UE21CS352B - Object Oriented Analysis & Design using Java

Mini Project Report

"Hospital Management System"

Submitted by:

Jugal Kothari	**PES1UG21CS251**
Krish S Shah	**PES1UG21CS287**
Kshitij Agarwal	**PES1UG21CS292**
Maanasa Gowda	**PES1UG21CS312**

6th Semester E Section

Prof. Bhargavi Mokashi
Assistant Professor

January - May 2024

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

Problem Statement :

To develop a Hospital Management System to address the operational, administrative, and clinical needs of a modern hospital.

Models Used :

Architecture Patterns, Design Principles, and Design Patterns used in this project:

ARCHITECTURAL PATTERNS:

- MainController.java:
- Model-View-Controller (MVC) Pattern: This controller handles requests for main pages (`/`, `/main`), showing the main interface to users.

- PatientController.java:
- Model-View-Controller (MVC) Pattern: Handles patient-related views and actions (`/patients`, `/cancel`, `/confirmm`, etc.).

- SecurityConfig.java:
- Template Method Pattern: Extends `WebSecurityConfigurerAdapter` to customize Spring Security configurations.

DESIGN PATTERNS:

- PatientController.java:
- Dependency Injection (DI) Pattern: Uses `@Autowired` for injecting `appointmentService` dependency.

- appointment.java:

- Object-Relational Mapping (ORM) Pattern: Uses JPA annotations for mapping entity properties to database columns.

- appointmentService.java:

- Service Layer Pattern: Provides business logic for managing appointments (`save`, `delete`, `setConfirmation`, etc.).
- Dependency Injection (DI) Pattern: Injects `appointmentRepository` for data access.

- doctorController.java:

- Model-View-Controller (MVC) Pattern: Handles doctor-related views and actions (`/doctors`, `/createPrescription`, etc.).
- Dependency Injection (DI) Pattern: Injects `appointmentService` dependency.

- invoice.java:

- Object-Relational Mapping (ORM) Pattern: Maps invoice data to the database using JPA annotations.

- **invoiceservice.java:**

- Service Layer Pattern: Provides business logic for managing invoices (`save`, `view`, etc.).
- Dependency Injection (DI) Pattern: Injects `invoiceRepository` for data access.

- **prescription.java:**

- Object-Relational Mapping (ORM) Pattern: Maps prescription data to the database using JPA annotations.

- **prescriptionService.java:**
- Service Layer Pattern: Provides business logic for managing prescriptions (`save`, `findByPatientName`, etc.).
- Dependency Injection (DI) Pattern: Injects `prescriptionRepository` for data access.

- **receptionistController.java:**

- Model-View-Controller (MVC) Pattern: Handles receptionist-related views and actions (`/receptionistAppointments`, `/receptionistSchedule`, etc.).
- Dependency Injection (DI) Pattern: Injects services such as `appointmentService` and `EventJpaRepository`.

DESIGN PRINCIPLES :

Single Responsibility Principle (SRP):
- The controller classes (MainController, PatientController, doctorController, receptionistController) have specific responsibilities related to handling requests, interacting with services, and rendering views.

Open/Closed Principle (OCP):
- The application is designed with flexibility in mind, allowing for extensions without modifying existing code. For instance, adding new functionalities for doctors, patients, or receptionists can be done by extending existing classes or introducing new ones without altering the core logic.

Liskov Substitution Principle (LSP):
- Inheritance and polymorphism are utilized appropriately, such as extending WebSecurityConfigurerAdapter in SecurityConfig.java to customize security configurations while maintaining compatibility with Spring Security's framework.

Interface Segregation Principle (ISP):
- Interfaces are used where appropriate, such as Spring Data JPA repositories (appointmentRepository, invoiceRepository, prescriptionRepository) defining specific methods for data access, ensuring that clients only depend on the methods they need.

Dependency Inversion Principle (DIP):
- Dependency injection (DI) is extensively used throughout the application, such as autowiring services (appointmentService, invoiceservice, prescriptionService) into controllers, allowing for loose coupling and easier unit testing.

Don't Repeat Yourself (DRY):
- Code repetition is minimized by using Spring annotations (GetMapping, Autowired, etc.), inheritance (WebSecurityConfigurerAdapter, SpringBootServletInitializer), and service classes (appointmentService, invoiceservice, prescriptionService) to encapsulate common functionalities and promote code reusability.

Composition Over Inheritance:
- Instead of relying solely on class inheritance, the application utilizes composition, such as injecting dependencies (Autowired) into controllers and services, promoting flexibility and easier maintenance.

Github Repo Link :
https://github.com/kshitijagar/ooad_proj/tree/main