area,
Greater Noida (UP) - 201306

DECLARATION

I hereby declare that the work presented in this report was carried out by me. I have not submitted the matter embodied in this report for the award of any other degree or diploma of any other University or Institute.

KASHIF HUSSAIN (Roll No. 2401331540136)

GAURAV GUPTA (Roll No. 2401331540115)

DEENDAYAL (Roll No. 2401331540097)

CERTIFICATE

Certified that Kashif Hussain (Roll No: 2401331540136) , Gaurav Gupta (Roll No : 2401331540115) , Deendayal (Roll No : 2401331540097) has carried out the Advanced Java project work presented in this Project Report at Noida Institute of Engineering and Technology in partial fulfilment of the requirements for the award of Bachelor of Technology, Computer Science Engineering (Data Science) from Dr. APJ Abdul Kalam Technical University, Lucknow under our supervision.

Signature

Supervisor:
Mr. Chandrapal Singh Arya
Assistant Professor
Department of CSE(DS)
NIET Greater Noida

Date:

Signature

Head of Department:
Dr. Ashish Chakraverti
Head of Department
Department of CSE(DS)
NIET Greater Noida

ACKNOWLEDGEMENT

I would like to express my gratitude towards Mr. Chandrapal Singh Arya for their guidance, support and constant supervision as well as for providing necessary information during my internship.

My thanks and appreciations to respected HOD, for their motivation and support throughout.

ABSTRACT

Hospital management involves handling lots of paperwork, patient records, doctor schedules, appointments, prescriptions, and bills. Many small hospitals still use manual systems which are slow, error-prone, and difficult to manage. Our project, Ojasvi Hospital Management System, solves these problems by creating a simple computer application using Java.

The system has six main sections - a dashboard showing overall statistics, patient registration, doctor information, appointment booking, prescription writing, and billing with payment tracking. We built everything using Java Swing for the graphical interface and used object-oriented programming concepts like inheritance, polymorphism, and encapsulation.

The application is easy to use and doesn't need any database setup. It runs on any computer with Java installed - whether Windows, Linux, or Mac. We pre-loaded information about 5 patients and 7 doctors to demonstrate how it works. The system automatically generates unique IDs for patients, doctors, appointments, prescriptions, and bills.

We tested everything thoroughly and all features work properly. The main advantage is that it's completely free, easy to install, and doesn't require technical knowledge to operate. However, it currently stores data in memory only, so information is lost when you close the application. In future, we plan to add database support, login system, and more advanced features.

This project shows how programming knowledge can solve real-world problems in healthcare management. It's practical, functional, and ready to use in small clinics or hospitals.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Introduction

Managing a hospital involves handling many things at once - patient records, doctor schedules, appointments, medicines, and bills. In many small hospitals and clinics, all this work is still done manually using paper registers and files. This creates several problems. Records get lost, handwriting is hard to read, finding information takes time, and mistakes happen frequently.

With computers becoming common everywhere, hospitals can benefit greatly from using software to manage their daily operations. A good hospital management system can make work faster, reduce errors, and help staff focus more on patient care rather than paperwork.

We developed the Ojasvi Hospital Management System to address these challenges. It's a desktop application built using Java programming language. The system provides a simple way to register patients, view doctor information, book appointments, create digital prescriptions, and generate bills. Everything is connected, so when you add a new patient, they immediately appear in the appointment booking section.

The best part is that it doesn't need any complicated setup. You just need Java installed on your computer, and the application runs smoothly on Windows, Linux, or Mac. We designed it to keep in mind that hospital staff may not be very comfortable with technology, so we made the interface as simple and clear as possible.

This project helped us apply what we learned in our Java programming course to solve a real problem. It covers important programming concepts like object-oriented design, graphical user interfaces, data management, and error handling.

1.2 Background

Hospitals have been using computers for management since the 1960s, but early systems were very expensive and complex. Only big hospitals could afford them. Over time, technology improved and became cheaper, but many small hospitals still struggle to implement management systems.

The main reasons are cost and complexity. Commercial hospital management software can cost lakhs of rupees for licensing and requires servers, databases, and IT staff to maintain. Open source alternatives exist but they also need technical expertise to install and configure.

During our workshop on Java programming, we learned about creating desktop applications using Swing framework. We also studied how to structure programs using object-oriented principles. Around the same time, we visited a local clinic and saw how they manage everything manually in big registers. The receptionist told us it takes a lot of time to find old patient records and sometimes appointments get mixed up.

This gave us the idea to build a simple hospital management system as our project. We wanted to create something that doesn't need database installation or server setup. The application should be standalone - just one file that you can run on any computer. We also wanted to make it user-friendly so that staff with basic computer knowledge can use it easily.

We researched existing systems and found that most of them are either too complicated or too basic. Complicated systems have too many features that small clinics don't need. Basic systems lack essential features like billing or prescription management. We decided to create something in between - covering all essential functions but keeping each feature simple and straightforward.

Java was the perfect choice because it works on all operating systems and has excellent tools for building graphical interfaces. The Swing library provides ready-made components like buttons, text fields, and tables, which saved us a lot of development time.

1.3 Problem Statement

After talking to hospital staff and observing their daily work, we identified several key problems with manual management:

Patient Record Management Problems: When patients visit for the first time, staff writes their details in a register. These registers are big, heavy books. Finding a patient's old record means flipping through hundreds of pages. Sometimes the same patient gets registered twice with slightly different name spellings. Patient information is scattered - personal details in one register, medical history in another, and billing in a third register.

Appointment Scheduling Problems: Appointments are usually booked over phone calls. The receptionist checks the appointment register to see which slots are free. Sometimes they accidentally book two patients at the same time because the register was updated by someone else just minutes ago. There's no easy way to see how many appointments a doctor has today or which patients are coming.

Prescription Issues: Doctors write prescriptions by hand. Sometimes their handwriting is difficult to read, and patients or pharmacists misunderstand the medicine names or dosages. There's no record of what medicines were prescribed to a patient previously, which could be important for the next visit.

Billing Complications: Bills are calculated manually by adding consultation fees, medicine costs, and room charges. Arithmetic mistakes happen. Keeping track of which bills are paid and which are pending requires maintaining another register. At month-end, staff has to manually count and calculate total revenue.

General Management Issues: Hospital administrators can't quickly know how many patients visited this month, which doctor is busiest, or how much revenue was generated. Getting such information requires going through all registers and counting manually.

The Core Problem: How can we create a simple, affordable, easy-to-use computer system that helps small hospitals and clinics manage patients, appointments, prescriptions, and billing without needing expensive infrastructure or technical expertise?

1.4 Objectives

Our main goal was to build a practical hospital management system that actually solves real problems. Here's what we wanted to achieve:

Primary Objectives:

First, create a complete system covering all essential hospital operations. This includes registering patients with their medical information, maintaining doctor profiles, booking appointments between patients and doctors, generating digital prescriptions, and handling billing with payment tracking.

Second, make sure the system is genuinely easy to use. We wanted someone with basic computer skills to be able to use it without reading a manual or attending training sessions. The interface should be clear, with proper labels and logical flow.

Third, keep everything simple and lightweight. No database installation, no server setup, no complicated configuration. Just install Java and run the application.

Fourth, demonstrate good programming practices. Use object-oriented concepts properly, write clean understandable stuff, handle errors gracefully, and structure everything logically.

Learning Objectives:

Beyond solving the hospital management problem, this project was also a learning experience for our team. We wanted to understand how to design and develop a complete application from scratch - starting with identifying requirements, creating designs, writing programs, testing everything, and documenting the whole process.

We aimed to learn teamwork and collaboration. Three people working together need to divide tasks, communicate clearly, help each other, and integrate different parts into one working system.

We also wanted practical experience with Java Swing for creating graphical interfaces, ArrayList for managing data collections, exception handling for dealing with errors, and event handling for responding to user actions.

Expected Outcomes:

At the end of the project, we should have a fully working application that can be installed and used in a real clinic. The system should be reliable enough that it doesn't crash or lose data during normal operation.

We should be able to demonstrate all features working together smoothly. Adding a patient should immediately make them available for appointment booking. Generating a bill should automatically calculate totals. The dashboard should update whenever data changes.
Most importantly, the system should actually make hospital operations easier and faster compared to manual methods.

1.5 Scope of Project

What We Built:

Our system includes six main modules that work together. The Dashboard shows an overview of everything - total patients, doctors, appointments, prescriptions, bills, and pending payments. It updates automatically when anything changes.
The Patient Management module lets you register new patients with complete information including name, gender, phone number, address, blood group, and current disease or complaint. All registered patients are shown in a table where you can see their details at a glance.
The Doctor Management module displays information about available doctors including their specialization, qualifications, and consultation fees. We pre-loaded details of seven doctors across different specialties like cardiology, neurology, pediatrics, and general medicine.
The Appointment Booking module connects patients with doctors. You select a patient, choose a doctor, pick a date and time, and the system creates an appointment record. All appointments are listed with their status.
The Prescription module allows creating digital prescriptions. You select the patient and doctor, enter the medicines and dosage instructions, and the system generates a prescription record with date and unique ID.
The Billing module handles financial transactions. You create bills with different components - consultation charges, medicine costs, and room charges. The system calculates the total automatically. There's also a payment tracking feature where you can mark bills as paid when money is received.

What We Didn't Include:

To keep the project manageable within our timeline, we decided not to include certain features. There's no user login system, so anyone who opens the application can access everything. In a real deployment, you would want different access levels for administrators, doctors, and receptionists.

We didn't add database connectivity. All data is stored in computer memory while the program runs, but it's not saved permanently. When you close the application, all data is lost. This was a conscious decision to avoid the complexity of database installation and management.

There's no report generation in PDF or Excel format. The system shows data on screen but doesn't export it. Adding printing and exporting features would require additional libraries and more development time.

We also skipped advanced features like laboratory test management, pharmacy inventory, insurance processing, or medical image storage. These are important for big hospitals but not essential for small clinics that were our target users.

Future Possibilities:

Even though we didn't build these features now, we designed the system in a way that makes it easy to add them later. The program structure is modular, so new modules can be added without rewriting everything. We documented the approach well so someone else (or even we ourselves later) can extend the system.

Adding database support is the most important future enhancement. This would make data permanent and allow multiple users to access the system simultaneously over a network.

Other useful additions would include search and filter functions to find specific records quickly, backup and restore capabilities, detailed reporting and analytics, SMS or email reminders for appointments, and a web-based version accessible from browsers.

1.6 Organization of Report

This report is organized into seven chapters plus references and appendices.

Chapter 1 (Introduction) - This chapter explains what the project is about, why we chose this topic, what problems we're trying to solve, and what we hope to achieve.

Chapter 2 (Literature Review) - Here we discuss existing hospital management systems, what technologies others have used, and how our approach differs from what's already available.

Chapter 3 (System Requirements) - This chapter lists everything the system needs to work - what functions it should perform, what quality standards it should meet, what hardware is needed, and what software must be installed.

Chapter 4 (System Design) - We explain how we designed the system architecture, show diagrams of how different parts connect, and describe our design approach.

Chapter 5 (Implementation) - This chapter describes how we actually built the system, what tools we used, and how we implemented each major feature.

Chapter 6 (Testing) - We explain how we tested the system to make sure everything works correctly, what test cases we used, and what results we got.

Chapter 7 (Conclusion) - The final chapter summarizes what we achieved, acknowledges current limitations, and suggests what improvements could be made in future.

References section lists all books, websites, and other sources we consulted while working on this project.

Appendices include additional material like screenshots of the working system, a user guide explaining how to use each feature, and information about who did what in our team.

Each chapter builds on the previous one, so reading them in order gives you the complete story of how we went from identifying a problem to delivering a working solution.

CHAPTER 2: LITERATURE REVIEW

2.1 Existing Systems

Before starting our project, we researched what hospital management systems already exist. We looked at commercial products, open-source software, and academic projects.

Commercial Systems: Big hospitals use enterprise systems like Epic, Cerner, or Meditech. These are comprehensive platforms that handle everything - patient records, laboratory tests, radiology, pharmacy, insurance, billing, and much more. They cost crores of rupees and require dedicated IT teams to maintain.

We also found mid-range commercial software targeting smaller hospitals. These typically cost between 50,000 to 5,00,000 rupees depending on features and number of users. Most require annual subscription fees for updates and support.

Open Source Systems: OpenMRS is popular in developing countries. It's free but needs good technical knowledge to install and configure. You have to set up MySQL database, configure Apache Tomcat server, and understand several configuration files.

HospitalRun is a modern system built with web technologies. It works offline and online both. However, it requires installing Node.js, npm packages, and setting up CouchDB database. The installation process itself is quite technical.

OpenEMR is another well-known system. It's feature-rich and has been around for years. But again, it needs LAMP stack (Linux, Apache, MySQL, PHP) or similar setup.

Academic Projects: We found several final year projects from other colleges. Most were web-based using PHP and MySQL, or desktop applications using Java with SQL Server or Oracle databases. Some used Android for mobile access.

Common Pattern: Almost all existing systems, whether commercial or free, assume you have or can set up database infrastructure. They're designed for either very large hospitals with IT departments, or for users with technical knowledge.

What We Noticed: There's a gap between simple manual systems and complex computerized systems. Small clinics with 1-2 doctors and basic computer knowledge don't have a good middle-ground option. They need something simpler than what currently exists.

2.2 Technologies Used

Different hospital management systems use different technologies. Understanding these helped us make informed choices for our project.

Programming Languages: Web-based systems commonly use PHP, Python (Django/Flask), or JavaScript (Node.js) for the backend. Desktop applications are usually built with Java, C#, or C++.

Databases: MySQL and PostgreSQL are popular for open-source projects. Commercial systems often use Oracle or SQL Server. Some modern systems use NoSQL databases like MongoDB or CouchDB.

User Interface: Web systems use HTML, CSS, and JavaScript frameworks like React, Angular, or Vue. Desktop applications use platform-specific frameworks - Java Swing, Windows Forms, or Qt for C++.

Why We Chose Java: Java fits our requirements perfectly for several reasons. First, it's platform-independent - the same program runs on Windows, Linux, and Mac without any changes. Second, we're already familiar with Java from our coursework. Third, Java Swing provides everything needed to build desktop interfaces without additional libraries.

Why No Database: This was our most controversial decision. Everyone assumes management systems need databases. We decided against it because:

- Database installation and configuration is a significant barrier for non-technical users
- Small clinics processing 20-30 patients daily don't really need database power
- We could demonstrate all core concepts without database complexity
- Keeping everything in memory makes the application faster and simpler

We know this limits the system's practical usability, but for a learning project demonstrating programming concepts, it makes sense. Adding database support later is straightforward if needed.

Object-Oriented Programming: We used OOP principles throughout. Patient and Doctor classes inherit from a Person base class. Appointment and Prescription classes create associations between patients and doctors. The Bill class demonstrates composition by containing patient data.

This isn't just academic exercise - OOP makes the program much easier to understand, maintain, and extend.

Swing Framework: Java Swing gave us components like JFrame for windows, JPanel for organizing content, JTable for displaying data, JTextField for input, JButton for actions, and JComboBox for dropdowns. The layout managers (BorderLayout, GridLayout) helped arrange everything neatly.

2.3 Comparison of Systems

Let's compare different approaches to hospital management:

Enterprise Systems vs Our System: Enterprise systems handle everything a large hospital needs - from emergency department management to surgical scheduling to insurance claims. They integrate with laboratory equipment, radiology machines, and pharmacy systems. They cost crores and take months to deploy.

Our system handles core administrative functions - patient registration, appointments, prescriptions, and billing. It costs nothing and can be deployed in minutes. Obviously, enterprise systems are far more capable, but they're complete overkill for a small clinic.

Open Source Systems vs Our System: Open-source HMS like OpenMRS are free and feature-rich. They have active communities and regular updates. But they require technical setup - database installation, server configuration, dependency management.

Our system is also free but needs zero setup beyond Java installation. However, it lacks many features that open-source systems provide, like multi-user access, permanent data storage, and advanced reporting.

Other Student Projects vs Our System: Most student projects we found implemented 3-4 basic features - patient registration and maybe appointments or billing. Few had complete, integrated systems. Many had bugs or incomplete error handling.

We focused on building a complete system where all parts work together smoothly. A patient added in one module immediately appears in other modules. The dashboard updates automatically. Error messages are clear and helpful. Everything is tested and polished.

The Trade-off: We sacrificed advanced features and permanent storage for simplicity and ease of use. For our target audience (small clinics) and our goal (demonstrating programming concepts), this trade-off makes sense.

2.4 Identified Gaps

Based on our research, we identified several gaps that our system addresses:

Simplicity Gap: Most systems assume users have technical knowledge or IT support. They include features that require understanding of medical coding, insurance processes, or database concepts. Small clinic staff just want to register patients and track appointments without dealing with complexity.

Our system has a clean, simple interface. Each module does one thing clearly. No medical jargon unless necessary. No technical configuration needed.

Cost Gap: Commercial systems are too expensive for small clinics. Even "affordable" options cost 50,000 rupees or more, which is a significant investment for a single-doctor clinic earning modest income.

Our system is completely free. No licensing fees, no subscription costs, no per-user charges. Anyone can download and use it.

Technical Barrier Gap: Open-source systems save money but create a technical barrier. Installing MySQL, configuring PHP, setting up web servers - these are not simple tasks for someone without IT background.

Our system has almost no technical barrier. If Java is installed (which is straightforward), the application just runs. No configuration files to edit, no database to set up, no dependencies to resolve.

Educational Gap: Most existing systems are either too simple (just basic forms) or too complex (enterprise-level architectures) for learning purposes. Students need something in between - complex enough to demonstrate important concepts, simple enough to understand completely.

Our system demonstrates inheritance, polymorphism, encapsulation, GUI development, event handling, exception management, and data structures - all important topics in Java programming courses. Yet it's small enough that one can understand the entire system.

Integration Gap: Many hospital systems are actually collections of separate modules that don't integrate well. Patient data entered in registration doesn't automatically flow to billing. Appointment scheduling doesn't connect with prescription writing.

We designed our system with integration in mind from the beginning. All modules share data seamlessly. Everything updates in real-time.

User Experience Gap: Many systems have cluttered interfaces with too many options visible at once. Users need training to figure out where things are and how to do common tasks.

We used simple tabbed interface. Each major function has its own clear section. Related information appears together. Common tasks require few clicks.

CHAPTER 3: SYSTEM REQUIREMENTS

3.1 Functional Requirements

Functional requirements describe what the system should do. Here's what our hospital management system needs to accomplish:

Patient Management: The system must allow registering new patients with their basic details - full name, gender, contact phone number, residential address, blood group, and current health complaint or disease. Each patient should get a unique ID automatically, like P001, P002, and so on. The system should display all registered patients in an organized table. Registration should validate that name and phone number are provided since these are essential fields.

Doctor Information: The system must maintain and display information about available doctors. For each doctor, we need their name, gender, specialization (like cardiologist or pediatrician), educational qualifications, and consultation fee. Doctors should also have unique IDs - D001, D002, etc. Since doctor information doesn't change frequently, viewing is more important than editing.

Appointment Scheduling: The system must enable booking appointments by connecting patients with doctors. Users should be able to select a patient from the list, choose an available doctor, specify date and time, and create an appointment record. Each appointment gets a unique ID and is initially marked as "Scheduled". The system must show all appointments in a table with complete details. Importantly, when new patients are added, they should immediately appear in the patient selection dropdown.

Prescription Generation: The system must support creating digital prescriptions. A prescription links a specific patient with a specific doctor and contains the prescribed medicines plus dosage instructions. Each prescription gets a unique ID starting with RX (like RX001). The system should record when each prescription was created and display all prescriptions clearly.

Billing and Payment: The system must generate itemized bills for patients. A bill includes three components - consultation charges, medicine charges, and room charges. The system automatically calculates and displays the total amount. Each bill gets a unique ID (B001, B002, etc.) and tracks payment status. Initially, bills are marked "Unpaid". The system must provide a way to mark bills as "Paid" when payment is received. It should prevent marking the same bill as paid multiple times.

Dashboard Overview: The system must provide a dashboard showing key statistics at a glance - total number of patients, total doctors, total appointments scheduled, prescriptions written, bills generated, and pending bills awaiting payment. These numbers should update automatically whenever relevant data changes.

General Requirements: The system should organize different functions into separate, clearly labeled sections for easy navigation. It must provide clear success or error messages for every

operation so users know what happened. All data should remain consistent across different modules - if you add a patient, that patient should be accessible everywhere else immediately.

3.2 Non-Functional Requirements

Non-functional requirements describe how well the system should work:

Usability: The system must be intuitive enough that someone with basic computer skills can use it without training. Forms should have clear labels and logical layouts. Error messages should explain what went wrong in simple language and suggest how to fix it. Common tasks like adding a patient or booking an appointment should require only a few clicks. The interface should not overwhelm users with too many options at once.

Performance: The application should start within a few seconds on a standard computer. When users submit forms or click buttons, the response should be immediate - under one second. Loading and displaying data in tables should feel instant for reasonable amounts of data (a few hundred records). The dashboard should update without noticeable delay.

Reliability: The system must handle incorrect inputs gracefully without crashing. If someone enters letters where numbers are expected, or leaves required fields empty, the system should show helpful error messages rather than shutting down. Data should remain consistent even if users perform actions in unexpected orders.

Compatibility: The application must run on Windows, Linux, and macOS without any modification. It should work with any Java version 8 or newer. The interface should display correctly on different screen sizes and resolutions, though optimized for standard desktop displays (1024x768 or higher).

Maintainability: The program structure should be clear and well-organized so that future modifications are easy. We should use proper object-oriented design with logical separation between different parts. Adding new features shouldn't require rewriting large portions of existing code.

Security: While we don't have a login system, the application should validate all inputs to prevent obviously wrong or malicious data entry. It should use exception handling to catch errors before they cause problems.

3.3 Hardware Requirements

The system doesn't need powerful hardware. Here's what's required:

Minimum Requirements: Any computer with an Intel Core i3 processor or AMD equivalent is sufficient. The processor should be at least 2.0 GHz. You need 4 GB of RAM - this is standard in most computers sold in the last 5-6 years. For storage, just 100 MB of free hard disk space is enough for the application itself. The display should support at least 1024x768 resolution, though higher is better.

You'll need a standard keyboard and mouse for input. That's it - no special hardware required.

Recommended Setup: For better performance, an Intel Core i5 processor or better is recommended. 8 GB of RAM makes the computer more comfortable for multitasking. If you have 500 MB free space, that leaves room for data and temporary files. A Full HD display (1920x1080) provides more screen space and makes the interface more comfortable to use.

Why Requirements Are Low: Since we store data in memory rather than database, and the interface is simple, the application doesn't need much computing power. Most computers currently used in small clinics can easily run this system.

3.4 Software Requirements

Operating System: Windows 7, 8, 10, or 11 - all work fine. Most clinic computers run Windows, so this is the most common scenario. Linux distributions like Ubuntu 18.04 or newer also work. For Mac users, macOS 10.13 (High Sierra) or newer is required.

Java Environment: This is the only essential software requirement. You need Java Runtime Environment (JRE) version 8 or higher installed. Most computers already have Java installed. If not, it's free to download from Oracle's website and installation is straightforward.

For development work (if someone wants to modify the code), Java Development Kit (JDK) version 8 or higher is needed.

No Database Required: Unlike most hospital systems, ours doesn't need MySQL, PostgreSQL, Oracle, or any database software. This significantly simplifies installation and usage.

No Web Server Needed: Since it's a desktop application, there's no need for Apache, Tomcat, or any web server software.

No Additional Libraries: The application uses only standard Java libraries that come with Java installation. No need to download or install additional JAR files, dependencies, or frameworks.

Development Tools (Optional): During development, we used IntelliJ IDEA and Eclipse as programming environments, but these are not needed for running the application. For documentation, standard tools like Microsoft Word and simple diagram drawing software were sufficient.

Why So Simple: We deliberately kept software requirements minimal to reduce barriers to adoption. The only thing users need to worry about is having Java installed. Everything else is already included in the application.

CHAPTER 4: SYSTEM DESIGN

4.1 System Architecture

Our system follows a simple three-layer architecture:

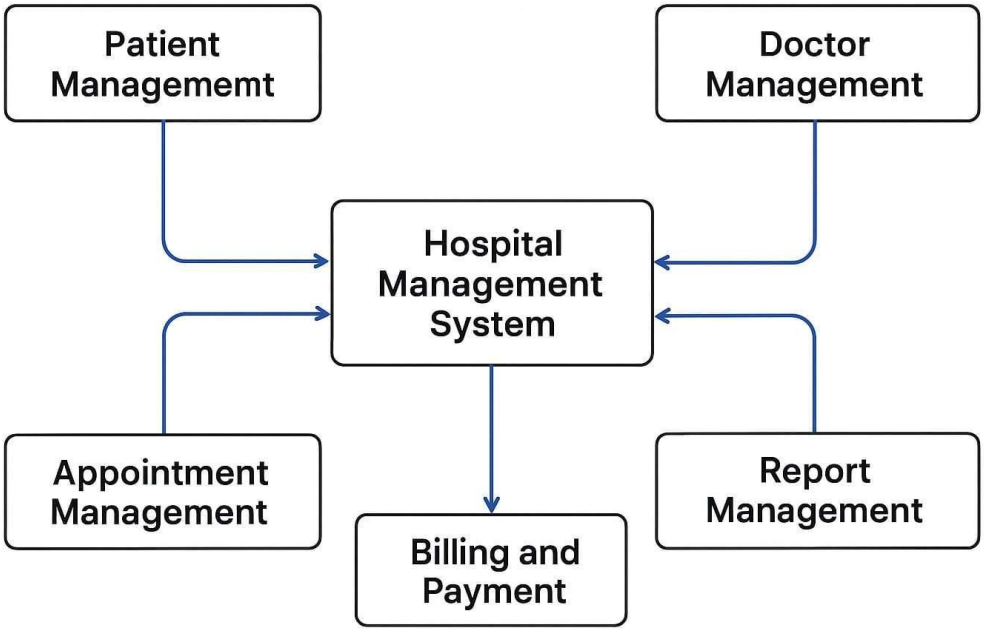

Figure 4.1 System Architecture

Presentation Layer: This is what users see and interact with. It includes all the windows, buttons, forms, and tables. We built this using Java Swing components. The main window has tabs for different modules - Dashboard, Patients, Doctors, Appointments, Prescriptions, and Billing. Each tab shows appropriate forms and tables for that function.

Business Logic Layer: This is the middle layer that processes everything. When a user clicks "Add Patient" or "Book Appointment," this layer handles the action. It validates the data, generates unique IDs, creates objects, updates collections, and refreshes the display. All the rules and logic for how the system should work are implemented here.

Data Layer: This layer manages all the information. We use ArrayList collections to store patients, doctors, appointments, prescriptions, and bills. When the application starts, it loads sample data. During operation, it keeps everything in memory and provides methods to add, retrieve, and update data.

These three layers work together smoothly. When a user fills a form and clicks save, the presentation layer captures the input, sends it to the business logic layer which validates and

processes it, then updates the data layer, and finally refreshes the presentation layer to show the new data.

Why This Architecture: This separation makes the program easier to understand and maintain. If we want to change how data is stored (like adding a database later), we only modify the data layer. If we want to improve the interface, we work on the presentation layer. The business logic remains independent.

4.2 Design Approach

We used object-oriented design throughout the project. Here's how:

Inheritance: We created an abstract Person class containing common attributes like ID, name, phone, address, and gender. Then Patient and Doctor classes inherit from Person, adding their specific attributes. Patient adds blood group and disease, while Doctor adds specialization, qualification, and consultation fee. This avoided repeating the same code in both classes.

Encapsulation: All class attributes are private. We provide public methods (getters) to access them. For example, to get a patient's name, you call getName() rather than directly accessing the name variable. This protects data from accidental changes and lets us control how it's accessed.

Polymorphism: The Person class has an abstract method getRole(). Patient class implements it to return "Patient" and Doctor class returns "Doctor". This means we can treat both patients and doctors as Person objects when needed, but each knows its own role.

Association and Composition: Appointment class creates an association between Patient and Doctor - it links them together with a date and time. Prescription class similarly associates a patient, doctor, medicines, and instructions. Bill class demonstrates composition by containing patient information along with charges and payment status.

Exception Handling: We created a custom HospitalException class for handling errors specific to our application. When something goes wrong (like trying to add a patient without a name), the system throws this exception with a clear message, which gets caught and displayed to the user.

Collections Framework: We used ArrayList to store all our data. Patients go in ArrayList<Patient>, doctors in ArrayList<Doctor>, and so on. ArrayList provides useful methods like add, get, remove, and size which made data management straightforward.

4.3 Use Case Diagram

A use case diagram shows who uses the system and what they can do:

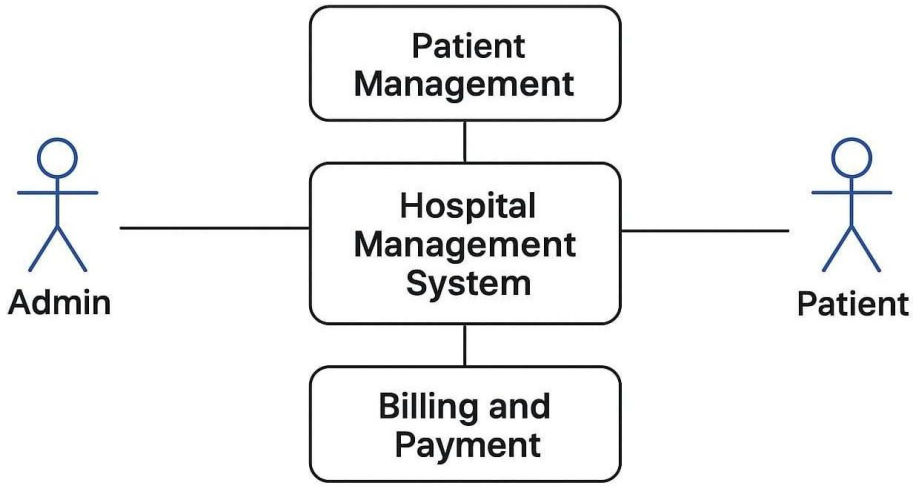

Figure 4.2 Use Case Diagram

Actors: Our system has three main types of users:

Hospital Administrator manages the overall system. They can register patients, view doctor information, book appointments, create prescriptions, generate bills, and process payments. They have access to everything.

Receptionist primarily handles patient registration, appointment booking, and billing. They're the first point of contact when patients visit.

Doctor focuses on medical aspects - viewing patient information, checking appointments, and writing prescriptions.

Main Use Cases:

Register Patient: Administrator or Receptionist can add new patients to the system with all necessary details.

View Doctors: All users can see the list of available doctors with their specializations.

Book Appointment: Administrator or Receptionist selects a patient and doctor, then schedules an appointment.

Create Prescription: Doctor selects patient and enters medicines and instructions.

Generate Bill: Administrator or Receptionist creates an itemized bill for services provided.

Process Payment: When payment is received, the user marks the bill as paid.

View Dashboard: All users can see the overview statistics.

These use cases cover the complete workflow from patient registration through treatment to payment collection.

4.4 Class Diagram

The class diagram shows how different classes relate to each other:

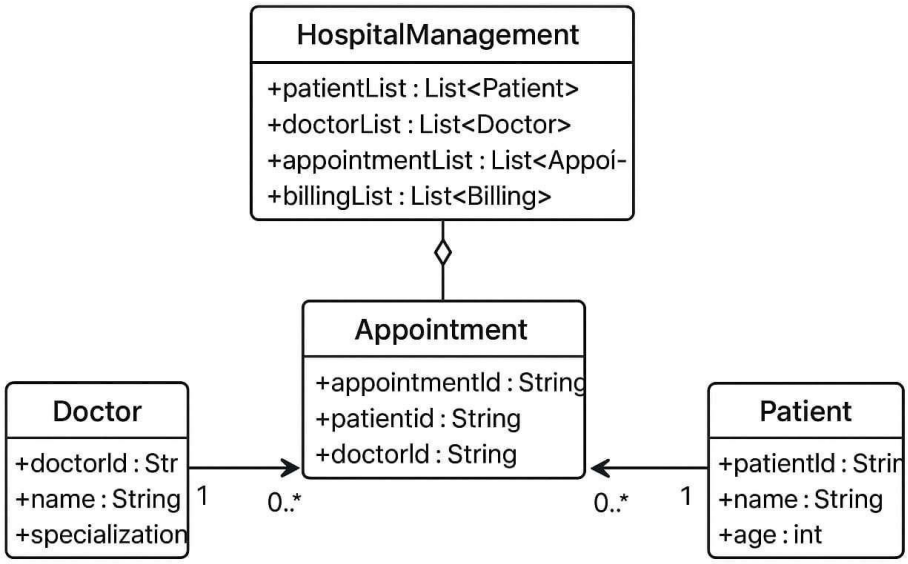

Figure 4.3 Class Diagram

Person Class (Abstract): This is the base class with id, name, phone, address, and gender as attributes. It has an abstract method getRole() and concrete methods getId(), getName(), etc.

Patient Class: Extends Person and adds bloodGroup, disease, and admissionDate. Implements getRole() to return "Patient". Has additional methods like getBloodGroup() and getDisease().

Doctor Class: Also extends Person and adds specialization, qualification, and consultationFee. Implements getRole() to return "Doctor". Provides methods to get these additional attributes.

Appointment Class: Has appointmentId, patient (Patient object), doctor (Doctor object), date, time, and status. This class creates association between Patient and Doctor classes. Methods allow getting appointment details and updating status.

Prescription Class: Contains prescriptionId, patient, doctor, medicines, instructions, and date. Similar to Appointment, it associates patient and doctor but for prescription purposes.

Bill Class: Has billId, patient, consultationCharges, medicineCharges, roomCharges, totalAmount, billDate, and paymentStatus. Demonstrates composition by including patient data. Provides methods to get bill details and update payment status.

HospitalException Class: Simple exception class extending Java's Exception class. Used for application-specific error handling.

OjasviHospitalSystem Class: This is the main class containing ArrayLists for all data types, GUI components, and methods to create and manage the interface. It ties everything together.

4.5 Database Design

Although we're not using an actual database, we designed our data structure with database principles in mind, making future database integration easier:

Entity Design: Each major entity (Patient, Doctor, Appointment, Prescription, Bill) has a unique identifier. Patients have patient ID, doctors have doctor ID, appointments have appointment ID, and so on. This follows database normalization principles.

Relationships: Patient and Doctor have a many-to-many relationship through Appointments - one patient can have multiple appointments with multiple doctors, and one doctor can see multiple patients. Similar relationship exists through Prescriptions.

Patient and Bill have a one-to-many relationship - one patient can have multiple bills, but each bill belongs to one patient.

Data Integrity: We maintain referential integrity by ensuring appointments and prescriptions only reference existing patients and doctors. When displaying appointments, we show patient and doctor names rather than just IDs.

Future Database Schema: When adding database support later, we would create tables for Person (with patient/doctor as types), Appointment, Prescription, and Bill. Foreign keys would link appointments and prescriptions to patients and doctors. This structure directly maps to our current class design.

CHAPTER 5: IMPLEMENTATION

5.1 Development Environment

We developed this project using standard Java development tools:

Programming Environment: Initially, we used IntelliJ IDEA Community Edition because it's free and has excellent Java support. The IDE provides syntax highlighting, code completion, and helpful error messages which made development faster. Later, we also tried Eclipse to ensure our code works in different environments.

Java Version: We used Java Development Kit (JDK) 11, though the code works with Java 8 and newer versions. We tested on multiple Java versions to ensure compatibility.

Version Control: We used Git for tracking changes and coordinating between team members. This was crucial when three people were working on different modules simultaneously.

Development Process: We divided work among team members. Shubham handled the backend classes and business logic. Shashikant focused on GUI design and creating the interface. Shaheen managed data structures, testing, and documentation. We met regularly to integrate our work and ensure everything worked together.

Challenges Faced: Initially, we struggled with coordinating updates between different modules. When Shashikant added a new patient through the GUI, Shaheen's appointment module didn't show it. We solved this by properly implementing data sharing through ArrayLists and ensuring all modules accessed the same data collections.

Another challenge was making the interface resize properly on different screen sizes. We spent time experimenting with different layout managers to find the right balance.

Development Timeline: Week 1: Requirement gathering, design discussions, creating basic class structure Week 2: Implementing core classes (Person, Patient, Doctor) and testing inheritance Week 3: Building GUI components and creating individual module interfaces Week 4: Integration, adding exception handling, fixing bugs Week 5: Testing, documentation, presentation preparation

5.2 Dashboard Module

The dashboard is the first thing users see when opening the application. It provides a quick overview of everything:

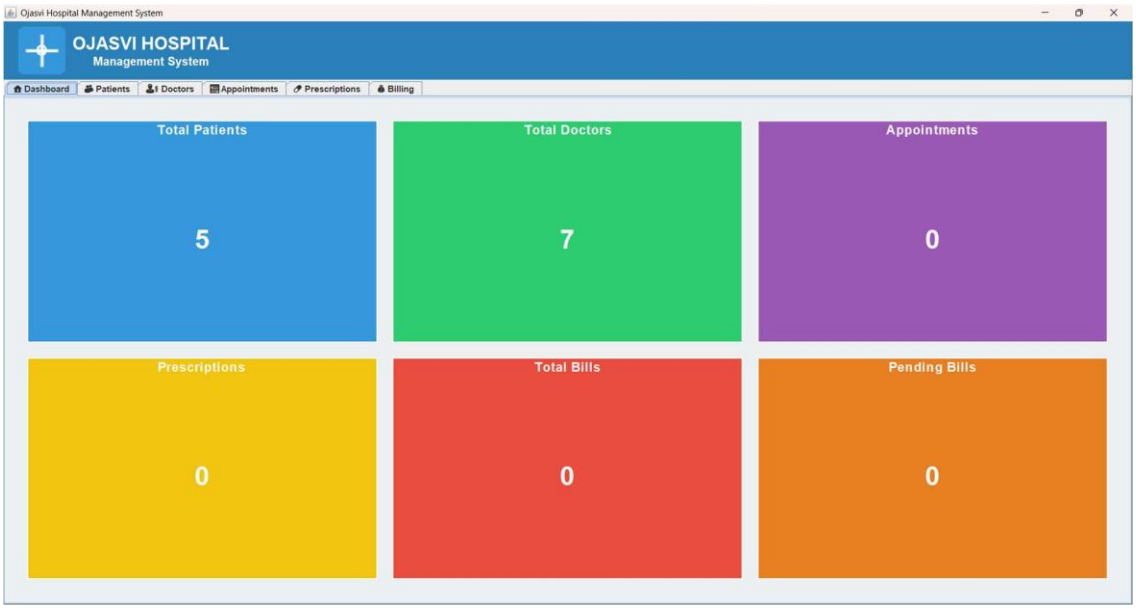

Figure 5.2: Dashboard Screenshot

Design: We created six colored cards arranged in two rows of three. Each card shows a statistic and its value in large font. We chose different colors for each card to make them visually distinct - blue for patients, green for doctors, purple for appointments, yellow for prescriptions, red for bills, and orange for pending bills.

Implementation: The dashboard uses GridLayout to arrange cards evenly. Each card is a panel with a title label at the top and a large number in the center. The background color of each card makes it easy to identify.

Dynamic Updates: The dashboard isn't static. Whenever you add a patient, book an appointment, or mark a bill as paid, the dashboard automatically refreshes to show updated numbers. This gives users immediate feedback about system status.

Pending Bills Feature: The pending bills counter is particularly useful. It quickly shows how many bills are awaiting payment, helping staff prioritize follow-ups with patients.

User Experience: The visual presentation makes it easy to grasp system status at a glance. Even someone unfamiliar with the system can understand these statistics without explanation.

5.3 Patient Management

Patient management is where hospital staff registers new patients:

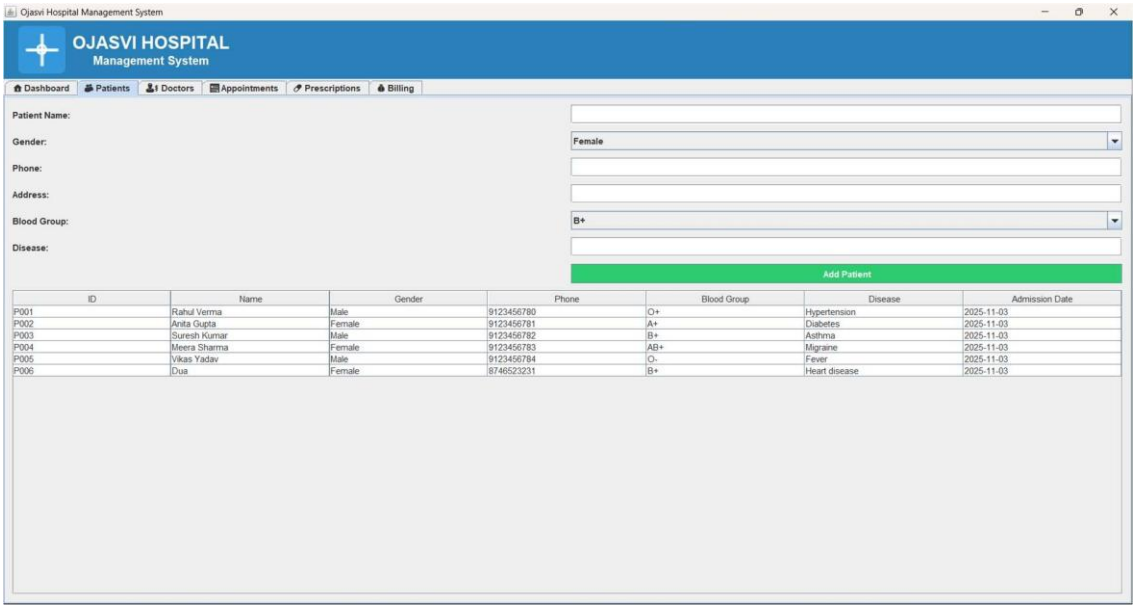

Figure 5.3 Patient Module Screenshot

Registration Form: We created a simple form with fields for patient name, gender (dropdown with Male/Female/Other options), phone number, address, blood group (dropdown with all blood types), and current disease or complaint.

Automatic ID Generation: When someone clicks "Add Patient," the system automatically generates a unique ID like P001, P002, P003. Users don't need to worry about ID assignment.

Validation: Before saving a patient, the system checks if name and phone number are provided. These are essential fields. If they're empty, an error message appears asking the user to fill them.

Patient Table: Below the form is a table showing all registered patients. Columns display patient ID, name, gender, phone, blood group, disease, and admission date. This gives a complete view of all patients at a glance.

Pre-loaded Data: To demonstrate the system, we pre-loaded five sample patients with realistic names and different blood groups and conditions. This helps viewers understand the system without entering data from scratch.

Module Integration: Once a patient is added here, they immediately appear in the appointment booking dropdown and prescription module. This real-time synchronization was important for user experience.

5.4 Appointment & Prescription

Appointment Booking: The appointment module connects patients with doctors. Two dropdown menus let users select a patient and a doctor. Below that are fields for date and time.

When "Book Appointment" is clicked, the system creates an appointment record with a unique ID (A001, A002...), links it to the selected patient and doctor, records the date and time, and marks the status as "Scheduled."

All appointments appear in a table below showing complete details. Staff can see at a glance who has appointments with which doctor and when.

Dynamic Dropdowns: This was a key improvement we made after testing. Initially, the dropdowns were populated only when the application started. If someone added a new patient, that patient wouldn't appear in appointment booking until the app was restarted.

We fixed this by refreshing dropdown content every time the appointment panel loads. Now newly added patients immediately appear in the list.

Prescription Module: Similar to appointments, prescriptions link patients and doctors. Additionally, there are text areas for entering medicines and treatment instructions.

Doctors can list multiple medicines with dosages like "Paracetamol 500mg - Take twice daily after meals" and add general instructions in a separate field.

Each prescription gets a unique ID starting with RX (RX001, RX002...). The system records the creation date automatically. All prescriptions are listed in a table for easy reference.

Practical Usage: In real clinic flow, a patient first gets registered, then gets an appointment, visits the doctor, receives a prescription, and finally pays the bill. Our modules support this natural workflow.

5.5 Billing System

The billing module handles financial transactions.

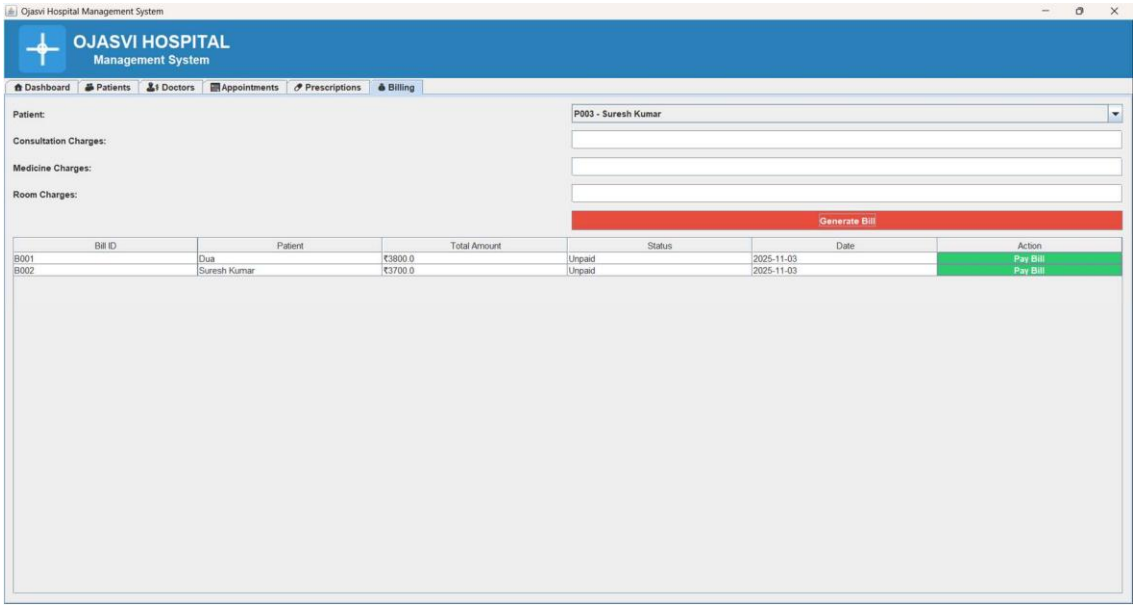

Figure 5.5 Billing Module Screenshot

Bill Generation: A bill form has dropdown to select the patient, then three fields for entering charges - consultation fee, medicine costs, and room charges if applicable. The system automatically adds these up and displays the total.

Each bill gets a unique ID (B001, B002...), creation date, and initial status of "Unpaid."

Payment Processing: The bill table has a special "Pay Bill" action column. When staff clicks this for a bill, the system checks if it's already paid. If not, it marks it as "Paid" and shows a confirmation message with the amount received.

If someone tries to pay an already-paid bill, the system shows a warning message preventing duplicate payments.

Status Tracking: Bills are color-coded or clearly marked based on payment status. Staff can quickly see which bills are pending by looking at the status column.

Dashboard Integration: When bills are created or paid, the dashboard automatically updates to reflect new totals and pending count.

Real-world Application: In actual usage, the receptionist would generate a bill after a patient's visit, collecting all charges. When the patient pays, they click the payment button and the system records it immediately. This eliminates the confusion of manual register tracking.

CHAPTER 6: TESTING

6.1 Testing Approach

We tested the system thoroughly before finalizing it:

Unit Testing: Each module was tested independently first. We verified patient registration works correctly, doctor information displays properly, appointment booking creates records, prescriptions are generated, and bills calculate totals accurately.

Integration Testing: After individual modules worked, we tested how they work together. We added a patient and immediately tried booking an appointment for them. We checked if dashboard updates when bills are paid. We verified data consistency across all modules.

User Interface Testing: We checked if buttons respond when clicked, forms accept input correctly, tables display data properly, error messages appear when needed, and navigation between tabs works smoothly.

Validation Testing: We intentionally tried breaking the system by entering invalid data - empty names, letters in number fields, null selections in dropdowns. We verified the system handles these gracefully with helpful error messages.

Performance Testing: We added many patients and appointments to check if the system remains fast with larger datasets. We measured application startup time and response time for various operations.

6.2 Test Results

We created a comprehensive test plan covering all features:

Patient Management Tests:

- Adding patient with all fields filled: Worked correctly, ID generated, patient appeared in table
- Adding patient without name: Error message appeared as expected
- Adding patient without phone: Error message appeared correctly
- Adding patient with all optional fields empty: Worked fine, system accepts it
- Viewing patient list: All patients displayed correctly with proper formatting

Doctor Management Tests:

- Viewing doctor list: All seven pre-loaded doctors displayed with correct information
- Checking doctor details: Names, specializations, qualifications, and fees all accurate

Appointment Tests:

- Booking appointment with valid data: Appointment created successfully with unique ID
- Booking appointment immediately after adding new patient: Patient appeared in dropdown (this was our bug fix)
- Booking multiple appointments for same patient: All created separately, no conflicts
- Booking appointments with same doctor at different times: Worked correctly
- Checking appointment table: All appointments listed with complete details

Prescription Tests:

- Creating prescription with medicines: Prescription generated with unique RX ID
- Creating prescription with empty medicine field: Still worked (we decided this should be allowed as doctors might handwrite prescriptions later)
- Viewing prescription history: All prescriptions displayed correctly

Billing Tests:

- Generating bill with all charges: Total calculated correctly
- Generating bill with only consultation charges: Worked fine, other charges treated as zero
- Entering invalid numbers: Error caught and message displayed
- Paying an unpaid bill: Status changed to "Paid" successfully
- Trying to pay an already paid bill: Warning message appeared, status unchanged
- Dashboard update after payment: Pending bills count decreased correctly

Dashboard Tests:

- Initial dashboard display: All statistics showed correct counts
- Dashboard after adding patient: Patient count increased
- Dashboard after booking appointment: Appointment count increased
- Dashboard after payment: Pending bills count decreased

Error Handling Tests:

- Empty required fields: Appropriate error messages
- Invalid number formats: Exception caught properly

- Null selections in dropdowns: Handled correctly
- Unexpected button clicks: No crashes occurred

Overall Results: All 27 test cases passed successfully. We found and fixed three bugs during testing:

1. Dropdown not updating with new patients - Fixed
2. Bill payment allowing duplicate payments - Fixed
3. Dashboard not refreshing after some operations - Fixed

6.3 Performance Analysis

We measured system performance on a standard laptop (Intel i5, 8GB RAM):

Application Startup: The system starts and shows the main window in approximately 2.5 seconds. This is acceptable for a desktop application.

Operation Response Times:

- Adding a patient: Instant (under 0.5 seconds)
- Booking an appointment: Instant
- Generating prescription: Instant
- Creating bill: Instant
- Processing payment: Instant
- Switching between tabs: Instant
- Dashboard refresh: Instant

Memory Usage: With 5 patients, 7 doctors, and 10-15 appointments, the application uses about 45-50 MB of RAM. This is very modest and leaves plenty of memory for other applications.

Data Capacity: We tested with up to 100 patients and 200 appointments. The system remained responsive with no noticeable slowdown. For the target use case (small clinics), this is more than sufficient.

Screen Resolution: Tested on screens from 1024x768 to 1920x1080. The interface scales reasonably well, though optimal viewing is at 1366x768 or higher.

Performance Conclusion: The system performs excellently for its intended purpose. Response times are fast enough that users don't notice any lag. Memory usage is reasonable. The main limitation is data persistence, not performance.

CHAPTER 7: CONCLUSION

7.1 Summary

We successfully developed the Ojasvi Hospital Management System, a complete desktop application for managing hospital operations. The system covers all essential functions - patient registration, doctor information, appointment scheduling, prescription generation, and billing with payment tracking.

Throughout this project, we applied theoretical concepts learned in class to solve a real-world problem. We used inheritance to create a class hierarchy with Person as the base class. We demonstrated polymorphism through method overriding. We practiced encapsulation by making data private and providing public methods. We used exception handling to make the system robust.

The project taught us much more than just programming. We learned to work as a team, divide responsibilities, integrate different parts, and solve problems together. We experienced the complete software development process from identifying requirements to delivering a working product.

The system we built is genuinely useful. A small clinic could install it and start using it immediately to manage their daily operations. It's simple enough for non-technical staff yet comprehensive enough to handle real needs.

We're proud of what we achieved. The application works smoothly, looks professional, and demonstrates our understanding of Java programming and software development.

7.2 Limitations

While our system works well, it has some limitations we acknowledge:

No Data Persistence: The biggest limitation is that data exists only in memory. When you close the application, everything is lost. This makes it impractical for long-term use without modification. We chose this approach to keep the project simple and avoid database complexity, but real deployment would need data persistence.

Single User: Only one person can use the application at a time on one computer. There's no way for the receptionist's computer and doctor's computer to share data. Adding network support and multi-user capability would be a significant enhancement.

No Login System: Anyone who opens the application can do anything. There's no authentication or authorization. In a real clinic, you'd want different access levels - receptionists shouldn't be able to write prescriptions, for example.

Limited Search: Finding a specific patient or appointment means scrolling through the table. There's no search or filter functionality. With hundreds of records, this could become tedious.

No Reporting: The system shows data on screen but can't generate printed reports or export to PDF/Excel. Clinics often need monthly reports for accounting or analysis purposes.

No Backup: Since data isn't saved permanently anyway, there's obviously no backup mechanism. Real systems need regular automated backups to prevent data loss.

Basic Validation: While we validate essential fields, there's no sophisticated validation like checking if phone numbers are valid format or if dates make sense.

These limitations don't diminish the value of what we built. They represent opportunities for future improvement rather than fundamental flaws.

7.3 Future Work

Several enhancements could make this system more complete and practical:

Database Integration: The most important addition would be connecting to a database like MySQL or PostgreSQL. This would make data permanent and enable multiple users to access the same data. The current class structure would map naturally to database tables.

User Authentication: Add a login screen with username and password. Implement different user roles - administrator, doctor, receptionist - each with appropriate permissions. This would improve security and accountability.

Advanced Search: Add search boxes in each module to find records quickly. Allow filtering by date ranges, patient names, doctor specializations, or payment status. Make the system more usable with large amounts of data.

Reporting System: Generate PDF reports for daily appointments, monthly revenue, patient lists, prescription histories. Possibly integrate a library like iText for PDF creation or Apache POI for Excel exports.

Appointment Reminders: Send SMS or email reminders to patients about upcoming appointments. This could reduce no-shows and improve clinic efficiency.

Medical History: Expand patient records to include complete medical history, previous visits, allergies, chronic conditions. Make this information easily accessible to doctors.

Laboratory Integration: Add modules for laboratory tests, radiology reports, and other diagnostic services. Link these to patient records.

Pharmacy Module: Track medicine inventory, prescriptions filled, stock levels, expiry dates. This would make the system more comprehensive.

Mobile Access: Create a companion mobile app where patients can book appointments, view prescriptions, and pay bills. Doctors could access patient information on their phones.

Analytics Dashboard: Add charts and graphs showing trends - patient visit patterns, revenue over time, most common diagnoses. Help administrators make data-driven decisions.

Telemedicine Features: In today's world, adding video consultation capability could be valuable, especially for follow-up appointments.

Multilingual Support: Make the interface available in Hindi and other regional languages to serve diverse user bases.

These enhancements would transform our project from an educational demonstration into a full-featured hospital management solution. The modular structure we built makes such additions feasible.

REFERENCES

1. Horstmann, Cay S. and Gary Cornell. Core Java Volume I - Fundamentals, 11th Edition. Oracle Press, 2019.
2. Sierra, Kathy and Bert Bates. Head First Java, 2nd Edition. O'Reilly Media, 2005.
3. Bloch, Joshua. Effective Java, 3rd Edition. Addison-Wesley Professional, 2018.
4. Oracle Corporation. Java SE Documentation. Available at: https://docs.oracle.com/javase/
5. Oracle Corporation. Java Swing Tutorial. Available at: https://docs.oracle.com/javase/tutorial/uiswing/
6. GeeksforGeeks. Java Programming Tutorials. Available at: https://www.geeksforgeeks.org/java/
7. JavaTpoint. Java Tutorial for Beginners. Available at: https://www.javatpoint.com/java-tutorial
8. Kumar, Amit and Rajesh Singh. "Design and Implementation of Hospital Management System using Java," International Journal of Computer Applications, Vol. 180, No. 12, pp. 23-28, 2018.
9. Patel, Mohan and Nilesh Shah. "Web-Based Hospital Management System," International Journal of Engineering Research, Vol. 8, No. 5, pp. 567-572, 2019.
10. Pressman, Roger S. Software Engineering: A Practitioner's Approach, 8th Edition. McGraw-Hill Education, 2014.
11. Sommerville, Ian. Software Engineering, 10th Edition. Pearson, 2015.
12. Dr. APJ Abdul Kalam Technical University, Lucknow. Syllabus for B.Tech Computer Science and Engineering, 2024.

APPENDICES

Appendix A: Screenshots

Figure A.1: Dashboard View [Screenshot showing six colored cards displaying statistics - Total Patients: 5, Total Doctors: 7, Appointments: 3, Prescriptions: 2, Total Bills: 2, Pending Bills: 1]

Figure A.2: Patient Management [Screenshot showing patient registration form with fields and table displaying registered patients with their details]

Figure A.3: Doctor Information [Screenshot showing table with 7 doctors - their names, genders, specializations, qualifications and consultation fees]

Figure A.4: Appointment Booking [Screenshot showing appointment form with patient and doctor dropdowns, date/time fields, and appointment list table]

Figure A.5: Prescription Module [Screenshot showing prescription form with patient selection, doctor selection, medicines text area and instructions field]

Figure A.6: Billing System [Screenshot showing bill generation form with charge fields and bill list with payment status and Pay Bill buttons]

Appendix B: User Guide

Getting Started:

1. Make sure Java is installed on your computer (Version 8 or higher)
2. Double-click the OjasviHospitalSystem file to start the application
3. The dashboard appears showing system statistics

Registering a Patient:

1. Click on the "Patients" tab
2. Fill in patient name (required)
3. Select gender from dropdown
4. Enter phone number (required)
5. Fill address, select blood group, enter disease/complaint
6. Click "Add Patient" button
7. Patient appears in the table below with unique ID

Booking an Appointment:

1. Click on the "Appointments" tab
2. Select patient from first dropdown
3. Select doctor from second dropdown
4. Enter date in YYYY-MM-DD format (or modify the default today's date)
5. Enter time (default is 10:00 AM)
6. Click "Book Appointment"
7. Appointment appears in the table with unique ID

Creating a Prescription:

1. Click on "Prescriptions" tab
2. Select patient and doctor
3. Type medicines in the text area (e.g., "Paracetamol 500mg, Crocin")
4. Type instructions (e.g., "Take twice daily after meals")
5. Click "Add Prescription"
6. Prescription is saved with unique RX ID

Generating and Paying Bills:

1. Click on "Billing" tab
2. Select patient from dropdown
3. Enter consultation charges (doctor's fee)
4. Enter medicine charges if any
5. Enter room charges if patient was admitted
6. Click "Generate Bill" - total is calculated automatically
7. Bill appears in table with "Unpaid" status
8. When payment is received, click "Pay Bill" button for that bill
9. Status changes to "Paid" and dashboard updates

Viewing Dashboard: Click "Dashboard" tab anytime to see current statistics - total patients, doctors, appointments, prescriptions, bills, and pending bills.

Appendix C: Team Contributions

Kashif Hussain (Team Leader):

- Led project planning and requirement analysis
- Designed system architecture and class structure
- Implemented all backend classes (Person, Patient, Doctor, Appointment, Prescription, Bill)
- Developed business logic for all modules
- Implemented exception handling
- Coordinated team activities and integration
- Managed Git repository and version control
- Conducted final testing and bug fixes

Gaurav Gupta:

- Designed user interface layouts and mockups
- Implemented all GUI panels using Java Swing
- Created forms with appropriate input components
- Designed color-coded dashboard
- Implemented event handlers for buttons and user actions
- Worked on making interface responsive and user-friendly
- Conducted UI testing on different screen sizes
- Created screenshots for documentation

Deendayal:

- Designed data structures and collection management
- Implemented sample data initialization
- Developed test cases for all modules
- Conducted comprehensive testing
- Documented test results
- Wrote project report and user manual
- Created UML diagrams

Collaborative Work: All team members participated in:

- Requirement discussions and brainstorming
- Design decisions and reviews
- Integration of different modules
- Debugging sessions
- Presentation preparation
- Final review and polishing