POLITÉCNICO DE SANTARÉM

POLITÉCNICO DE SANTARÉM
ESCOLA SUPERIOR DE GESTÃO E TECNOLOGIA

Licenciatura de Engenharia de Informática

Software Engineering

Software Architecture

Work done by

Beatriz Fernandes, 230001155

Bianca Fernandes, 230001154

January 1, 2026

Architectural Overview

1.1 Selected Architectural Style

The Hospital Management System is built upon a Layered Architecture. We selected this pattern to ensure a separation of concerns, maintainability and testability, key requirements for a system handling critical medical data.

In this architecture, components are organized into horizontal layers, where each layer performs a specific role and only communicates with the layer immediately below it. This ensures that changes in the user interface do not impact the core business logic and changes in data storage do not break the application rules

1.2 Major Components Structure and Layers

The system is divided into three primary layers:

Presentation Layer (UI):

- **Responsibility:** Handles user interactions and input validation.
- **Components:** Main.java (Console Entry)
- **Interaction:** Calls the Service Layer to execute tasks like "Login" or "Calculate Bill".

Business Logic Layer (Service):

- **Responsibility:** Contains the core functionality and rules of the hospital.
- **Components:**
    - **InteractionEngine:** Checks for dangerous drug interactions (e.g., Aspirin + Warfarin).
    - **BillingService:** Calculates costs using insurance strategies.
    - **ScheduleManager:** Handles conflict detection for appointments.
    - **Interaction:** Processes data received from the UI and utilizes the Domain Model.

Domain & Data Layer:

- **Responsibility:** Represents the core entities and manages their state.
- **Components:** User (and subclasses Doctor, Patient, Nurse, Pharmacist, Administrator), Appointment, Prescription.
- **Interaction:** These objects are passed between layers to carry data.

1.3 Architectural Constraints

- **Statelessness:** The Business Logic services are designed to be stateless. They do not hold user session data, ensuring the system can be easily restarted or scaled without losing transaction context.
- **Modularity:** The "Pharmacy" module is loosely coupled with the "Clinical" module. They communicate only via defined Interfaces, ensuring that updates to inventory logic do not break the doctor's dashboard.

2. Cross-Cutting Concerns

Beyond the horizontal layers, the architecture includes vertical components that impact the entire system.

- **Security and Authentication:** Implemented via a SecurityContext component that intercepts every request to the Business Layer. It verifies the user's role before executing methods. For example, if a Nurse object attempts to call deletePatient(), the Security component throws an AccessDeniedException.

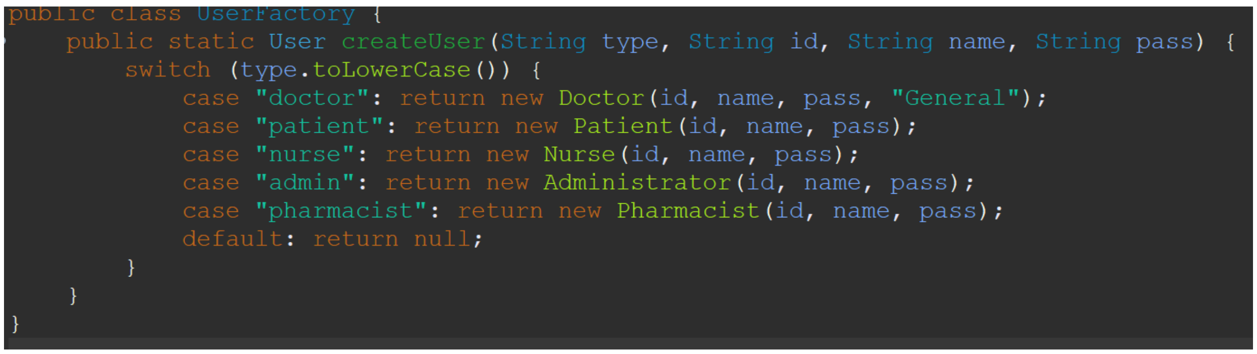
- **Exception Handling:** A global ExceptionHandler ensures that system crashes or database errors are translated into user-friendly messages. It also ensures that stack traces are logged internally but never shown to the end-user, maintaining security.

- **Logging:** A centralized AuditLogger records all critical modifications. This component is triggered by events in the Service Layer to satisfy the Non-Functional Requirement for traceability.

Design Patterns

Pattern 1: Factory Method

Pattern Name: Factory Method
Problem Solved: The system needs to create various types of users (Doctor, Patient, Nurse, Administrator and Pharmacist) based on input strings during login or registration, without coupling the client code to specific classes.
Implementation Location: Hospital/src/main/UserFactory
Code Evidence:



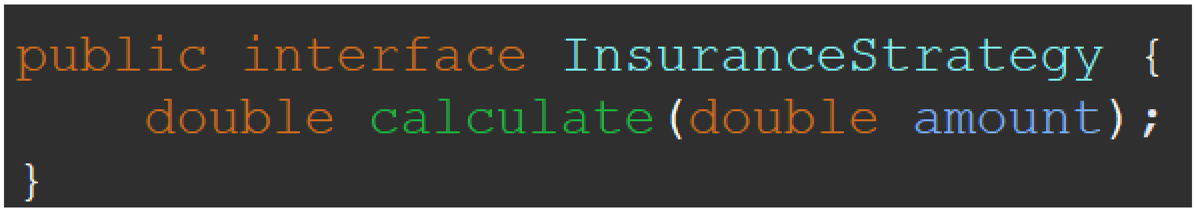
```
public class UserFactory {
    public static User createUser(String type, String id, String name, String pass) {
        switch (type.toLowerCase()) {
            case "doctor": return new Doctor(id, name, pass, "General");
            case "patient": return new Patient(id, name, pass);
            case "nurse": return new Nurse(id, name, pass);
            case "admin": return new Administrator(id, name, pass);
            case "pharmacist": return new Pharmacist(id, name, pass);
            default: return null;
        }
    }
}
```

Benefits: This centralizes object creation. If we need to add a "Surgeon" type later, we only modify the Factory, not the Login logic.

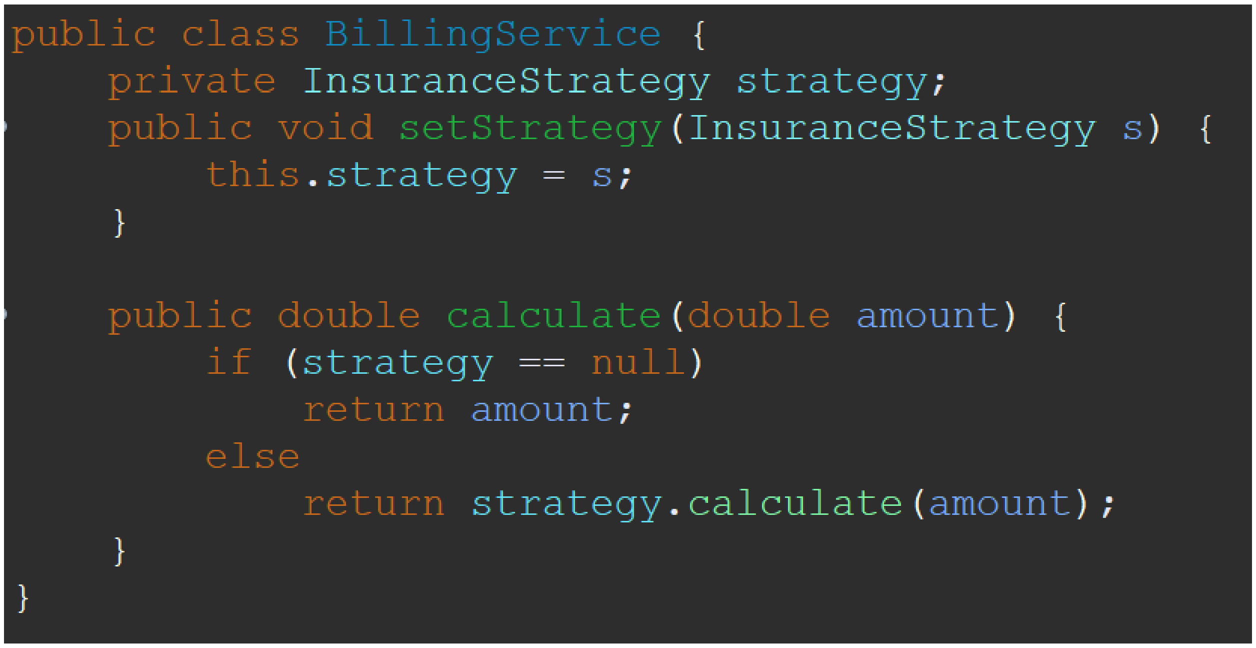
Pattern 2: Strategy Pattern

Pattern Name: Strategy Pattern
Problem Solved: Different patients have different insurance providers (Private vs Public), each with unique calculation formulas. Hardcoding these if-else checks inside the billing service would violate the Open-Closed Principle.
Implementation Location: Hospital/src/main/BillingService and Hospital/src/main/InsuranceStrategy

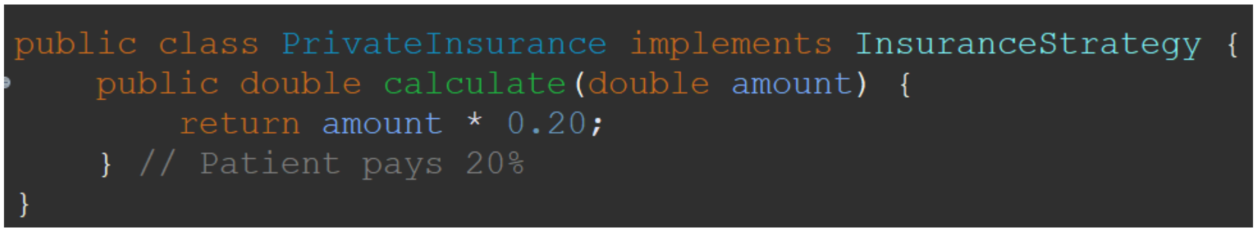
Code Evidence:



```java
public interface InsuranceStrategy {
    double calculate(double amount);
}
```



```java
public class BillingService {
    private InsuranceStrategy strategy;
    public void setStrategy(InsuranceStrategy s) {
        this.strategy = s;
    }

    public double calculate(double amount) {
        if (strategy == null)
            return amount;
        else
            return strategy.calculate(amount);
    }
}
```

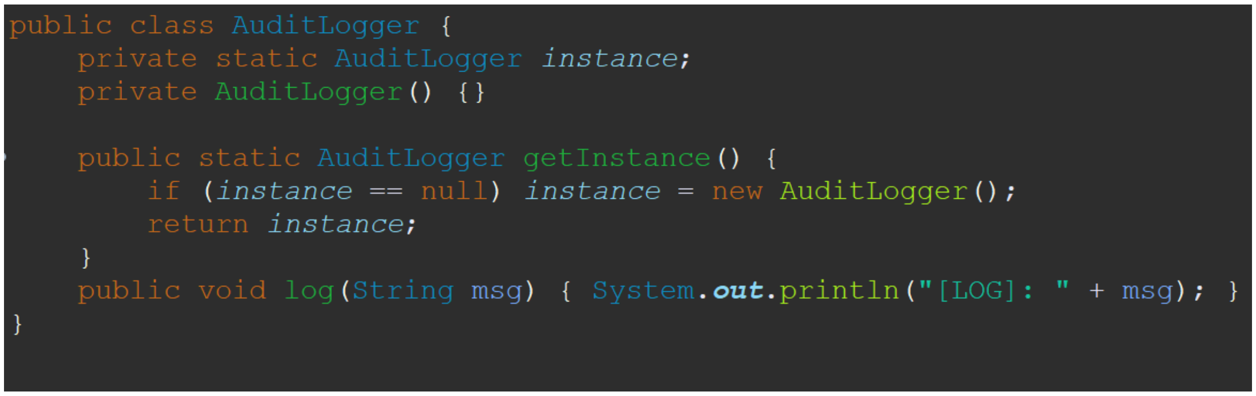


```java
public class PrivateInsurance implements InsuranceStrategy {
    public double calculate(double amount) {
        return amount * 0.20;
    } // Patient pays 20%
}
```

Benefits: We can swap insurance logic (bill.setStrategy(...)). This also makes it possible to modify the logic of a specific insurance type and create new ones (Open-Closed Principle).

Pattern 3: Singleton Pattern

Pattern Name: Singleton
Problem Solved: The system requires a centralized logging mechanism to record security events (like overridden safety alerts) to ensure a consistent audit trail.
Implementation Location: Hospital/src/main/AuditLogger
Code Evidence:

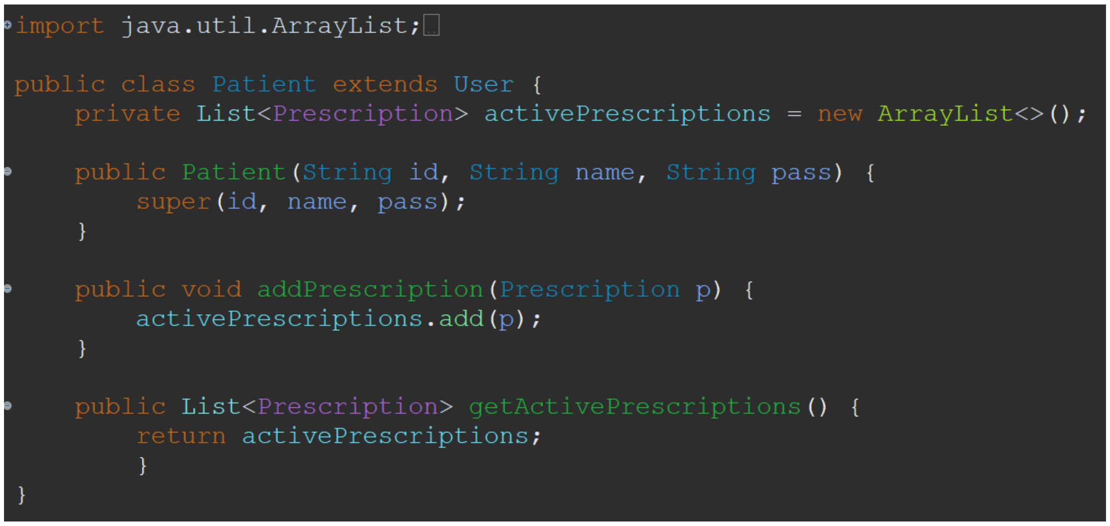


```java
public class AuditLogger {
    private static AuditLogger instance;
    private AuditLogger() {}

    public static AuditLogger getInstance() {
        if (instance == null) instance = new AuditLogger();
        return instance;
    }
    public void log(String msg) { System.out.println("[LOG]: " + msg); }
}
```

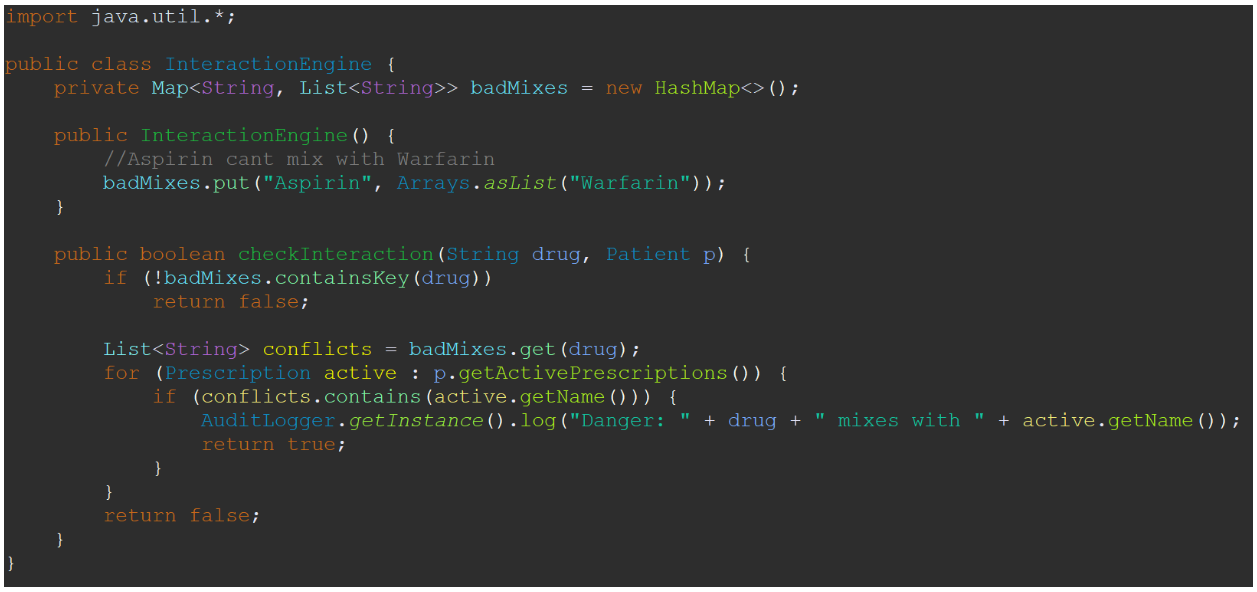
Benefits: Guarantees a single point of access for logging events across the entire application.

Solid Principles Application

Single Responsibility Principle (SRP)

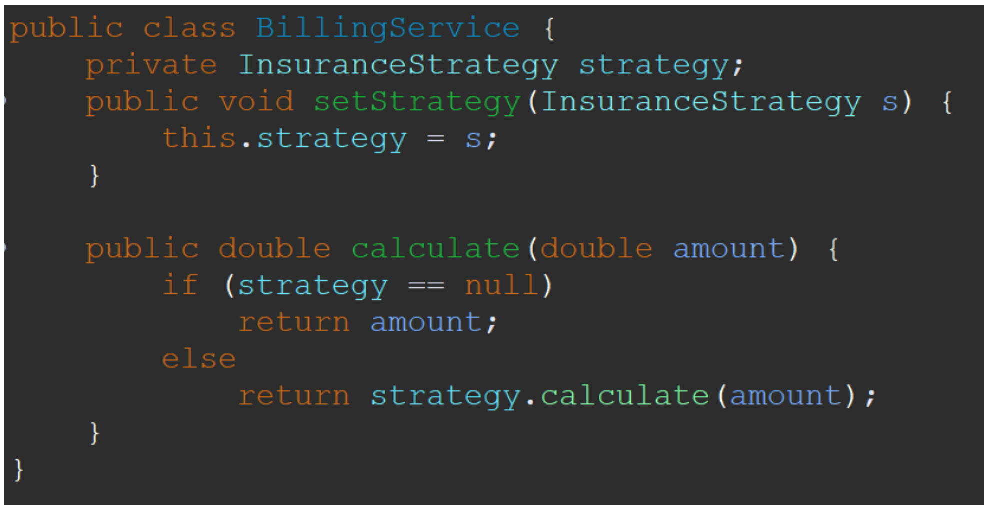


```java
import java.util.ArrayList;

public class Patient extends User {
    private List<Prescription> activePrescriptions = new ArrayList<>();

    public Patient(String id, String name, String pass) {
        super(id, name, pass);
    }

    public void addPrescription(Prescription p) {
        activePrescriptions.add(p);
    }

    public List<Prescription> getActivePrescriptions() {
        return activePrescriptions;
        }
}
```

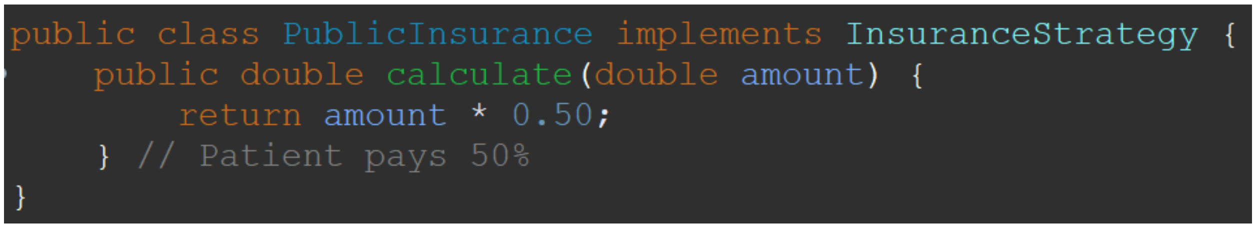


```java
import java.util.*;

public class InteractionEngine {
    private Map<String, List<String>> badMixes = new HashMap<>();

    public InteractionEngine() {
        //Aspirin cant mix with Warfarin
        badMixes.put("Aspirin", Arrays.asList("Warfarin"));
    }

    public boolean checkInteraction(String drug, Patient p) {
        if (!badMixes.containsKey(drug))
            return false;

        List<String> conflicts = badMixes.get(drug);
        for (Prescription active : p.getActivePrescriptions()) {
            if (conflicts.contains(active.getName())) {
                AuditLogger.getInstance().log("Danger: " + drug + " mixes with " + active.getName());
                return true;
            }
        }
        return false;
    }
}
```

Open-Closed Principle (OCP)

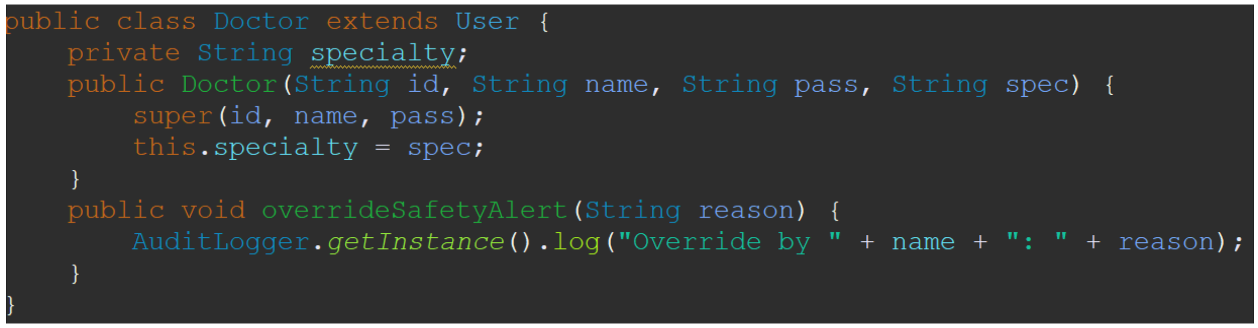


```
public class BillingService {
    private InsuranceStrategy strategy;
    public void setStrategy(InsuranceStrategy s) {
        this.strategy = s;
    }

    public double calculate(double amount) {
        if (strategy == null)
            return amount;
        else
            return strategy.calculate(amount);
    }
}
```



```
public class PublicInsurance implements InsuranceStrategy {
    public double calculate(double amount) {
        return amount * 0.50;
    } // Patient pays 50%
}
```

Interface Segregation Principle (ISP)



```
public class Doctor extends User {
    private String specialty;
    public Doctor(String id, String name, String pass, String spec) {
        super(id, name, pass);
        this.specialty = spec;
    }
    public void overrideSafetyAlert(String reason) {
        AuditLogger.getInstance().log("Override by " + name + ": " + reason);
    }
}
```

Technology Stack Justification

Programming Language: Java

Java was selected for its strong typing and Object-Oriented features, which are essential for implementing the complex **Design Patterns** (Strategy, Factory) required by the assignment.

Testing Framework: JUnit 5

Required to meet the project's quality assurance standards. JUnit 5 supports parameterized tests and detailed assertions, necessary for verifying the critical **Drug Interaction Engine**.

Integrated Development Environment (IDE): Eclipse

We selected **Eclipse** as our primary development environment due to the team's prior proficiency with the tool, which minimizes the learning curve and accelerates development velocity. As an IDE purpose-built for Java

Version Control: Git

Essential for team collaboration, allowing us to work on the Billing and Clinical modules simultaneously without file conflicts.