Hospital Management System

Project Report Submission

Submitted by:

Mannan Arora (24BAS10066)

Course: Java Programming

1 Introduction

The **Hospital Management System (HMS)** is a console-based Java application designed to digitize the fundamental operations of a hospital clinic. In many traditional healthcare settings, reliance on paper-based records leads to inefficiencies, data loss, and scheduling conflicts. This project aims to resolve these issues by providing a centralized digital platform for managing doctor profiles, patient registrations, and appointment scheduling.

The system is built using core Java concepts, emphasizing Object-Oriented Programming (OOP) principles and robust file handling for data persistence.

2 Problem Statement

Traditional manual hospital systems face several critical challenges:

- **Inefficient Record Keeping:** Physical files are difficult to search, update, and maintain.
- **Data Loss Risks:** Paper records are susceptible to damage or misplacement.
- **Scheduling Conflicts:** Without a central registry, booking appointments often leads to double-booking or confusion.
- **Lack of History:** Tracking a patient's medical history across multiple visits is cumbersome manually.

There is a need for a lightweight, reliable digital solution to streamline these operations for small to medium-sized clinics.

3 Functional Requirements

The system implements the following core functional modules:

3.1 Doctor Management

- Administrators can add new doctors to the system.
- Stores details including Doctor ID, Name, Age, and Specialization.
- View a list of all available doctors.

3.2 Patient Management

- Receptionists can register new patients.
- Captures Patient ID, Name, Age, and Current Ailment.
- Maintains a history of appointments (log) for each patient.

3.3 Appointment Scheduling

- Create appointments by linking a valid Doctor ID and Patient ID.

- Auto-generates unique Appointment IDs.
- Validates that both the doctor and patient exist before booking.

3.4 Data Persistence

- Automatically saves all data (Doctors, Patients, Appointments) to local binary files (`.dat`) upon exit.
- Reloads data automatically when the application restarts.

4 Non-functional Requirements

- **Reliability:** The system ensures data is never lost between sessions using Java Serialization.
- **Error Handling:** Robust handling of exceptions (e.g., `HospitalException`) prevents the application from crashing due to invalid input.
- **Usability:** A clean, text-based menu interface ensures the system is easy to navigate for non-technical staff.
- **Performance:** Lightweight console application with instant response times for queries and updates.
- **Maintainability:** The code follows a modular structure (Model-Service-Controller pattern), making it easy to update or extend.

5 System Architecture

The project follows a modular architecture separating data, logic, and interaction:

- **Presentation Layer (com.hms.main):** Handles user input via the console and displays data.
- **Service Layer (com.hms.service):** Contains the business logic (e.g., validating IDs, linking objects).
- **Data Layer (com.hms.model):** Defines the structure of data objects (POJOs).
- **Persistence Layer (com.hms.util):** Manages reading and writing objects to the file system.

6 Design Diagrams

6.1 Use Case Diagram

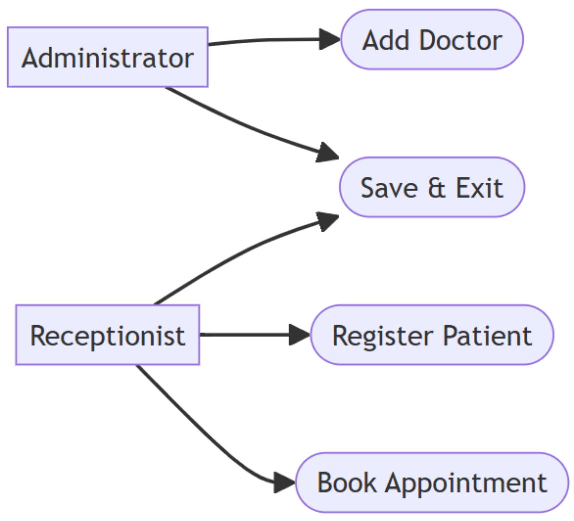


Figure 1: Use Case Diagram illustrating user interactions.

6.2 Class Diagram

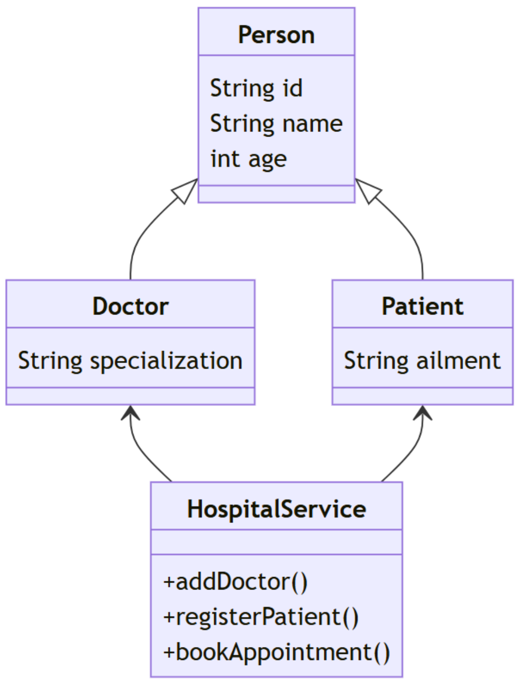


Figure 2: UML Class Diagram showing inheritance and relationships.

6.3 Sequence Diagram (Booking Appointment)

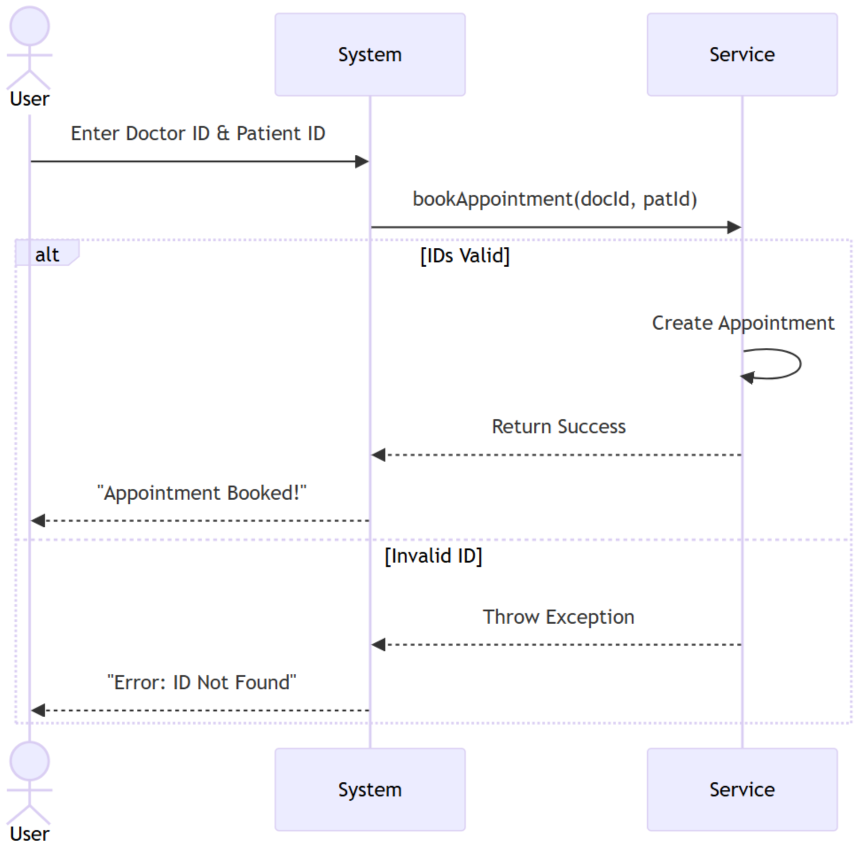


Figure 3: Sequence of events for booking an appointment.

7 Design Decisions & Rationale

- **File Storage vs. Database:** For this project scope, local file storage (Serialization) was chosen over a SQL database to keep the application self-contained and portable without requiring external server installations.
- **Inheritance Strategy:** An abstract `Person` class was created to hold common fields (ID, Name, Age) for both `Doctor` and `Patient`. This reduces code duplication and demonstrates OOP Polymorphism.
- **Custom Exceptions:** Instead of generic Java errors, a `HospitalException` was created to handle domain-specific errors (like "Doctor Not Found"), improving clarity for the user.

8 Implementation Details

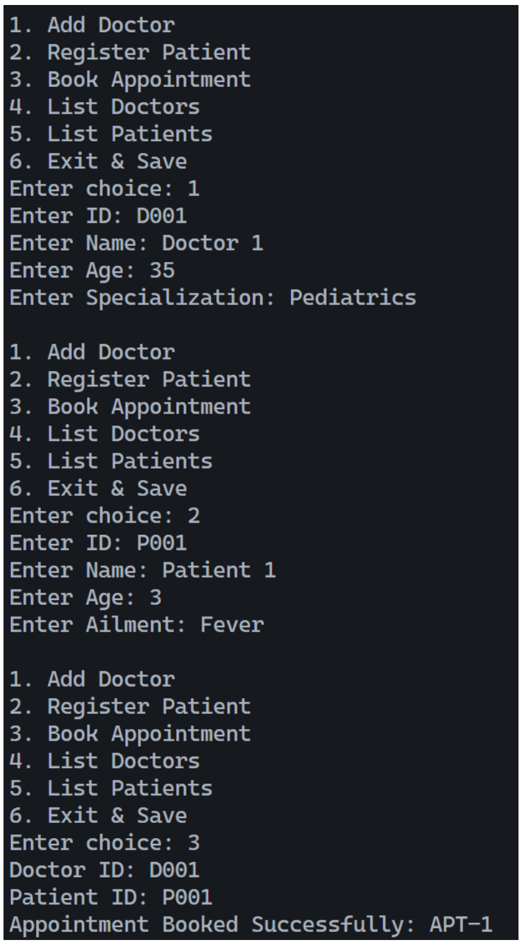
The application is structured into the following Java packages:

- `com.hms.model`: Contains entity classes (`Doctor`, `Patient`, `Appointment`) implementing `Serializable`.
- `com.hms.service`: `HospitalService.java` acts as the controller, managing lists of data and business logic.

- `com.hms.util`: `DataManager.java` handles the `ObjectOutputStream` and `ObjectInputStream` operations.
- `com.hms.exception`: Defines the custom `HospitalException`.
- `com.hms.main`: The entry point containing the `Scanner` loop for user interaction.

9 Screenshots / Results

9.1 Booking Success



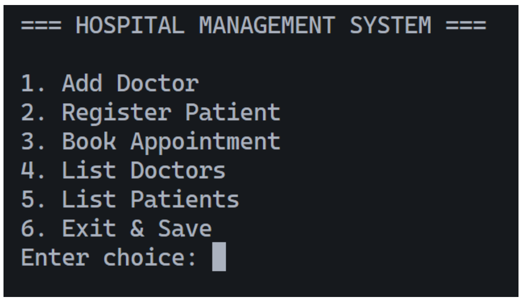
```
1. Add Doctor
2. Register Patient
3. Book Appointment
4. List Doctors
5. List Patients
6. Exit & Save
Enter choice: 1
Enter ID: D001
Enter Name: Doctor 1
Enter Age: 35
Enter Specialization: Pediatrics

1. Add Doctor
2. Register Patient
3. Book Appointment
4. List Doctors
5. List Patients
6. Exit & Save
Enter choice: 2
Enter ID: P001
Enter Name: Patient 1
Enter Age: 3
Enter Ailment: Fever

1. Add Doctor
2. Register Patient
3. Book Appointment
4. List Doctors
5. List Patients
6. Exit & Save
Enter choice: 3
Doctor ID: D001
Patient ID: P001
Appointment Booked Successfully: APT-1
```

Figure 4: Successful Appointment Booking

9.2 Main Menu



```
=== HOSPITAL MANAGEMENT SYSTEM ===

1. Add Doctor
2. Register Patient
3. Book Appointment
4. List Doctors
5. List Patients
6. Exit & Save
Enter choice: █
```

Figure 5: Main Application Menu

10 Testing Approach

Testing was conducted using a hybrid approach:

1. **Manual Functional Testing:** Each menu option was executed manually to verify inputs and outputs.
2. **Automated Test Runner:** A custom `TestRunner.java` class was created to programmatically test core functions:
   - *Positive Testing:* Verifying that valid doctors and patients can be added and linked.
   - *Negative Testing:* Verifying that the system throws a `HospitalException` when attempting to book an appointment with non-existent IDs.

11 Challenges Faced

- **Data Serialization:** Initial issues occurred when modifying class structures after saving data, leading to `InvalidClassException`. This was resolved by stabilizing the class design before saving substantial data.
- **Reference Handling:** Ensuring that the `Appointment` object correctly referenced the `Doctor` and `Patient` IDs without creating deep copies that would desynchronize data.

12 Learnings & Key Takeaways

- Gained practical experience in **Java File I/O** and Object Serialization.
- Understood the importance of **Separation of Concerns** (SOC) by keeping UI code separate from business logic.
- Learned how to implement and handle **Custom Exceptions** to control program flow gracefully.
- Improved proficiency in **Git version control** for tracking incremental changes.

13 Future Enhancements

- **Database Integration:** Migrate from flat files to a MySQL/PostgreSQL database for better scalability.
- **Graphical User Interface (GUI):** Implement a JavaFX or Web-based interface (Spring Boot) to replace the console UI.
- **Doctor Login:** Add a specific portal for doctors to view only their own appointments.

14 References

- Oracle Java Documentation (File I/O, Streams).
- Course Materials of the Vityarthi course
- StackOverflow: Handling `serialVersionUID` best practices.