HEALTHCARE SERVICE MANAGEMENT SYSTEM PROPOSAL

Design Pattern CSE351

20200808030 NALAN GELİR

20200808054 MERYEM AHISKALI

TABLE OF CONTENTS

PROJECT GOALS

This project aims to create hospital management system software that facilitates effective communication and services between patients and hospital systems. Doctor, patient, radiologist, technician and laboratory are the entities in our software system. During the software development process, we meticulously applied design patterns. These design patterns were employed with the aim of making our software more organized, easier to maintain, and readily extensible.

Our goal is to establish a software platform for patients to schedule appointments, manage their treatment processes, and enable hospital staff to efficiently interact with patients.

Within the appointment system, our aim is to provide real-time notifications, enabling both patients and hospital staff to receive current and instant information regarding appointments. With this feature, possible loss of time for the patient or doctor will be prevented.

Additionally, this system will grant patients access to supplementary services during their treatment. Patients can request these services based on their needs and receive the necessary additional support throughout their treatments. They are billed on these additional services.

The objective of this system is to enhance the efficiency of both patients and hospital staff, thereby improving the quality of healthcare services.

During the development of this project, Java language was preferred to create the basic infrastructure of the software and implement the application. The reason why Java was preferred in the project is that it offers the opportunity to effectively implement design patterns and supports the basic principles of Object-Oriented Programming.

DESIGN PATTERNS

In the Healthcare Service Management Project, we utilized 4 design patterns.

Those patterns are:

1. Command Pattern
2. Singleton Pattern
3. Observer Pattern
4. Decorator Pattern

1. Command Pattern

The Command Pattern is a design pattern that packages a request as an object, thereby loosening the coupling between the requester and the executor, enhancing the system's flexibility. In the Hospital Management System, this pattern was used to structure the process of requesting laboratory tests between doctors and laborant, technician or radiologist .

The doctor's request for laboratory testing was represented through a command object, reducing the dependency between the client (doctor) and the recipient (laboratory, technician, or radiologist). In this process, the nurse took on the role of invoker, taking the command object and transmitting it to the laboratory or technician according to the test request, ensuring that the request was fulfilled. In addition, this pattern ensured that the request made by the doctor is personalized for the patient. This pattern also determines who will perform the test as well as the patient.

The doctor may request the following tests for the relevant patient: Cholesterol test, hormone test, hemogram test, MRI test, X-ray test, and eye radiology test.

2. Singleton Pattern

The Singleton design pattern is a structure that ensures only one instance of a class is created and provides a global access point to that instance. We decided to make a small change in this part during the implementation phase. In the previous stage, we decided to use this model for departments. When we looked at the system we developed, we decided to create departments as Enums because using this structure for departments would cause an unnecessary pile of code in the system we designed. Additionally, we saw that we could use this pattern more effectively in the appointment model we will use, and we implemented the singleton pattern in this way. Thanks to this model, appointments can be controlled from a single center.

3. Observer Pattern

The Observer Pattern is a design pattern in which multiple objects (observers) listen and react to changes in the state of another object (subject). In the context of appointment scheduling between patients and doctors, this model enabled real-time detection of changes in appointment statuses and notification of the relevant patient. In the previous report in the Notify section, we thought that the doctor should be informed if the patient cancels the appointment. However, when we moved on to the implementation phase, we thought that in such a case there was no need for the patient to inform the doctor.

In terms of compatibility with real life, if the doctor has to cancel the appointment, the relevant patients will be notified. As a result, we decided that the doctor will take on the role of subject and the patient will take on the role of observer.

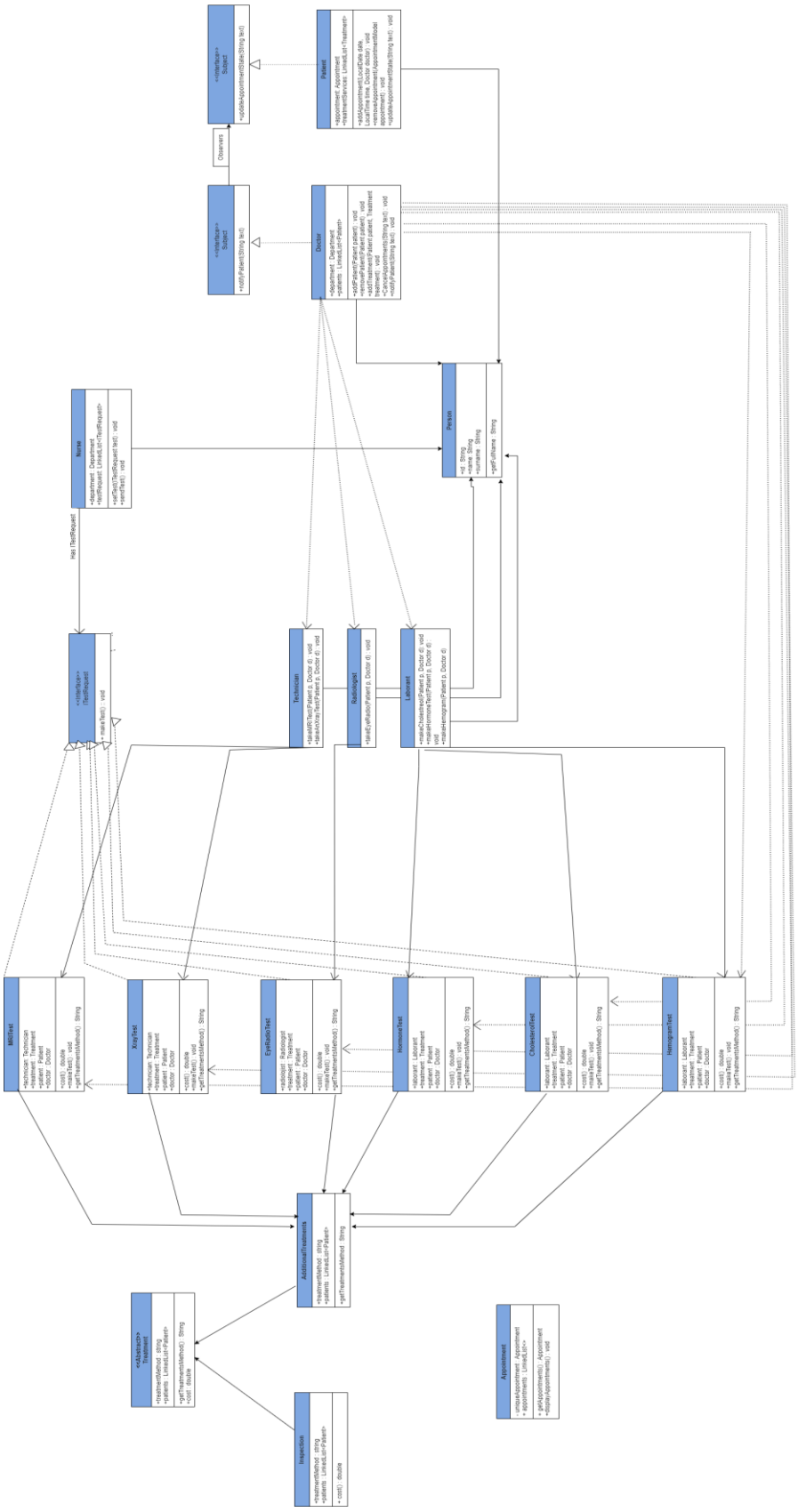
During the implementation phase, we used a slightly different approach than normal usage. The topic interface did not define the registerObserver() and removeObserver() methods. Because in the system, the doctor-patient relationship is already established when the doctor requests a test for the relevant patient. So the topic interface only defined the signature of the notifyObserver() method.

4. Decorator Pattern

The decorator pattern is a design pattern that uses the principle of wrapping to dynamically add or change the behavior of objects.

We provided various medical services, treatments and additional services within the scope of the Health Services Management System project. All kinds of tests that the doctor performs on his patient for treatment are considered additional services. We used different pricing rules for these. For example, although the basic examination has a standard price, additional tests and extra services performed in addition to this base price are charged by adding them to the initial examination fee. While applying this multi-layered pricing to the hospital discharge invoice, we managed this process by dynamically adding or changing the behavior of objects with the decorator pattern.

UML DIAGRAM


«interface»
Subject
+notifyAppointmentState(String text)
Observers
Patient
-appointment: Appointment
-treatmentService: LinkedList<Treatment>
+bookAppointment(LocalDate date, LocalTime time): void
+removeAppointment(AppointmentModel appointment): void
+updateAppointmentState(String text): void
«interface»
Subject
+notify(Patient3String text)
Doctor
-department: Department
-patients: LinkedList<Patient>
+addPatient(Patient patient): void
+removePatient(Patient patient): void
+addTreatment(Patient patient, Treatment treatment): void
+CancelingAppointments(String text): void
+notify(Patient3String text): void
Person
-id: String
-name: String
-surname: String
+getFullName: String
Nurse
-department: Department
-testRequest: LinkedList<TestRequest>
+setTest(TestRequest test): void
+sendTest(): void
Has TestRequest
«interface»
TestRequest
+ makeTest(): void
Technician
+takeAllrTest(Patient p, Doctor d): void
+takeAllrTest(Patient p, Doctor d): void
Receptionist
+takeEyeRate(Patient p, Doctor d): void
Laborant
+makeCholesterol(Patient p, Doctor d): void
+makeHormone(Patient p, Doctor d): void
+makeHemogram(Patient p, Doctor d): void
MriTest
-technician: Technician
-department: Department
-patient: Patient
-doctor: Doctor
+cost(): double
+getTreatmentsMethod(): String
XrayTest
-technician: Technician
-department: Department
-patient: Patient
-doctor: Doctor
+cost(): double
+getTreatmentsMethod(): String
EyeTestsTest
-receptionist: Receptionist
-treatment: Treatment
-patient: Patient
-doctor: Doctor
+cost(): double
+mainTest(): void
+getTreatmentsMethod(): String
HormoneTest
-laborant: Laborant
-treatment: Treatment
-patient: Patient
-doctor: Doctor
+cost(): double
+mainTest(): void
+getTreatmentsMethod(): String
CholesterolTest
-laborant: Laborant
-treatment: Treatment
-patient: Patient
-doctor: Doctor
+cost(): double
+mainTest(): void
+getTreatmentsMethod(): String
HemogramTest
-laborant: Laborant
-treatment: Treatment
-patient: Patient
-doctor: Doctor
+cost(): double
+mainTest(): void
+getTreatmentsMethod(): String
AdditionalTreatments
-treatmentMethod: string
-patients: LinkedList<Patient>
+getTreatmentsMethod: String
«Abstract»
Treatment
-treatmentMethod: string
-patients: LinkedList<Patient>
+getTreatmentsMethod(): String
+cost: double
Inspection
-treatmentMethod: string
-patients: LinkedList<Patient>
+ cost(): double
Appointment
-virtualAppointment: Appointment
+ appointments: LinkedList<>
+ getAppointments(): Appointment
+displayAppointments(): void