Design Patterns

Team Members:

- Shpëtim Shabanaj
- Artjol Zaimi
- Arjan Muka
- Eglis Braho
- Nikola Rigo
- Arlin Bashllari
- Marin Tartaraj

Shpëtim Shabanaj

Implementation of Builder and Strategy Design Patterns

In this class diagram, I have utilized two design patterns to ensure flexibility and scalability

Creational Design Pattern – Builder

To handle the step-by-step construction of complex Bill objects, I implemented the Builder design pattern.

- I created a BillBuilder interface that defines the necessary steps for building a Bill.
- ConcreteBillBuilder is a concrete implementation of BillBuilder that encapsulates the actual creation logic for a Bill object.
- To manage the construction process, I introduced a BillCreationDirector class. This class holds a reference to a BillBuilder object and is responsible for orchestrating the step-by-step creation process by calling the builder's methods in a specific sequence.
- This design allows different types of builders (like DetailedBillBuilder, SimpleBillBuilder) to be introduced in the future without modifying the existing logic.

The final result is an instance of the Bill class constructed through the defined sequence of method calls on the builder.

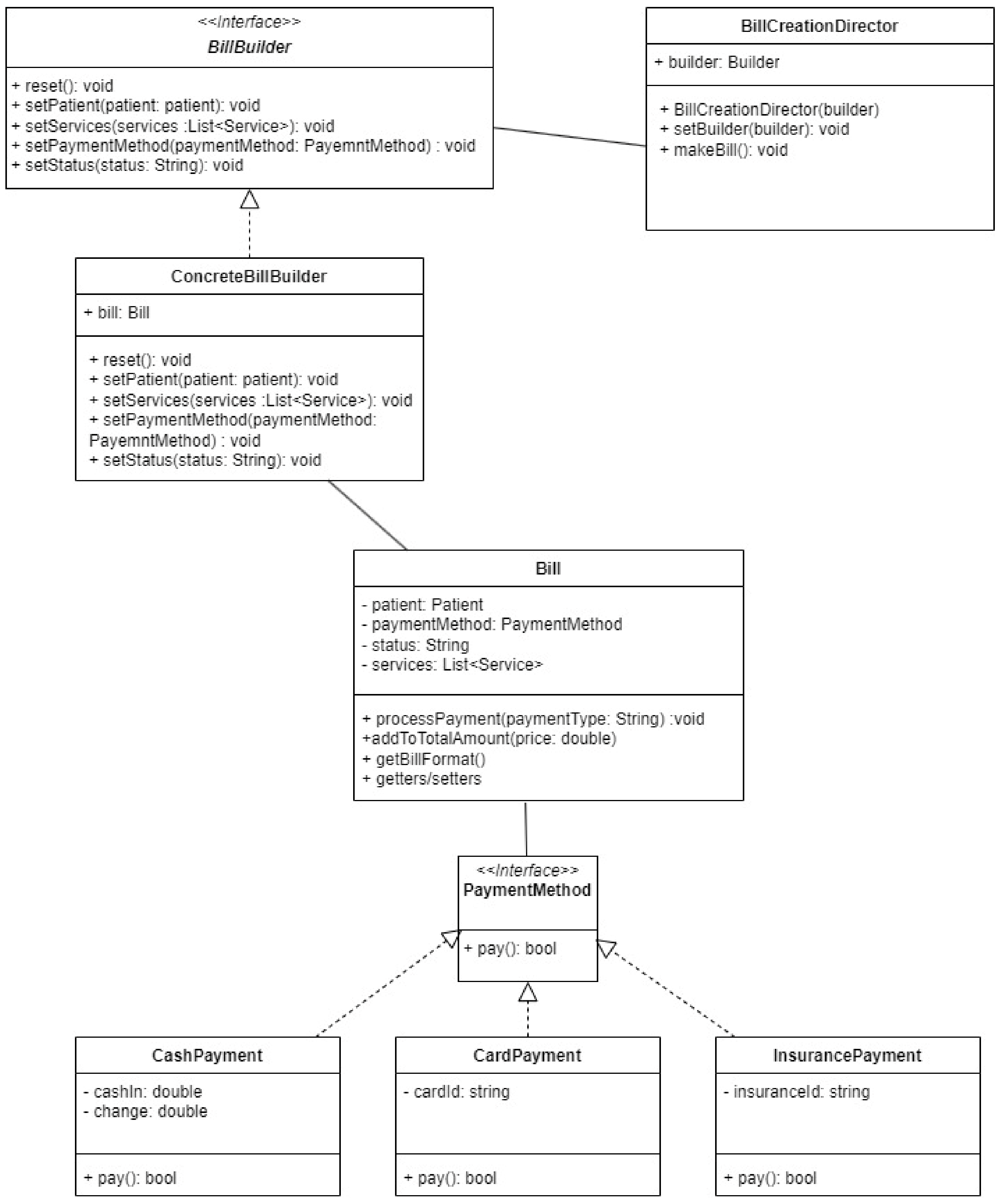
Behavioral Design Pattern – Strategy

Inside the Bill class, there is a reference to a PaymentMethod interface, representing a payment strategy.

- I implemented multiple concrete classes such as CreditCardPayment, CashPayment, and InsurancePayment, each providing its own implementation of the PaymentMethod interface.
- This follows the Strategy design pattern, enabling the Bill object to use different payment strategies at runtime without altering the core logic.
- This approach supports Open/Closed Principle — new payment methods can be added without changing the existing codebase.

Advantages

- The system can be easily extended with additional builder implementations or payment strategies, without violating SOLID principles or affecting existing functionality.


<<Interface>>
BillBuilder
+ reset(): void
+ setPatient(patient: patient): void
+ setServices(services :List<Service>): void
+ setPaymentMethod(paymentMethod: PayemntMethod) : void
+ setStatus(status: String): void
BillCreationDirector
+ builder: Builder
+ BillCreationDirector(builder)
+ setBuilder(builder): void
+ makeBill(): void
ConcreteBillBuilder
+ bill: Bill
+ reset(): void
+ setPatient(patient: patient): void
+ setServices(services :List<Service>): void
+ setPaymentMethod(paymentMethod: PayemntMethod) : void
+ setStatus(status: String): void
Bill
- patient: Patient
- paymentMethod: PaymentMethod
- status: String
- services: List<Service>
+ processPayment(paymentType: String) :void
+addToTotalAmount(price: double)
+ getBillFormat()
+ getters/setters
<<Interface>>
PaymentMethod
+ pay(): bool
CashPayment
- cashIn: double
- change: double
+ pay(): bool
CardPayment
- cardId: string
+ pay(): bool
InsurancePayment
- insuranceId: string
+ pay(): bool

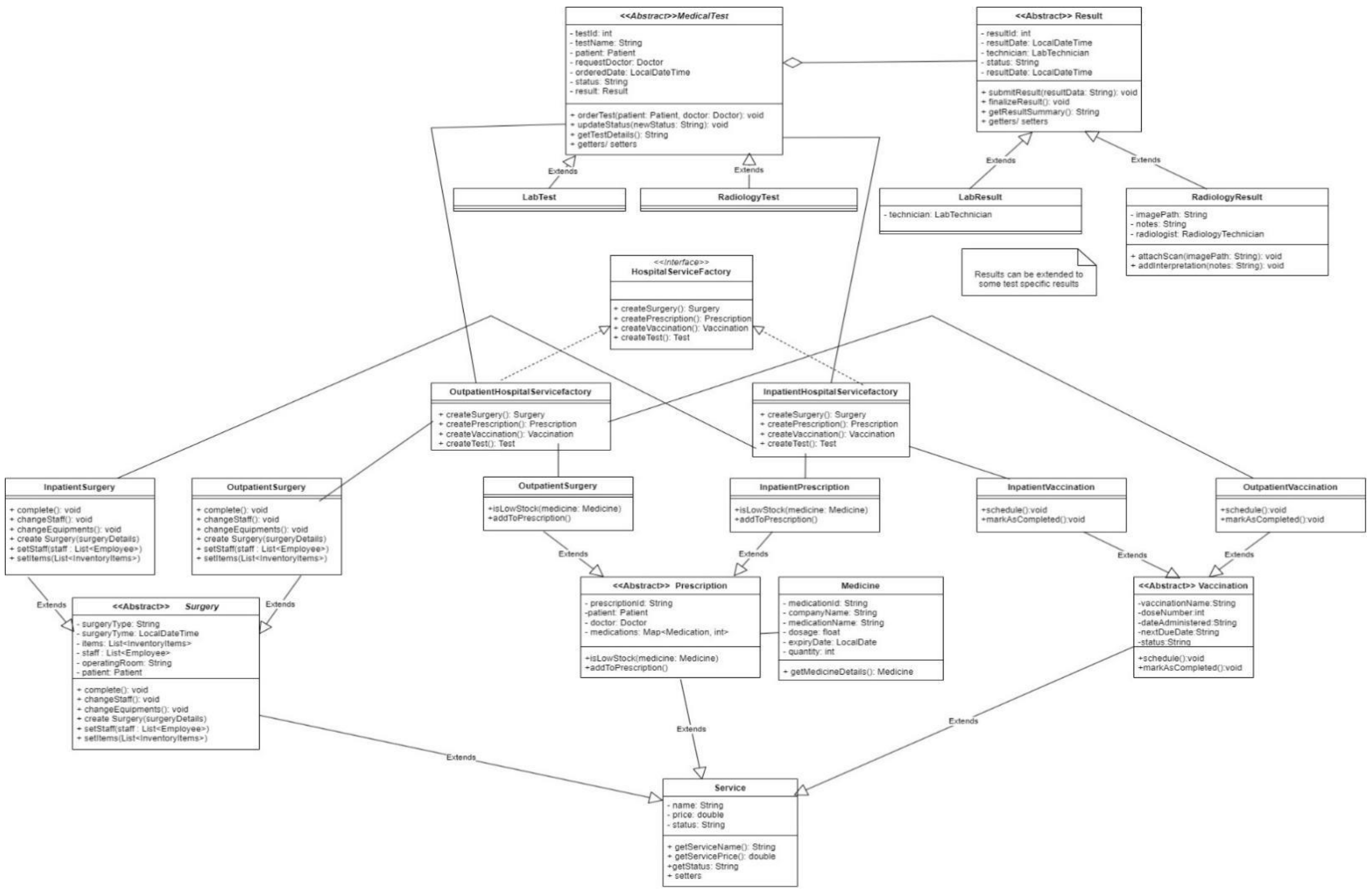
Shpëtim Shabanaj

Implementation of Abstract Factory pattern

In this design, I have implemented the Abstract Factory Pattern through an abstract class HospitalServiceFactory. This factory is responsible for creating families of related medical service objects for two distinct categories of patients: inpatients and outpatients.

Each concrete factory—InpatientHospitalServiceFactory and OutpatientHospitalServiceFactory—encapsulates the creation of related services such as surgeries, prescriptions, vaccinations, and medical tests tailored to the specific type of patient.

This pattern promotes extensibility and scalability by allowing new types of services or patient categories to be added without modifying existing code. It also provides a clear and organized structure for object creation, ensuring consistency between related service components. In practice, this is especially useful in hospital systems where service implementations differ significantly between inpatient and outpatient workflows.



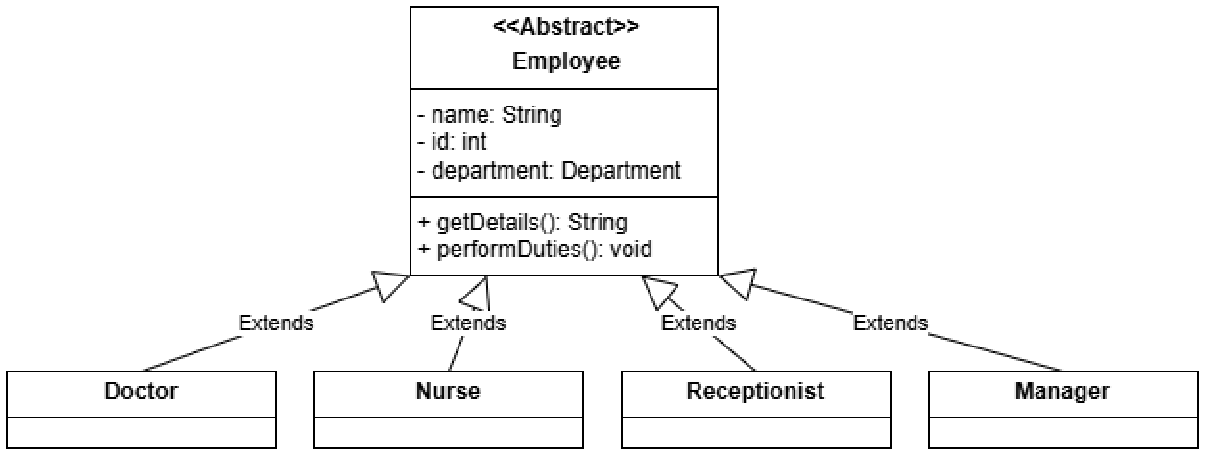
<<Abstract>>MedicalTest
- testId: int
- testName: String
- patient: Patient
- requestDoctor: Doctor
- orderedDate: LocalDateTime
- status: String
- result: Result
+ orderTest(patient: Patient, doctor: Doctor): void
+ updateStatus(newStatus: String): void
+ getTestDetails(): String
+ getters/ setters
<<Abstract>> Result
- resultId: int
- resultDate: LocalDateTime
- technician: LabTechnician
- status: String
- resultDate: LocalDateTime
+ submitResult(resultData: String): void
+ finalizeResult(): void
+ getResultSummary(): String
+ getters/ setters
Extends
Extends
Extends
Extends
LabTest
RadiologyTest
LabResult
- technician: LabTechnician
RadiologyResult
- imagePath: String
- notes: String
- radiologist: RadiologyTechnician
+ attachScan(imagePath: String): void
+ addInterpretation(notes: String): void
Results can be extended to some test specific results
<<Interface>>
HospitalServiceFactory
+ createSurgery(): Surgery
+ createPrescription(): Prescription
+ createVaccination(): Vaccination
+ createTest(): Test
OutpatientHospitalServicefactory
+ createSurgery(): Surgery
+ createPrescription(): Prescription
+ createVaccination(): Vaccination
+ createTest(): Test
InpatientHospitalServicefactory
+ createSurgery(): Surgery
+ createPrescription(): Prescription
+ createVaccination(): Vaccination
+ createTest(): Test
InpatientSurgery
+ complete(): void
+ changeStaff(): void
+ changeEquipments(): void
+ create Surgery(surgeryDetails)
+ setStaff(staff : List<Employee>)
+ setItems(List<InventoryItems>)
OutpatientSurgery
+ complete(): void
+ changeStaff(): void
+ changeEquipments(): void
+ create Surgery(surgeryDetails)
+ setStaff(staff : List<Employee>)
+ setItems(List<InventoryItems>)
OutpatientSurgery
+isLowStock(medicine: Medicine)
+addToPrescription()
InpatientPrescription
+isLowStock(medicine: Medicine)
+addToPrescription()
InpatientVaccination
+schedule():void
+markAsCompleted():void
OutpatientVaccination
+schedule():void
+markAsCompleted():void
Extends
Extends
Extends
Extends
Extends
Extends
<<Abstract>> Surgery
- surgeryType: String
- surgeryTyme: LocalDateTime
- items: List<InventoryItems>
- staff : List<Employee>
- operatingRoom: String
- patient: Patient
+ complete(): void
+ changeStaff(): void
+ changeEquipments(): void
+ create Surgery(surgeryDetails)
+ setStaff(staff : List<Employee>)
+ setItems(List<InventoryItems>)
<<Abstract>> Prescription
- prescriptionId: String
-patient: Patient
- doctor: Doctor
- medications: Map<Medication, int>
+isLowStock(medicine: Medicine)
+addToPrescription()
Medicine
- medicationId: String
- companyName: String
- medicationName: String
- dosage: float
- expiryDate: LocalDate
- quantity: int
+ getMedicineDetails(): Medicine
<<Abstract>> Vaccination
-vaccinationName:String
-doseNumber:int
-dateAdministered:String
-nextDueDate:String
-status:String
+schedule():void
+markAsCompleted():void
Extends
Extends
Extends
Service
- name: String
- price: double
- status: String
+ getServiceName(): String
+ getServicePrice(): double
+getStatus: String
+ setters

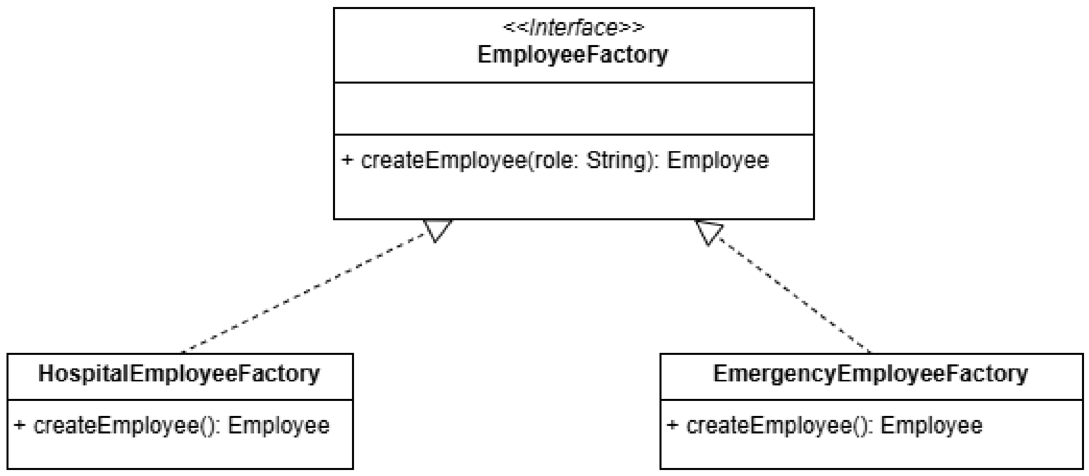
Arjan Muka

In this design, I have implemented the Factory Method Pattern through the use of an interface EmployeeFactory, which defines the method createEmployee(String role) for object creation. This pattern allows for the dynamic instantiation of different Employee types (such as Doctor, Nurse, Receptionist, Manager, etc.) based on input parameters like role or department.

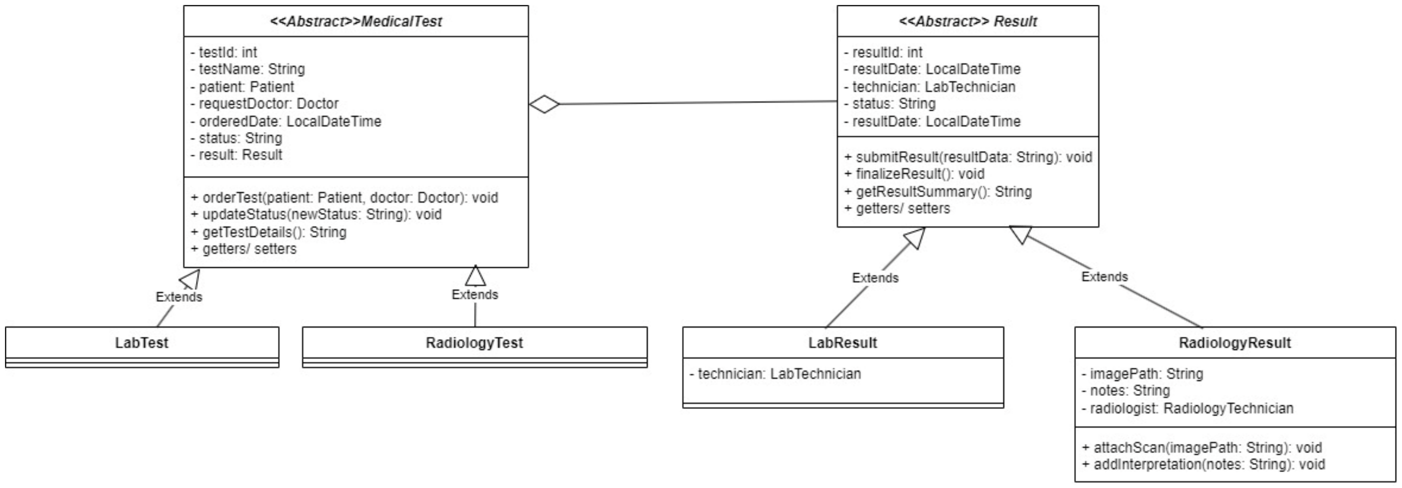
Each concrete factory—such as HospitalEmployeesFactory, EmergencyEmployeesFactory, and PharmacyEmployeesFactory—encapsulates the logic for creating specific types of employees. This design provides a clean separation of concerns between client code and object instantiation logic, which adheres to the Open/Closed Principle and eliminates the need for conditional statements like switch-case or if-else in client logic.

An abstract base class Employee, which defines common attributes (name, id, department, etc.) and behaviors (getDetails(), performDuties()). Several concrete subclasses (Doctor, Nurse, Receptionist, Pharmacist, etc.), each overriding performDuties() with specific responsibilities relevant to their roles. Multiple factories implementing the EmployeeFactory interface to create employees in different hospital contexts (general, emergency, pharmacy).




<<Interface>>
EmployeeFactory
+ createEmployee(role: String): Employee
HospitalEmployeeFactory
+ createEmployee(): Employee
EmergencyEmployeeFactory
+ createEmployee(): Employee

Nikola Rigo



<<Abstract>>MedicalTest
- testId: int
- testName: String
- patient: Patient
- requestDoctor: Doctor
- orderedDate: LocalDateTime
- status: String
- result: Result
+ orderTest(patient: Patient, doctor: Doctor): void
+ updateStatus(newStatus: String): void
+ getTestDetails(): String
+ getters/ setters
<<Abstract>> Result
- resultId: int
- resultDate: LocalDateTime
- technician: LabTechnician
- status: String
- resultDate: LocalDateTime
+ submitResult(resultData: String): void
+ finalizeResult(): void
+ getResultSummary(): String
+ getters/ setters
Extends
Extends
Extends
Extends
LabTest
RadiologyTest
LabResult
- technician: LabTechnician
RadiologyResult
- imagePath: String
- notes: String
- radiologist: RadiologyTechnician
+ attachScan(imagePath: String): void
+ addInterpretation(notes: String): void

Bridge Pattern Explanation (Test and Result):

In this design, I have applied the Bridge Pattern to decouple the abstraction (Test) from its implementation details (Result). The Test class is an abstract representation of a medical test (like lab test, radiology test), and it maintains an *aggregation* relationship with the Result class hierarchy, which represents the results of those tests.

By using this pattern, the Test abstraction and the Result implementation can evolve independently. For instance, different types of tests such as LabTest and RadiologyTest can be paired with different types of results like RadiologyResult (image-based) or lab-based reports without tightly coupling them. This provides a flexible and scalable solution where new test types or result formats can be introduced without altering the existing code structure.

This separation of concerns promotes code reuse, independent extensibility, and maintainability in the system. It is especially valuable in healthcare applications where diagnostic tests and result formats are diverse and subject to change over time.

Artjol Zaimi

Why Use the Proxy Pattern in My Project?

In a hospital system, there are multiple scenarios where **not all users should have the same level of access** to sensitive or resource-heavy objects. For example:

- **Patient medical data** must not be accessible to unauthorized staff.
- **System reports or documents** may need filtering or restricted generation rights.
- **Expensive database queries** (e.g., full financial reports) may need caching or delayed loading.

The **Proxy Pattern conveniently solves this** by placing a **"gatekeeper" class** in front of sensitive or resource-intensive classes. This allows us to:

- Apply **security policies** (e.g., only doctors can update a patient's medical profile).
- Add **logging and tracking** for accesses to protected objects.
- Delay the instantiation of heavyweight objects unless actually needed (lazy loading).

Proxy pattern — Example 1: PatientMedicalProfile

In the project, the PatientMedicalProfile class holds confidential medical data such as diagnosis history, test results, and prescriptions.

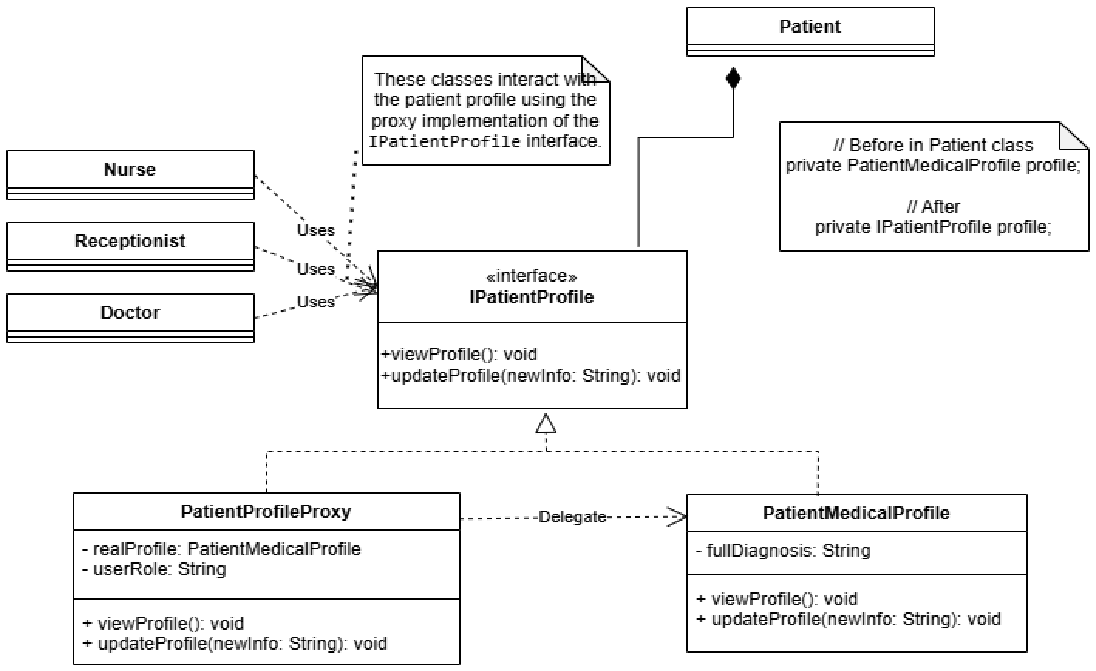
Initially, this class was directly referenced by the Patient class and could be accessed by roles like Doctor, Nurse, and Receptionist. However, giving direct access to such sensitive information violates key privacy and security principles.

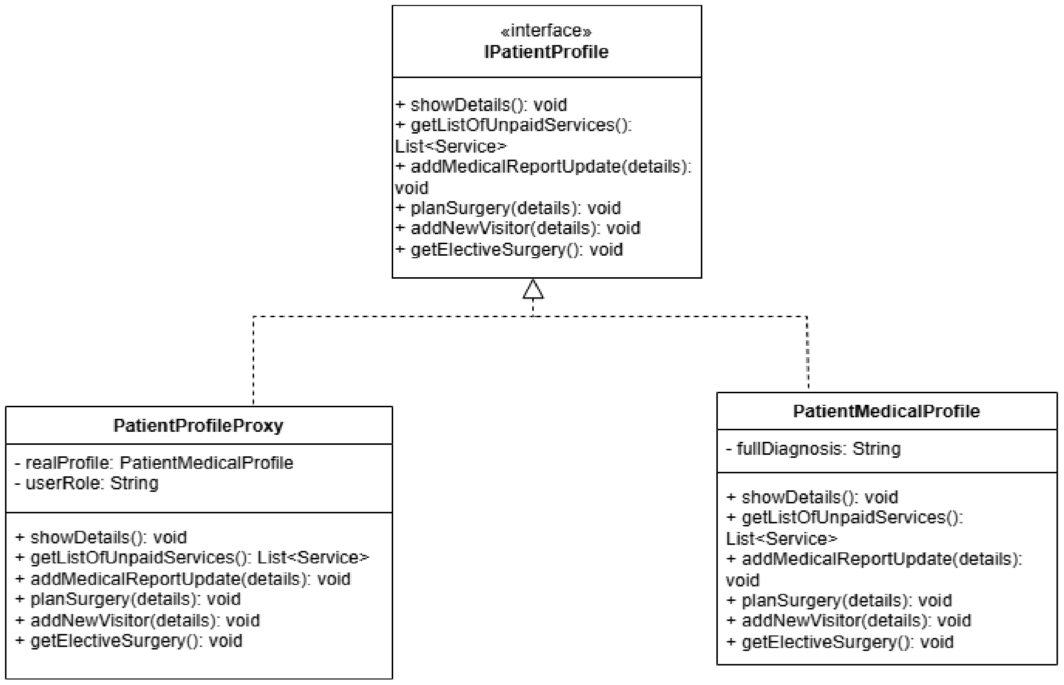
To solve this, I applied the Proxy Pattern by introducing a new class called PatientProfileProxy. This class implements the same interface (IPatientProfile) as the original profile and internally delegates actions to the real PatientMedicalProfile. The main advantage is that the proxy acts as a gatekeeper, adding role-based access control to determine whether the user can call viewProfile() or updateProfile() based on their role. For example, a Doctor or Nurse may have full access, while a Receptionist would be denied. In my implementation, classes like Doctor, Nurse, or Receptionist interact with the profile like this:

```
IPatientProfile profile = new PatientProfileProxy(patient.getProfile(), userRole);

profile.viewProfile();
```

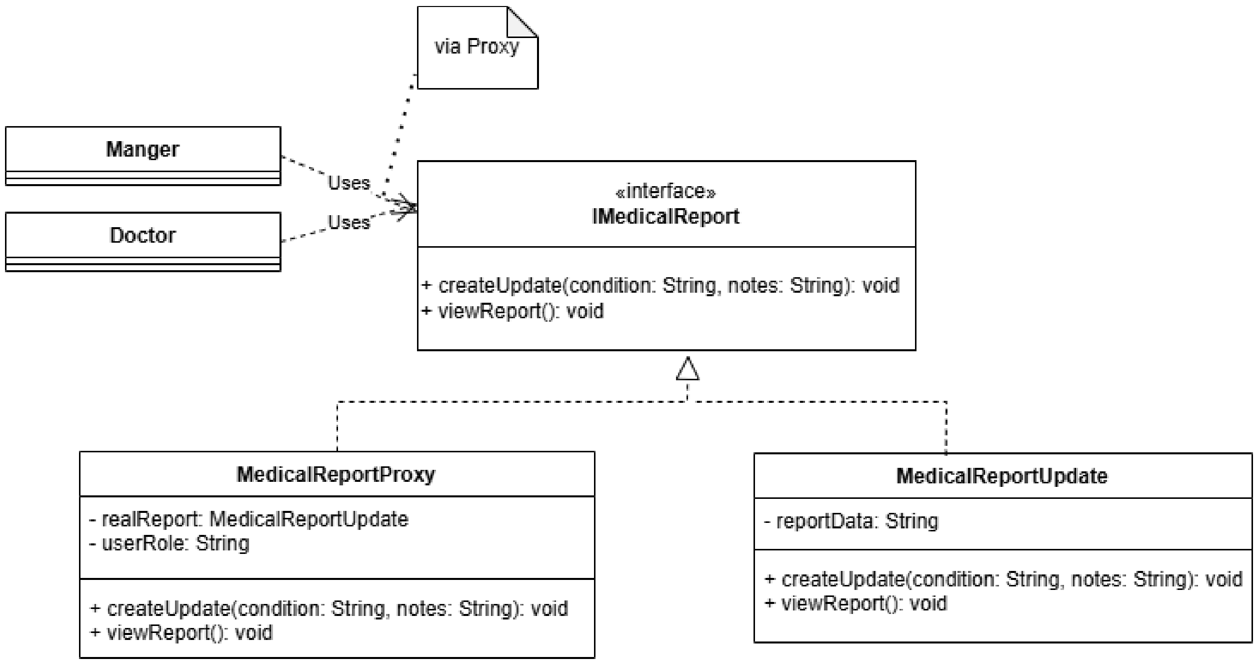
The important part is that these roles only rely on the interface (IPatientProfile) and are unaware of whether they're using the proxy or the real object. This abstraction ensures modularity and securely controls access to medical records without affecting the existing logic of the system.


Patient
These classes interact with the patient profile using the proxy implementation of the IPatientProfile interface.
// Before in Patient class
private PatientMedicalProfile profile;
// After
private IPatientProfile profile;
Nurse
Receptionist
Doctor
Uses
Uses
Uses
«interface»
IPatientProfile
+viewProfile(): void
+updateProfile(newInfo: String): void
PatientProfileProxy
- realProfile: PatientMedicalProfile
- userRole: String
+ viewProfile(): void
+ updateProfile(newInfo: String): void
Delegate
PatientMedicalProfile
- fullDiagnosis: String
+ viewProfile(): void
+ updateProfile(newInfo: String): void


«interface»
IPatientProfile
+ showDetails(): void
+ getListOfUnpaidServices(): List<Service>
+ addMedicalReportUpdate(details): void
+ planSurgery(details): void
+ addNewVisitor(details): void
+ getElectiveSurgery(): void
PatientProfileProxy
- realProfile: PatientMedicalProfile
- userRole: String
+ showDetails(): void
+ getListOfUnpaidServices(): List<Service>
+ addMedicalReportUpdate(details): void
+ planSurgery(details): void
+ addNewVisitor(details): void
+ getElectiveSurgery(): void
PatientMedicalProfile
- fullDiagnosis: String
+ showDetails(): void
+ getListOfUnpaidServices(): List<Service>
+ addMedicalReportUpdate(details): void
+ planSurgery(details): void
+ addNewVisitor(details): void
+ getElectiveSurgery(): void

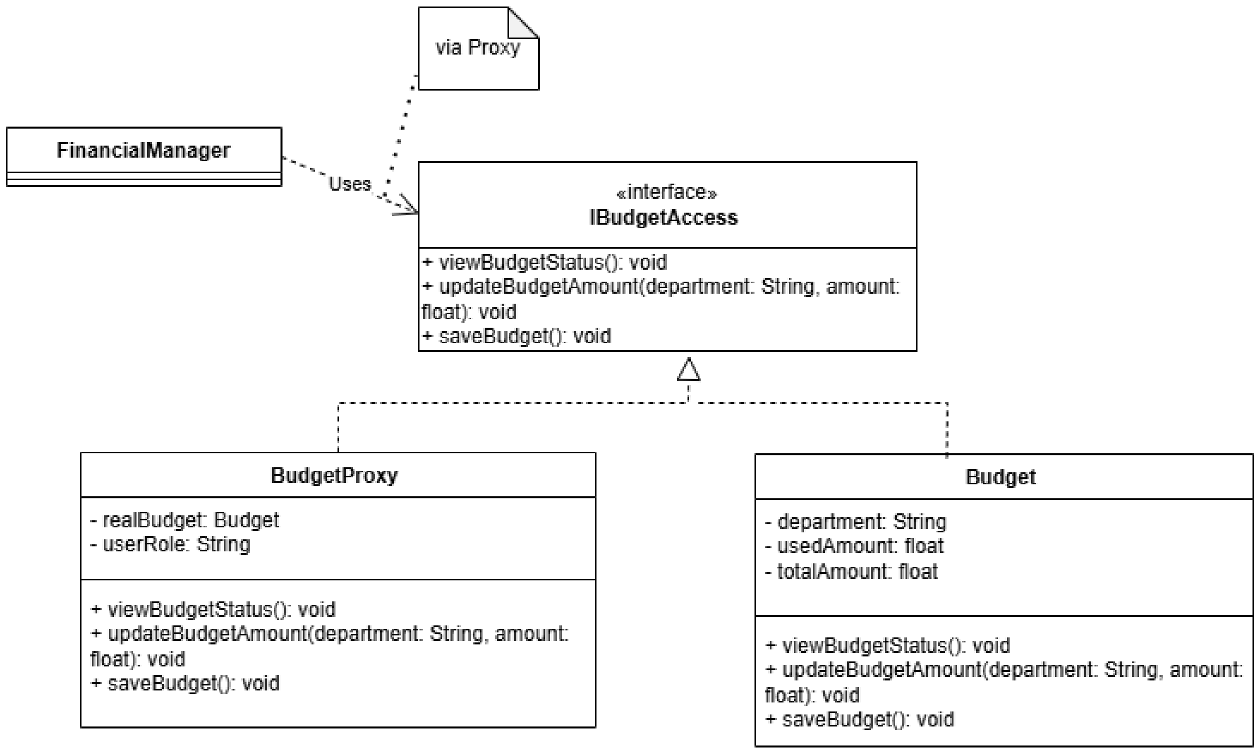
Proxy pattern — Example 2: MedicalReportUpdate

The **Proxy Pattern** is applied to the MedicalReportUpdate class in order to **secure sensitive patient information**. A proxy object, MedicalReportProxy, implements the same interface (IMedicalReport) and **controls access based on user role**. This ensures that only authorized users such as **Doctors** can update reports, while **Managers** may view them. Other roles are denied access. This approach keeps access logic separate from business logic, **increases modularity**, and complies with **data protection standards** in healthcare software.


via Proxy
Manger
Doctor
Uses
Uses
«interface»
IMedicalReport
+ createUpdate(condition: String, notes: String): void
+ viewReport(): void
MedicalReportProxy
- realReport: MedicalReportUpdate
- userRole: String
+ createUpdate(condition: String, notes: String): void
+ viewReport(): void
MedicalReportUpdate
- reportData: String
+ createUpdate(condition: String, notes: String): void
+ viewReport(): void

Proxy pattern — Example 3: Budget and financialtransaction

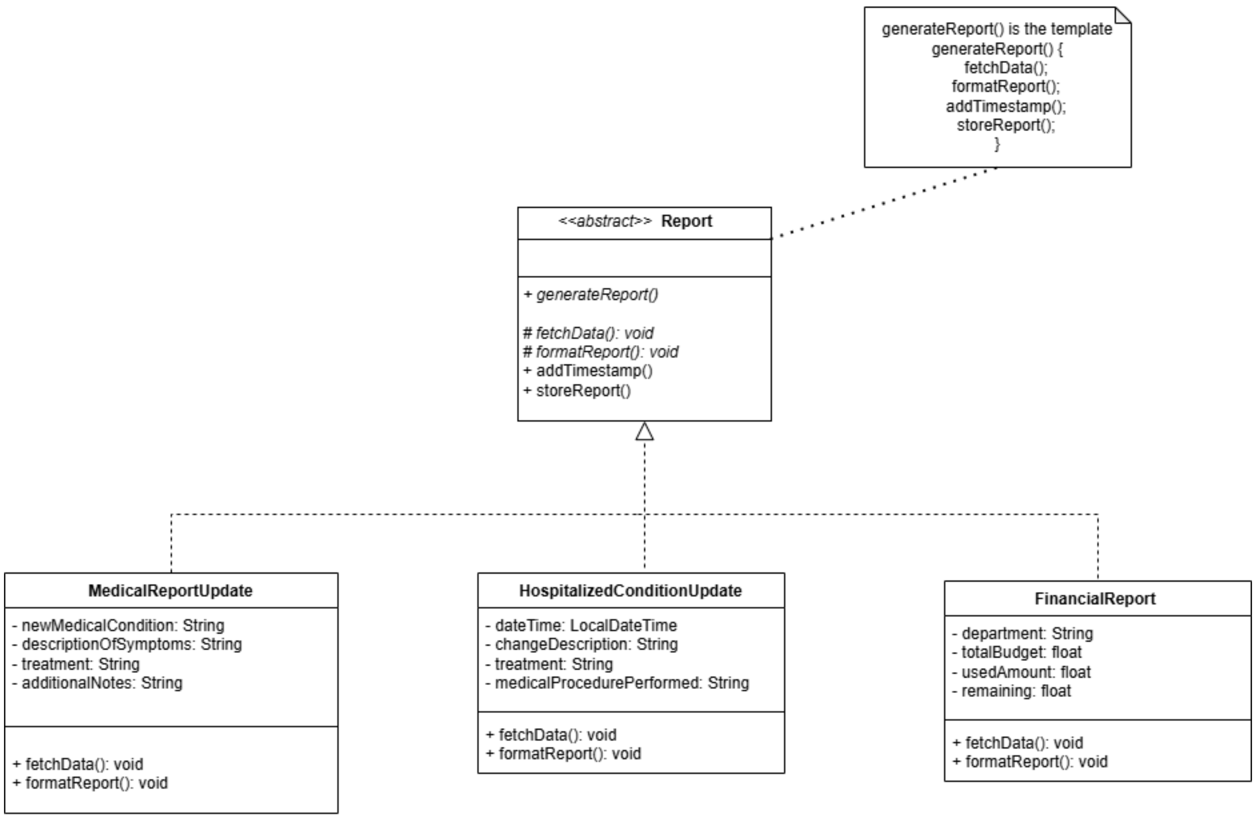
In the project, the Budget class manages sensitive financial data like department funds and usage. To make sure only authorized roles—like the Financial Manager—can modify this data, I applied the **Proxy Pattern**. I created a BudgetProxy that wraps the real Budget object and checks the user's role before allowing actions like updating or saving budget changes. Other roles can be limited to just viewing or even blocked entirely. This way, the proxy enforces access rules while keeping the actual budget logic clean and focused. Users only interact with the interface, without knowing whether it's the proxy or the real object behind it.


via Proxy
FinancialManager
Uses
«interface»
IBudgetAccess
+ viewBudgetStatus(): void
+ updateBudgetAmount(department: String, amount: float): void
+ saveBudget(): void
BudgetProxy
- realBudget: Budget
- userRole: String
+ viewBudgetStatus(): void
+ updateBudgetAmount(department: String, amount: float): void
+ saveBudget(): void
Budget
- department: String
- usedAmount: float
- totalAmount: float
+ viewBudgetStatus(): void
+ updateBudgetAmount(department: String, amount: float): void
+ saveBudget(): void

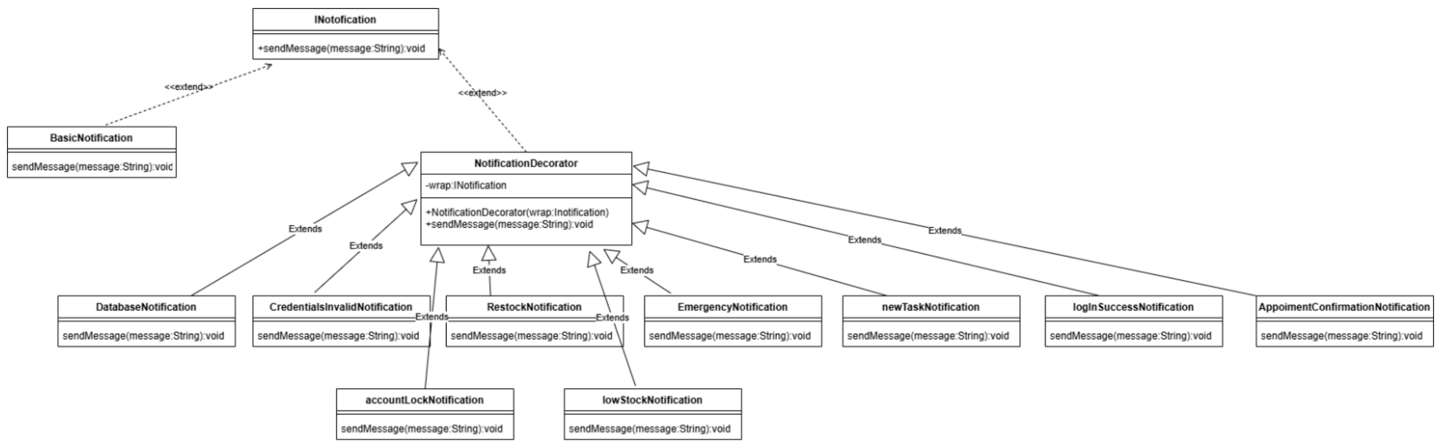
Template method pattern – Report Class Hierarchy

In the Hospital Management System, the report generation process was originally handled independently by each report class. For example, MedicalReportUpdate, HospitalizedConditionUpdate, and FinancialReport each had their own way of fetching data, formatting output, and storing results. This caused duplicated logic, inconsistencies in structure, and made it difficult to manage or scale the reporting system. To solve this, I applied the **Template Method Pattern** by creating an abstract class called Report which defines a fixed method called generateReport(). This method outlines the full workflow: fetchData(), formatReport(), addTimestamp(), and storeReport()—with the first two marked as abstract so each subclass can provide its own specific logic.

By doing this, I was able to ensure that every report follows the same structure while keeping the flexibility to change only what's necessary. For example, MedicalReportUpdate fetches diagnosis and treatment data, while FinancialReport pulls budget and transaction details—but both follow the same template when generating their report. The shared methods like addTimestamp() and storeReport() are handled in the base class, avoiding repetition. This not only made the code cleaner and more maintainable, but also aligned with the **Open/Closed Principle**, since I can now add new report types without changing the base logic. It's a scalable, modular solution that fits perfectly into a system where structured yet flexible workflows are essential.



Eglis Braho


INotofication
+sendMessage(message:String):void
<<extend>>
<<extend>>
BasicNotification
sendMessage(message:String):void
NotificationDecorator
-wrap:INotification
+NotificationDecorator(wrap:Inotification)
+sendMessage(message:String):void
Extends
DatabaseNotification
sendMessage(message:String):void
CredentialsInvalidNotification
sendMessage(message:String):void
RestockNotification
sendMessage(message:String):void
EmergencyNotification
sendMessage(message:String):void
newTaskNotification
sendMessage(message:String):void
logInSuccessNotification
sendMessage(message:String):void
AppoimentConfirmationNotification
sendMessage(message:String):void
accountLockNotification
sendMessage(message:String):void
lowStockNotification
sendMessage(message:String):void

Decorator Design Pattern

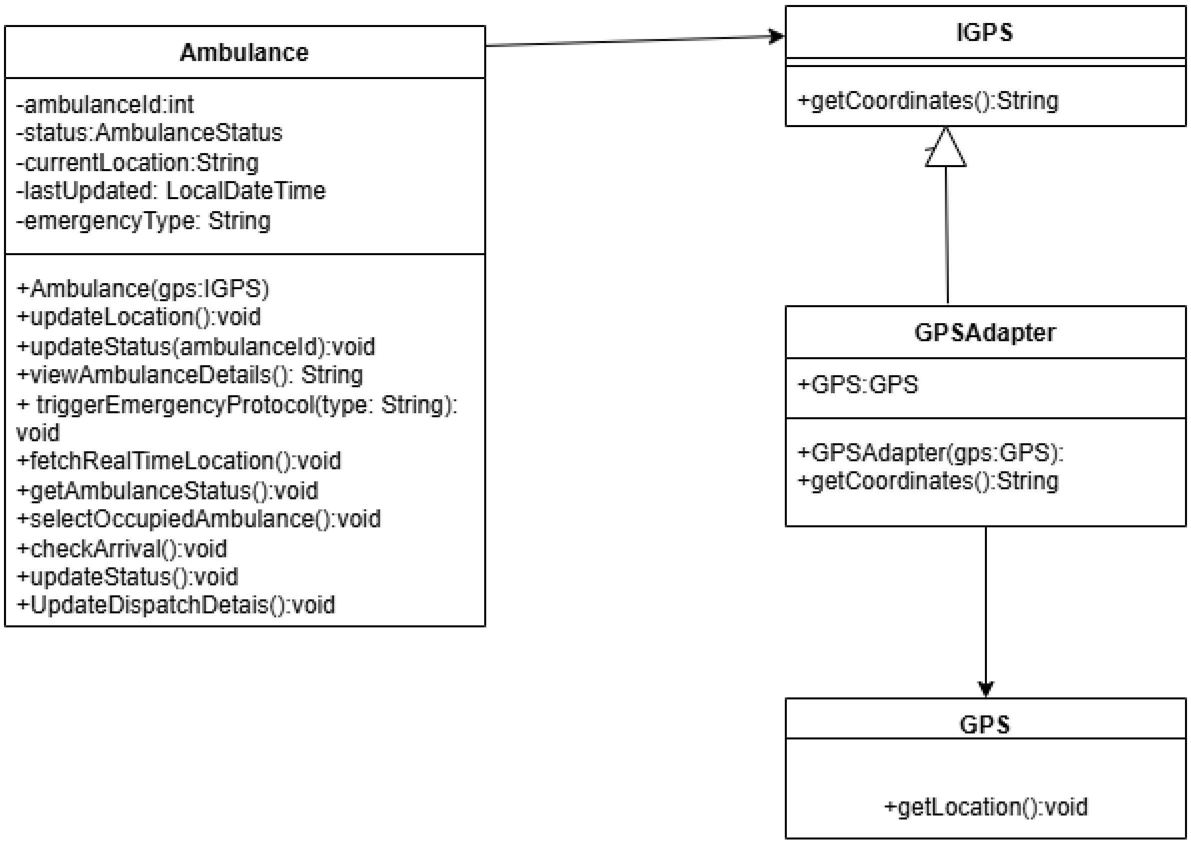
In this design, the Decorator Design Pattern is used to extend the functionality of the INotification interface without modifying its existing implementations. The BasicNotification class provides the core notification behavior, while additional features are dynamically added using decorators.

The abstract class NotificationDecorator implements the INotification interface and serves as a base for all concrete decorators. Specific decorators such as DatabaseNotification, CredentialsInvalidNotification, and others extend NotificationDecorator and enhance or modify the behavior of the wrapped notification object.

This pattern is particularly useful in scenarios where multiple notification types or behaviors (like logging to a database, alerting invalid credentials, sending to different channels) need to be composed flexibly and at runtime. It allows behaviors to be layered without changing the original code, promoting open/closed principle adherence.

By using the Decorator Pattern, the system becomes highly extensible, maintainable, and modular, enabling developers to mix and match different notification features in a clean and reusable way.

Eglis Braho



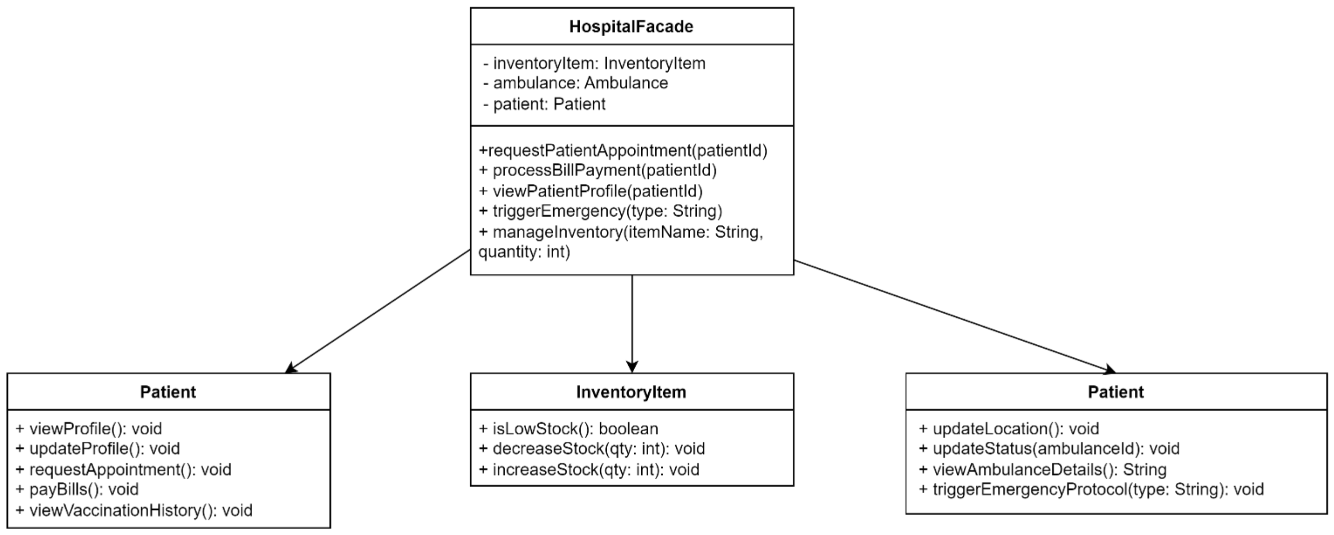
In this design, the Adapter Design Pattern is used to integrate an existing GPS class with the Ambulance system, which expects navigation functionality through the IGPS interface.

The GPS class represents a third-party or legacy component that provides location and routing services, but it does not implement the expected IGPS interface. To resolve this incompatibility without modifying the original GPS class, a GPSAdapter is introduced. This adapter class implements the IGPS interface and internally wraps an instance of GPS, translating calls from the interface into corresponding calls understood by the GPS object.

The Ambulance class depends on the IGPS abstraction, allowing it to work seamlessly with any future GPS implementations that also conform to this interface. The adapter thus enables reuse of existing functionality, ensures interface compatibility, and adheres to the dependency inversion principle by decoupling the high-level ambulance logic from the low-level GPS details.

This pattern is especially useful in systems that need to integrate with external or pre-existing components without altering their source code, providing a clean and flexible integration strategy.

Marin Tartaraj

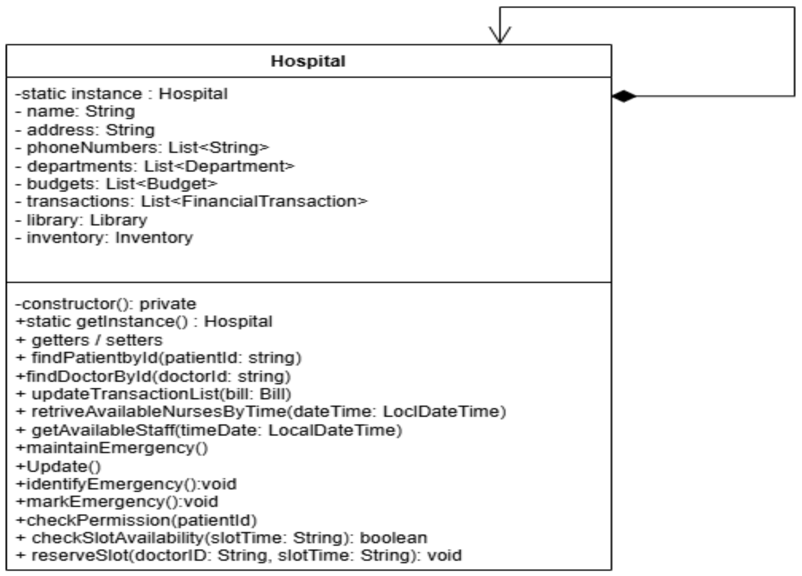

HospitalFacade
- inventoryItem: InventoryItem
- ambulance: Ambulance
- patient: Patient
+requestPatientAppointment(patientId)
+ processBillPayment(patientId)
+ viewPatientProfile(patientId)
+ triggerEmergency(type: String)
+ manageInventory(itemName: String, quantity: int)
Patient
+ viewProfile(): void
+ updateProfile(): void
+ requestAppointment(): void
+ payBills(): void
+ viewVaccinationHistory(): void
InventoryItem
+ isLowStock(): boolean
+ decreaseStock(qty: int): void
+ increaseStock(qty: int): void
Patient
+ updateLocation(): void
+ updateStatus(ambulanceId): void
+ viewAmbulanceDetails(): String
+ triggerEmergencyProtocol(type: String): void

In this design, the **Facade Design Pattern** is used to provide a simplified interface for interacting with the complex subsystems of the hospital management system. These subsystems—such as InventoryItem, Ambulance, and Patient each handle specific responsibilities, but exposing all of them directly to the client would result in high coupling and increased complexity.

To address this, a HospitalFacade class is introduced, which encapsulates and coordinates these subsystems through a unified set of high-level methods like requestPatientAppointment() or triggerEmergency(). This abstraction hides the internal details of the subsystems, allowing external components (like UI or external services) to interact with the hospital system easily and consistently.

By decoupling client code from the individual classes and their interactions, the facade improves maintainability, promotes modular design, and enables better scalability as the system grows.

Arlin Bashllari


Hospital
-static instance : Hospital
- name: String
- address: String
- phoneNumbers: List<String>
- departments: List<Department>
- budgets: List<Budget>
- transactions: List<FinancialTransaction>
- library: Library
- inventory: Inventory
-constructor(): private
+static getInstance() : Hospital
+ getters / setters
+ findPatientbyId(patientId: string)
+findDoctorById(doctorId: string)
+ updateTransactionList(bill: Bill)
+ retriveAvailableNursesByTime(dateTime: LoclDateTime)
+ getAvailableStaff(timeDate: LocalDateTime)
+maintainEmergency()
+Update()
+identifyEmergency():void
+markEmergency():void
+checkPermission(patientId)
+ checkSlotAvailability(slotTime: String): boolean
+ reserveSlot(doctorID: String, slotTime: String): void

In this UML diagram, the `Hospital` class is designed as a **Singleton**, which ensures that **only one instance of the hospital exists in the application**. Here's how it's implemented:

Private Static Instance:

```
- static instance: Hospital
```

This holds the single instance of the class.

Private Constructor:

```
- constructor(): private
```

The constructor is private so that no external class can instantiate the `Hospital` class directly.

Public Static Accessor:

```
+ static getInstance(): Hospital
```

This method returns the single instance of the `Hospital` class, creating it if it doesn't already exist.

Single Point of Access
A hospital system should have **one central controller** that manages resources like departments, staff, transactions, inventory, etc. Singleton ensures centralized management.**Shared Global State**
Resources like doctors, patients, inventory, and budgets should be managed **consistently** across the entire application. Singleton ensures that all parts of the system refer to the **same data**.**Resource Optimization**
Since hospital data is often resource-intensive, creating multiple instances would be inefficient. The singleton avoids unnecessary duplication and ensures **resource efficiency**.**Thread Safety and Coordination**
In systems where multiple threads or modules might interact with hospital data (e.g., booking slots or updating budgets), using a singleton helps **synchronize access**.

- ## Benefits of using this design pattern

Centralized control over the hospital system

Ensures consistency in data handling and updates

Easy to maintain and extend as hospital requirements grow

Prevents issues from multiple instances conflicting with one another