Patient Appointment & Billing Management System (P.A.B.M.S.)

1) Story

A growing multi-specialty clinic chain, **CarePlus Clinics**, operates 4 branches. They currently manage:

- Patient registrations
- Doctor appointments
- Treatment records
- Medicine stock
- Billing and payments

using Excel sheets and paper files.

This causes issues:

- Double booking of doctors
- Lost patient history
- Incorrect medicine stock tracking
- No daily revenue visibility
- No doctor performance insights

CarePlus wants a **Java-based Clinic Management System** that uses a relational DB, supports role-based actions (Admin vs Receptionist), logs all operations, and produces business reports.

You are hired as the backend team to build the first working version.

---

2) What You Will Build

A console-based Java application (no UI frameworks required) that:

- Manages Patients, Doctors, Appointments, Treatments, Medicines, Bills, and Payments
- Persists data in MySQL / PostgreSQL using SQL + JDBC
- Uses layered architecture
- Has JUnit tests for core services/validations
- Uses Log4j logging

- Uses Git with proper branching + commit discipline

---

3) Mandatory Tech / Concept Coverage

Core Java (Must Use)

- OOP (encapsulation, abstraction, interfaces)
- Collections (List/Map in service layer)
- Enums (roles, appointment status, bill status, payment mode)
- Java Time API (LocalDateTime, LocalDate)
- Custom exceptions
- Defensive validation

SQL + JDBC (Must Use)

- Normalized schema (min 3NF)
- CRUD with PreparedStatement
- Transactions (commit/rollback) during billing
- Joins & aggregation queries for reporting
- Constraints: PK, FK, UNIQUE, NOT NULL, CHECK

Git (Must Use)

- Feature branches
- Meaningful commits
- PR-style workflow

JUnit (Must Use)

- Service-layer tests
- At least 12 meaningful test cases
- Clear naming conventions

Log4j (Must Use)

- Separate logger categories (flow vs DB)
- Log levels: INFO/WARN/ERROR/DEBUG
- File + console output
- No System.out.println() except menu prompts

Architecture + Quality

- Strict layered design
- No SQL in service layer
- Centralized exception strategy
- Properties file config
- Clean code principles (SRP, no god classes)

---

4) Functional Requirements

A) User Roles

ADMIN

- Add/update/deactivate doctors
- Add/update medicines
- Adjust medicine stock
- View reports

RECEPTIONIST

- Register patient
- Create appointment
- Create bill
- Process payment
- Print receipt

Authentication: simple username + role input.

B) Patient Management

Each patient must have:

- patientId (auto)
- name
- phone (unique)
- dateOfBirth
- createdAt

Operations:

- Create patient
- Search by phone
- View patient history

C) Doctor Management

Each doctor must have:

- doctorId (auto)
- name
- specialization
- consultationFee
- active (true/false)
- createdAt

Operations:

- Add doctor
- Deactivate doctor
- List active doctors

D) Appointments

Each appointment must have:

- appointmentId
- patientId FK
- doctorId FK
- appointmentDateTime
- status (BOOKED, COMPLETED, CANCELLED)
- createdAt

Rules:

- Cannot book inactive doctor
- Cannot double-book same doctor at same date/time

- Status transitions must be logical

E) Medicines (Inventory)

Each medicine must have:

- medicineId
- medicineCode (unique)
- name
- unitPrice
- availableQuantity
- reorderLevel
- active
- createdAt

Operations:

- Add medicine
- Update price
- Adjust stock (admin)
- Low stock report

Audit: every stock adjustment must insert into stock_adjustments table.

F) Billing (Core Transaction)

A bill includes:

- billId
- appointmentId FK
- consultationFee
- medicinesTotal
- totalAmount
- status (CREATED, PAID)
- createdAt

Bill items:

- medicineId
- quantity
- unitPriceAtSale
- lineTotal

Rules:

- Cannot use more medicine than available stock
- Payment marks bill as PAID
- Must be transactional

Design decision (mandatory):

Stock must be deducted only after payment SUCCESS.

G) Payments

Payment includes:

- paymentId
- billId FK
- mode (CASH, CARD, UPI)
- amount
- status (SUCCESS, FAILED)
- paidAt

Rules:

- Amount must equal totalAmount (v1)

H) Reporting Requirements

1. Daily Revenue Summary
    - total appointments
    - total revenue
    - top 3 medicines sold
2. Doctor Performance Report
    - completed appointments count
    - revenue generated
3. Low Stock Medicines Report
4. Patient Visit History

Reports must use SQL joins + GROUP BY.

---

5) Exception Handling Rules

Create:

- ValidationException
- EntityNotFoundException
- InsufficientStockException
- DoubleBookingException
- DatabaseOperationException

---

6) JUnit Testing Requirements

Minimum required tests:

1. shouldRejectInactiveDoctorBooking()
2. shouldPreventDoubleBooking()
3. shouldThrowInsufficientStock()
4. shouldCalculateBillTotalCorrectly()
5. shouldFailPaymentWhenAmountMismatch()
6. shouldRollbackWhenBillingFails()
7. shouldDeactivateDoctorAndPreventBooking()
8. shouldNormalizePhone()
9. shouldRejectNegativeConsultationFee()
10. shouldRejectZeroQuantityMedicine()
11. shouldReturnDoctorPerformanceAggregated()
12. shouldReturnLowStockMedicines()

---

7) Evaluation Rubric

(Same as previous projects)

- Architecture & separation: 25%
- SQL & JDBC correctness: 20%
- Transaction safety: 15%
- Testing: 15%
- Logging & exception handling: 10%
- Git hygiene: 10%
- Code cleanliness: 5%

Starter Kit

None

```
|careplus-pabms/
├─ README.md
│  └─ HINT: Document setup steps, DB initialization, how to run, sample menu flow,
│           transaction design (stock deduction after payment SUCCESS), and any assumptions.
│
├─ schema.sql
│  └─ HINT: Create all tables + constraints + seed data (5+ doctors, 8+ medicines).
│           Enforce UNIQUE, FK, NOT NULL, CHECK where possible.
│
├─ pom.xml  (or build.gradle)
│  └─ HINT: Add JDBC driver (MySQL/Postgres), Log4j2, JUnit5, Surefire plugin.
│
├─ src/
│  ├─ main/
│  │  ├─ java/
│  │  │  └─ com/
│  │  │     └─ careplus/
│  │  │        └─ pabms/
│  │  │
│  │  │           ├─ App.java
│  │  │           │  └─ HINT: Application entry point.
│  │  │           │           Bootstraps controllers, shows main menu, handles global exceptions.
│  │  │           │
│  │  │           ├─ config/
│  │  │           │  ├─ AppConfig.java
│  │  │           │  │  └─ HINT: Manual wiring of services + DAOs (simple dependency injection).
│  │  │           │  └─ DbConfig.java
│  │  │           │     └─ HINT: Read db.properties and expose DB connection settings.
│  │  │           │
│  │  │           ├─ controller/
│  │  │           │  ├─ PatientController.java
│  │  │           │  │  └─ HINT: Register/search patients; collect input and call PatientService.
│  │  │           │  │           Never call DAO directly.
│  │  │           │  │
│  │  │           │  ├─ DoctorController.java
│  │  │           │  │  └─ HINT: Admin menu for doctor management (add/deactivate/list).
│  │  │           │  │
│  │  │           │  ├─ AppointmentController.java
│  │  │           │  │  └─ HINT: Receptionist flow for booking appointments;
│  │  │           │  │           validates date/time and calls AppointmentService.
│  │  │           │  │
│  │  │           │  ├─ BillingController.java
│  │  │           │  │  └─ HINT: Handles billing workflow:
│  │  │           │  │           add medicines → compute totals → payment → print receipt.
│  │  │           │  │
│  │  │           │  └─ ReportController.java
│  │  │           │     └─ HINT: Menu for reports (daily revenue, doctor performance,
│  │  │           │              low stock, patient history). Formats output.
│  │  │           │
│  │  │           ├─ dao/
│  │  │           │  ├─ PatientDao.java
│  │  │           │  │  └─ HINT: CRUD + findByPhone.
│  │  │           │  │           No validation, no business logic.
│  │  │           │  │
│  │  │           │  ├─ DoctorDao.java
│  │  │           │  │  └─ HINT: CRUD + listActiveDoctors.
│  │  │           │  │
│  │  │           │  ├─ AppointmentDao.java
│  │  │           │  │  └─ HINT: Create appointment, update status,
│  │  │           │  │           check double-booking via query.
│  │  │           │  │
│  │  │           │  ├─ MedicineDao.java
│  │  │           │  │  └─ HINT: CRUD medicines + findByCode + search.
```

```
|  |  |           |  |
|  |  |           |  ├─ MedicineInventoryDao.java
|  |  |           |  |  └─ HINT: Update and fetch availableQuantity safely.
|  |  |           |  |           Must prevent negative stock.
|  |  |           |  |
|  |  |           |  ├─ BillDao.java
|  |  |           |  |  └─ HINT: Insert bill header + update totals/status +
|  |  |           |  |           fetch bill for printing.
|  |  |           |  |
|  |  |           |  ├─ BillItemDao.java
|  |  |           |  |  └─ HINT: Insert bill line items; batch insert recommended.
|  |  |           |  |
|  |  |           |  ├─ PaymentDao.java
|  |  |           |  |  └─ HINT: Insert payment + fetch by billId.
|  |  |           |  |
|  |  |           |  ├─ StockAdjustmentDao.java
|  |  |           |  |  └─ HINT: Insert medicine stock adjustments (delta + reason).
|  |  |           |  |
|  |  |           |  ├─ ReportDao.java
|  |  |           |  |  └─ HINT: Contains SQL-heavy queries (JOIN, GROUP BY, SUM).
|  |  |           |  |           Return DTOs, not formatted strings.
|  |  |           |  |
|  |  |           |  └─ impl/
|  |  |           |     ├─ JdbcPatientDao.java
|  |  |           |     |  └─ HINT: PreparedStatements only; wrap SQLException.
|  |  |           |     ├─ JdbcDoctorDao.java
|  |  |           |     ├─ JdbcAppointmentDao.java
|  |  |           |     ├─ JdbcMedicineDao.java
|  |  |           |     ├─ JdbcMedicineInventoryDao.java
|  |  |           |     ├─ JdbcBillDao.java
|  |  |           |     ├─ JdbcBillItemDao.java
|  |  |           |     ├─ JdbcPaymentDao.java
|  |  |           |     ├─ JdbcStockAdjustmentDao.java
|  |  |           |     └─ JdbcReportDao.java
|  |  |           |     └─ HINT: All JDBC code lives here.
|  |  |           |              Catch SQLException → throw DatabaseOperationException.
|  |  |
|  |  |           ├─ service/
|  |  |           |  ├─ PatientService.java
|  |  |           |  |  └─ HINT: Normalize phone, validate input, create/find patients.
|  |  |           |  |
|  |  |           |  ├─ DoctorService.java
|  |  |           |  |  └─ HINT: Validate consultationFee > 0; deactivate doctor safely.
|  |  |           |  |
|  |  |           |  ├─ AppointmentService.java
|  |  |           |  |  └─ HINT: Enforce no double-booking + active doctor check.
|  |  |           |  |           Manage status transitions.
|  |  |           |  |
|  |  |           |  ├─ MedicineService.java
|  |  |           |  |  └─ HINT: Validate medicineCode uniqueness, positive price,
|  |  |           |  |           admin stock adjustments.
|  |  |           |  |
|  |  |           |  ├─ BillingService.java
|  |  |           |  |  └─ HINT: Core transaction:
|  |  |           |  |           validate stock → insert bill + items →
|  |  |           |  |           insert payment → deduct stock on SUCCESS → commit;
|  |  |           |  |           rollback on failure.
|  |  |           |  |
|  |  |           |  └─ ReportService.java
|  |  |           |     └─ HINT: Validate date ranges; call ReportDao; return DTOs.
|  |  |
|  |  |           ├─ model/
|  |  |           |  ├─ Patient.java
|  |  |           |  ├─ Doctor.java
|  |  |           |  ├─ Appointment.java
|  |  |           |  ├─ Medicine.java
|  |  |           |  ├─ Bill.java
|  |  |           |  ├─ BillItem.java
|  |  |           |  ├─ Payment.java
|  |  |           |  └─ report/
|  |  |           |     ├─ DailyRevenueRow.java
|  |  |           |     ├─ DoctorPerformanceRow.java
```

```
|  |  |           |     ├─ LowStockRow.java
|  |  |           |     └─ PatientHistoryRow.java
|  |  |           |     └─ HINT: DTO classes for report outputs.
|  |  |
|  |  |           ├─ exception/
|  |  |           |  ├─ ValidationException.java
|  |  |           |  ├─ EntityNotFoundException.java
|  |  |           |  ├─ InsufficientStockException.java
|  |  |           |  ├─ DoubleBookingException.java
|  |  |           |  └─ DatabaseOperationException.java
|  |  |           |  └─ HINT: Service throws domain exceptions; DAO wraps SQL errors.
|  |  |
|  |  |           ├─ util/
|  |  |           |  ├─ DbConnectionFactory.java
|  |  |           |  ├─ InputUtil.java
|  |  |           |  ├─ ValidationUtil.java
|  |  |           |  ├─ MoneyUtil.java
|  |  |           |  └─ DateUtil.java
|  |  |           |  └─ HINT: Shared utilities. No business logic here.
|  |  |
|  |  |           └─ constants/
|  |  |              ├─ Role.java
|  |  |              ├─ AppointmentStatus.java
|  |  |              ├─ BillStatus.java
|  |  |              ├─ PaymentMode.java
|  |  |              └─ PaymentStatus.java
|  |  |              └─ HINT: Store enum values as text in DB for readability.
|  |
|  └─ resources/
|     ├─ db.properties
|     |  └─ HINT: Template only (no passwords committed).
|     └─ log4j2.xml
|        └─ HINT: Configure console + file appenders.
|
└─ test/
   └─ java/
      └─ com/careplus/pabms/service/
         ├─ AppointmentServiceTest.java
         |  └─ HINT: Test double-booking + inactive doctor rule.
         ├─ BillingServiceTest.java
         |  └─ HINT: Test stock validation + totals + rollback behavior.
         ├─ MedicineServiceTest.java
         |  └─ HINT: Test medicine validations + uniqueness.
         └─ ValidationUtilTest.java
            └─ HINT: Test phone normalization + quantity validation.
```

Minimal bootstrap classes (copy/paste)

Java

```java
package com.careplus.pabms;

import com.careplus.pabms.controller.AppointmentController;
import com.careplus.pabms.controller.BillingController;
import com.careplus.pabms.controller.DoctorController;
import com.careplus.pabms.controller.PatientController;
import com.careplus.pabms.controller.ReportController;
```

```java
import com.careplus.pabms.util.InputUtil;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class App {
    private static final Logger log = LogManager.getLogger(App.class);

    public static void main(String[] args) {
        log.info("CarePlus P.A.B.M.S. started");

        PatientController patientController = new PatientController();
        DoctorController doctorController = new DoctorController();
        AppointmentController appointmentController = new
AppointmentController();
        BillingController billingController = new BillingController();
        ReportController reportController = new ReportController();

        while (true) {
            System.out.println("\n=== CarePlus P.A.B.M.S. ===");
            System.out.println("1. Patients");
            System.out.println("2. Doctors (Admin)");
            System.out.println("3. Appointments");
            System.out.println("4. Billing");
            System.out.println("5. Reports");
            System.out.println("0. Exit");

            int choice = InputUtil.readInt("Choose: ");
            switch (choice) {
                case 1 -> patientController.menu();
                case 2 -> doctorController.menu();
                case 3 -> appointmentController.menu();
                case 4 -> billingController.menu();
                case 5 -> reportController.menu();
                case 0 -> {
                    log.info("CarePlus P.A.B.M.S. stopped");
                    System.out.println("Bye!");
                    return;
                }
                default -> System.out.println("Invalid option.");
            }
        }
    }
}
```

DbConnectionFactory.java (single place for connections)

```java
package com.careplus.pabms.util;
```

```
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Properties;

public final class DbConnectionFactory {

    private static final Properties props = new Properties();

    static {
        try (InputStream in =
DbConnectionFactory.class.getClassLoader().getResourceAsStream("db.propertie
s")) {
            if (in == null) {
                throw new IllegalStateException("db.properties not found in
resources/");
            }
            props.load(in);
        } catch (Exception e) {
            throw new ExceptionInInitializerError("Failed to load
db.properties: " + e.getMessage());
        }
    }

    private DbConnectionFactory() {}

    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(
                    props.getProperty("db.url"),
                    props.getProperty("db.username"),
                    props.getProperty("db.password")
            );
        } catch (Exception e) {
            throw new RuntimeException("DB connection failed: " +
e.getMessage(), e);
        }
    }
}
```

DB Schema

None

```
CREATE DATABASE IF NOT EXISTS careplus_pabms;
USE careplus_pabms;

CREATE TABLE patients (
  patient_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(120) NOT NULL,
  phone VARCHAR(20) NOT NULL UNIQUE,
  date_of_birth DATE NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE doctors (
  doctor_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(120) NOT NULL,
  specialization VARCHAR(80) NOT NULL,
  consultation_fee DECIMAL(10,2) NOT NULL,
  active BOOLEAN DEFAULT TRUE,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE appointments (
  appointment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  patient_id BIGINT NOT NULL,
  doctor_id BIGINT NOT NULL,
  appointment_datetime DATETIME NOT NULL,
  status VARCHAR(20) NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  UNIQUE (doctor_id, appointment_datetime),
  FOREIGN KEY (patient_id) REFERENCES patients(patient_id),
  FOREIGN KEY (doctor_id) REFERENCES doctors(doctor_id)
);

CREATE TABLE medicines (
  medicine_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  medicine_code VARCHAR(50) UNIQUE NOT NULL,
  name VARCHAR(120) NOT NULL,
  unit_price DECIMAL(10,2) NOT NULL,
  available_quantity INT NOT NULL,
  reorder_level INT NOT NULL DEFAULT 5,
  active BOOLEAN DEFAULT TRUE
);

CREATE TABLE bills (
  bill_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  appointment_id BIGINT UNIQUE NOT NULL,
  consultation_fee DECIMAL(10,2) NOT NULL,
  medicines_total DECIMAL(12,2) NOT NULL,
  total_amount DECIMAL(12,2) NOT NULL,
  status VARCHAR(20) NOT NULL,
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (appointment_id) REFERENCES appointments(appointment_id)
);

CREATE TABLE bill_items (
  bill_item_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  bill_id BIGINT NOT NULL,
  medicine_id BIGINT NOT NULL,
  quantity INT NOT NULL,
  unit_price_at_sale DECIMAL(10,2) NOT NULL,
  line_total DECIMAL(12,2) NOT NULL,
  FOREIGN KEY (bill_id) REFERENCES bills(bill_id) ON DELETE CASCADE,
  FOREIGN KEY (medicine_id) REFERENCES medicines(medicine_id)
);

CREATE TABLE payments (
  payment_id BIGINT PRIMARY KEY AUTO_INCREMENT,
  bill_id BIGINT UNIQUE NOT NULL,
  mode VARCHAR(20) NOT NULL,
  amount DECIMAL(12,2) NOT NULL,
  status VARCHAR(20) NOT NULL,
  paid_at DATETIME,
```

```
  FOREIGN KEY (bill_id) REFERENCES bills(bill_id) ON DELETE CASCADE
);
```

`db.properties` (template)

```
db.url=jdbc:mysql://localhost:3306/careplus_pabms

db.username=root

db.password=YOUR_PASSWORD
```

Sample menu flow (console UX)

Main Menu

```
=== CarePlus P.A.B.M.S. ===

1. Patients

2. Doctors (Admin)

3. Appointments

4. Billing

5. Reports

0. Exit
```

Patients Menu

```
--- Patients ---

1. Register patient

2. Search patient by phone
```

```
3. View patient visit history (by phone)

0. Back
```

Doctors Menu (Admin)

```
--- Doctors (Admin) ---

1. Add doctor

2. Update doctor fee/specialization

3. Deactivate doctor

4. List active doctors

0. Back
```

Appointments Menu

```
--- Appointments ---

1. Book appointment (Receptionist)

2. Cancel appointment

3. Mark appointment as COMPLETED

4. Find appointment by ID

5. List appointments by date

0. Back
```

Recommended flow for “Book appointment”:

1. Ask patient phone → fetch patient (or register if new)

2. Show doctor list (active) → choose doctor
3. Enter appointment date/time
4. Validate: doctor active + not already booked at that slot
5. Create appointment with status BOOKED

Billing Menu (Core Workflow)

```
--- Billing ---

1. Create bill for appointment

2. Add medicine item (during billing)

3. Checkout + payment

4. Find bill by ID

5. Print bill receipt (by bill ID)

0. Back
```

Recommended flow for "Create bill + payment":

1. Ask appointmentId
2. Validate appointment exists + status is COMPLETED (or allow BOOKED if you choose, but document it)
3. Show consultation fee (from doctor)
4. Add medicines in loop: medicineCode + qty
5. Validate stock availability for each item
6. Show bill preview: consultationFee + medicinesTotal + totalAmount
7. Choose payment mode + confirm
8. Generate bill transactionally:

- insert bill header (CREATED)
- insert bill items
- insert payment (SUCCESS/FAILED)
- if SUCCESS → decrement medicine stock
- update bill status to PAID
- commit; rollback on failure

Reports Menu

```
--- Reports ---

1. Daily revenue summary (date)

2. Doctor performance report (date range)

3. Low stock medicines

4. Patient visit history (by phone)

0. Back
```