Hospital Management System

ERSOY EFE URUK – IPEK TEKIN- UMUTCAN BERK HASRET
20140601051-20140601048-20140601028

SE311 | Software Architecture | 18.05.2018

1. Pattern Choices

1.Abstact Factory Pattern: Hospital class is Abstract Factory in this system and object creation delegation to Endocrinology and Cardiology. There are different types of tests here. (RadiologicalTest and LabTest). Usage of the Abstract Factory pattern here will enable us to create different tests from different families without showing their inner parts.

2.Command Pattern: This pattern is also a straightforward request as well. The word "orders" in scenario suggests a doctor asks a test and the corresponding department performs the test. To be able to provide this, we use command pattern to encapsulate the command and forward the test request to the LabAttendant which performs as an Invoker here.

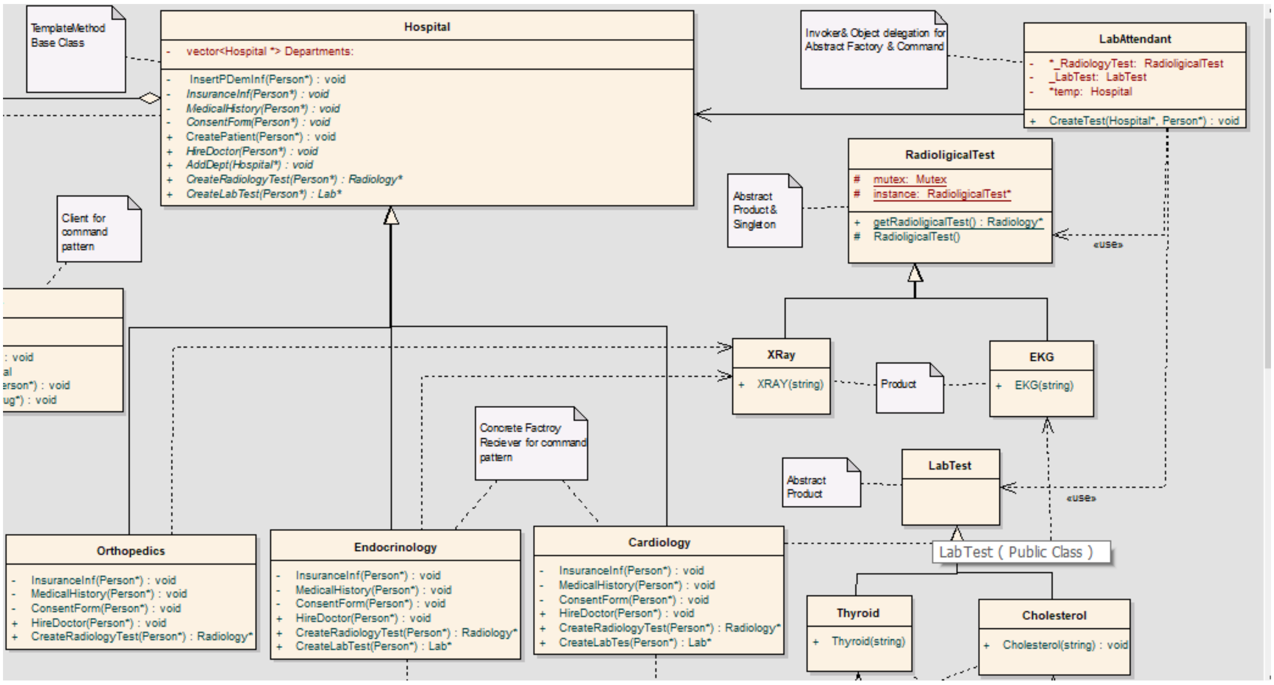
3.Singleton Pattern: We believe this was a straightforward request. Scenario says there should be only one Radiology department. The usage of the singleton pattern would provide a restriction for object creation; here, in our case, it's the creation of RadiologicalTest class.

4.Template Pattern: Creating patient process is similar for every department. There are only few differences between them. Usage of the Template pattern here will provide the following benefit; We do not need to implement the same function repeatedly. Instead, we'll be implementing the member functions that will change depending on the department.

5.Observer Pattern: In the scenario, we were told after a prescription of a drug to a patient, patient should be aware of any kind of changes in case of side effects for the drug. So, in order to solve this situation, Observer pattern would be the best choice since we will be able to attach patients as an observer to the notification network of the subject which is, in this case drug.

2. Participant of Each Pattern

I.Abstract Factory Pattern


TemplateMethod Base Class
Hospital
- vector<Hospital *> Departments:
- InsertPDemInf(Person*) : void
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ CreatePatient(Person*) : void
+ HireDoctor(Person*) : void
+ AddDept(Hospital*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTest(Person*) : Lab*
Invoker& Object delegation for Abstract Factory & Command
LabAttendant
- *_RadiologyTest: RadioligicalTest
- _LabTest: LabTest
- *temp: Hospital
+ CreateTest(Hospital*, Person*) : void
RadioligicalTest
# mutex: Mutex
# instance: RadioligicalTest*
+ getRadioligicalTest() : Radiology*
# RadioligicalTest()
Abstract Product & Singleton
«use»
Client for command pattern
: void
al
erson*) : void
ug*) : void
XRay
+ XRAY(string)
Product
EKG
+ EKG(string)
Concrete Factroy Reciever for command pattern
Abstract Product
LabTest
«use»
LabTest (Public Class)
Orthopedics
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
Endocrinology
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTest(Person*) : Lab*
Cardiology
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTes(Person*) : Lab*
Thyroid
+ Thyroid(string)
Cholesterol
+ Cholesterol(string) : void

- PARTICIPANS -

Abstract Factory: Hospital

Concrete Factory: Endocrinology & Cardiology & Orthopedics

Abstract Product: RadiologicalTest & LabTest

Concrete Product: X-RAY - EKG & Thyroid - Cholesterol

Client: LabAttendant

Hospital: Hospital class is the interface that responsible for the object creation delegation for the creation of both Radiological and Lab Tests with its two functions. (CreateRadiologicalTest(Person* patient), CreateLabTest(Person* patient)). The tests differ from clinic to clinic.

Factory Classes: Cardiology, Endocrinology and Orthopedics:

Cardiology consists CreateRadiologicalTest(Person* patient) to create EKG Tests and CreateLabTest(Person* patient) for Cholesterol Tests. Endocrinology consists CreateRadiologicalTest(Person* patient) to create XRAY Tests and CreateLabTest(Person* patient) for Thyroid Tests. Orthopedics consists CreateRadiologicalTest(Person* patient) to create XRAY. It doesn't create any Lab Tests. (We did creations as scenario suggests)

Abstract Product: RadiologicalTest and LabTest.

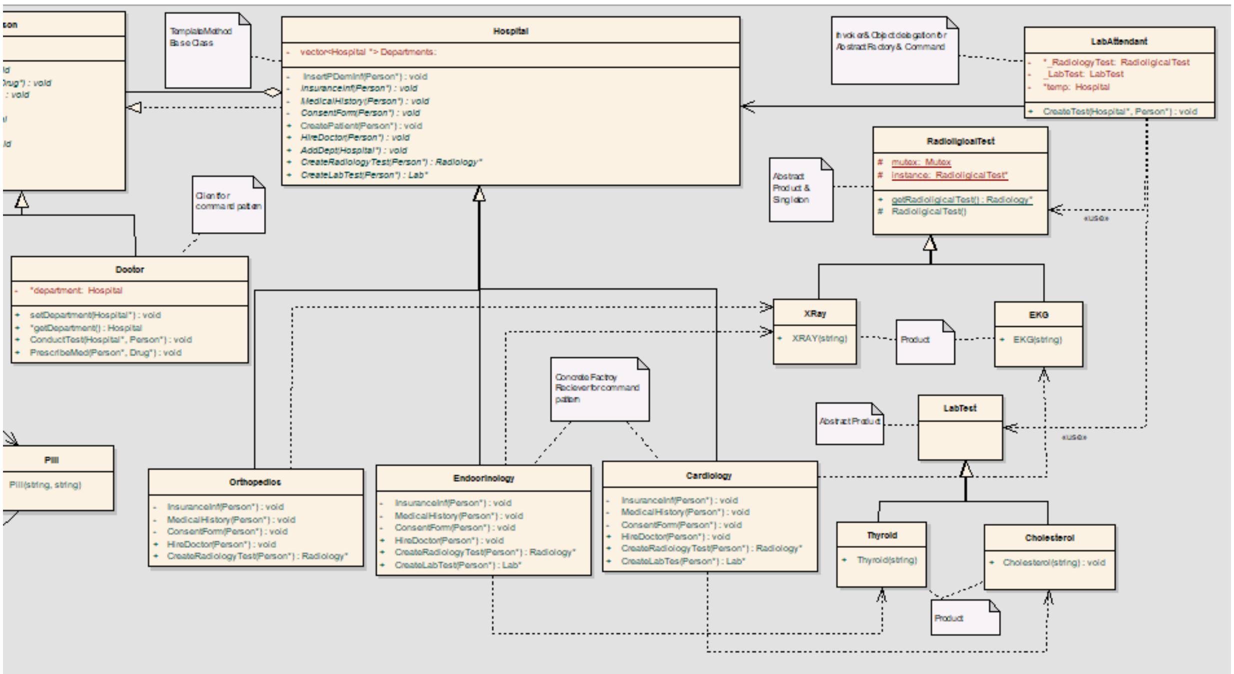
Concrete Product: XRAY - EKG & Thyroid - Cholesterol

XRAY and EKG are connected with IS-A relation to RadiologicalTest and they just have their constructors. Thyroid and Cholesterol are connected with IS-A relation to LabTest and they also just have their constructors.

Client class: LabTestAttendant

Its mission is to help object creation delegation with factory methods inside of its CreateTest(Hospital* dep,Person* patient) function.

II. Command Pattern


son
Drug*) : void
: void
TemplateMethod Base Class
Hospital
- vector<Hospital *> Departments:
- InsertPDemInf(Person*) : void
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ CreatePatient(Person*) : void
+ HireDoctor(Person*) : void
+ AddDept(Hospital*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTest(Person*) : Lab*
Invoker& Object delegation for Abstract Factory & Command
LabAttendant
- *_RadiologyTest: RadioligicalTest
- _LabTest: LabTest
- *temp: Hospital
+ CreateTest(Hospital*, Person*) : void
Client for command pattern
Doctor
- *department: Hospital
+ setDepartment(Hospital*) : void
+ *getDepartment() : Hospital
+ ConductTest(Hospital*, Person*) : void
+ PrescribeMed(Person*, Drug*) : void
Abstract Product & Singleton
RadioligicalTest
# mutex: Mutex
# instance: RadioligicalTest*
+ getRadioligicalTest() : Radiology*
# RadioligicalTest()
«use»
XRay
+ XRAY(string)
Product
EKG
+ EKG(string)
Concrete Factory Reciever for command pattern
AbstractProduct
LabTest
«use»
Pill
Pill(string, string)
Orthopedics
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
Endocrinology
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTest(Person*) : Lab*
Cardiology
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTes(Person*) : Lab*
Thyroid
+ Thyroid(string)
Cholesterol
+ Cholesterol(string) : void
Product

- PARTICIPANS –

Command: RadiologicalTest & LabTest

Concrete Command: X-RAY - EKG & Thyroid - Cholesterol

Client: Doctor

Invoker: LabAttendant

Receiver: Endocrinology & Cardiology & Orthopedic

Command: RadiologicalTest & LabTest

-> Here, in this scenario, we could have added a function as TestResult() or something like this in RadiologicalTest and LabTest. However, we thought it is already a command to order of creation of the test.

Concrete Command: X-RAY - EKG & Thyroid - Cholesterol

-> We considered creation of these classes are the commands, so their constructor will function as concrete commands.

Client: Doctor

-> Doctor orders the test and let the LabAttendant performs it job.

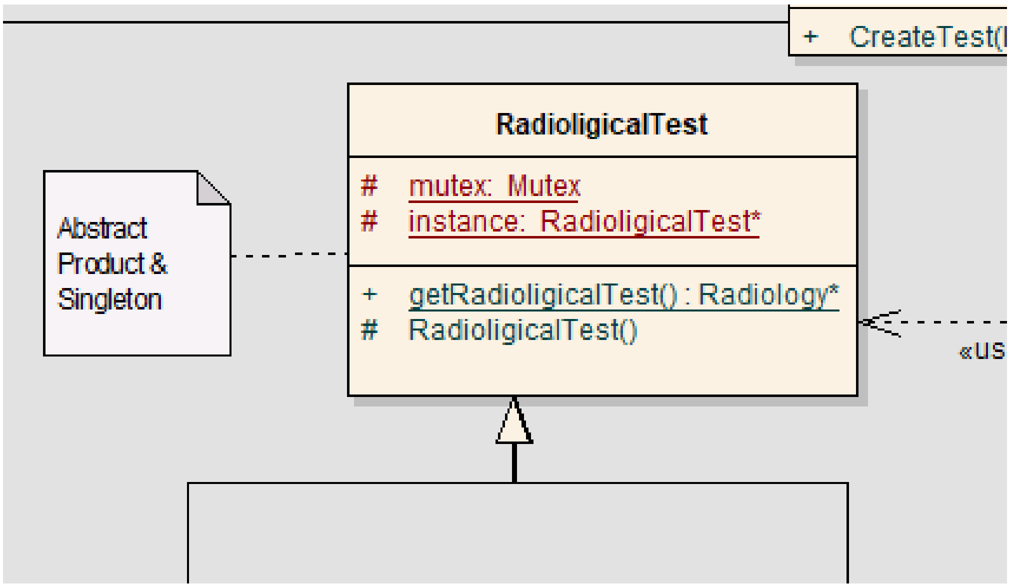
Invoker: LabAttendant

-> LabAttendant has CreateTest(Hospital* department,Person* patient). This function helps to create tests corresponding the doctor's department. By giving a Hospital* as a parameter, we are able to create tests dynamically.

Receiver: Endocrinology & Cardiology & Orthopedic

->These three classes are the receiver of the commands. The object creation is done here. Every department here has a different portion of the human the test. For example, Cardiology department only creates EKG and Cholesterol tests since these two tests are Cardiac system related. Another example would be here orthopedic. When we break a bone of ours, we are usually asked to get a XRAY to see if that bone is broken or not. More examples can be generated here.

III. Singleton Pattern


+ CreateTest(
RadioligicalTest
# mutex: Mutex
# instance: RadioligicalTest*
+ getRadioligicalTest() : Radiology*
# RadioligicalTest()
Abstract Product & Singleton
«us

- PARTICIPANS –

Singleton: RadiologicalTest

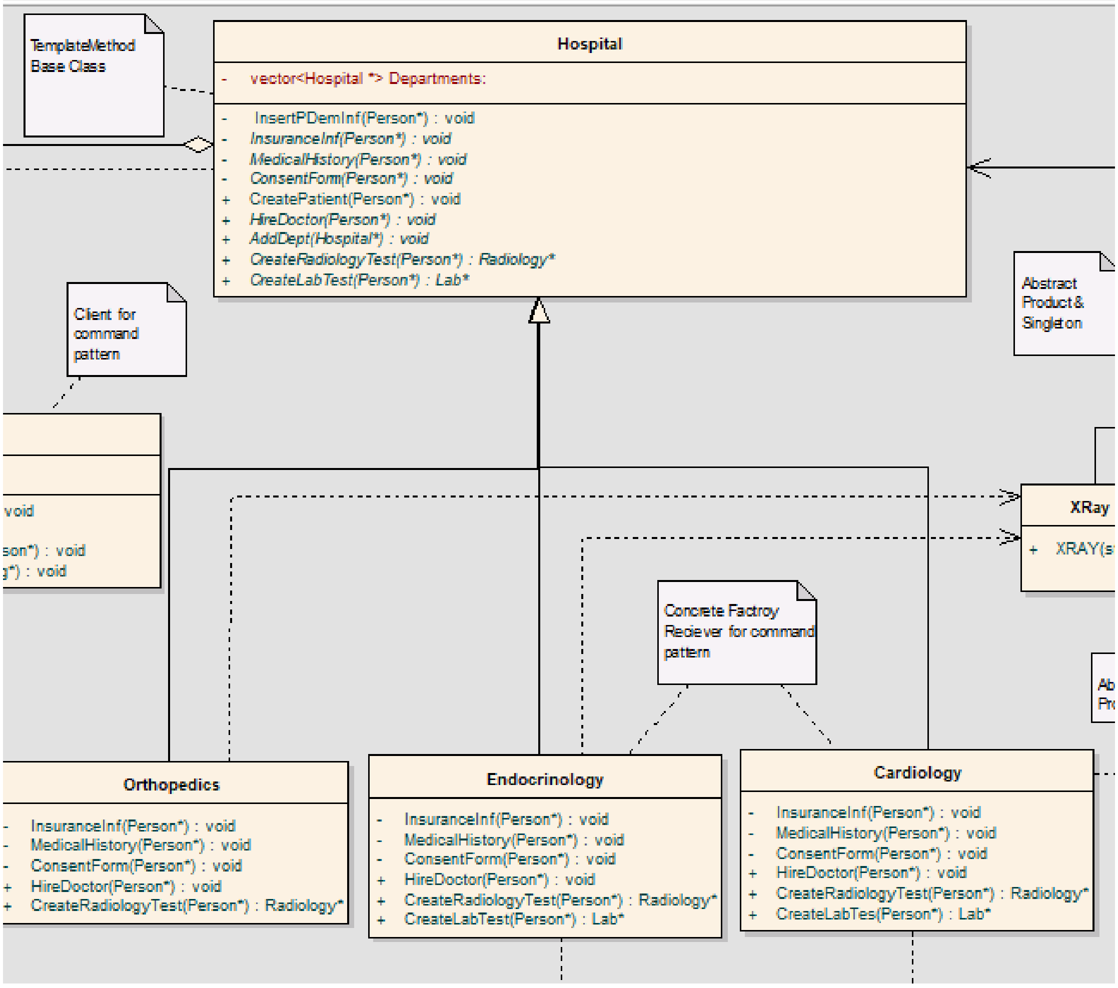
Mutex: We couldn't include the Mutex and Thread libraries into our code. We constantly got lots of errors about compiler. In order to have a well-functioning code we didn't include those two libraries. So, they are just dummy classes here. They are only here to say it could have been also "thread safe" as well.

Singleton class: RadiologicalTest

If we want to construct a singleton class we need first a private or protected constructor. So, the RadiologicalTest class's constructor is protected**.**

In this function the if statements are very important because they are creating only one instance. (Since the scenario says there should be only one Radiology Department we made this class Singleton)

IV. Template Pattern


TemplateMethod Base Class
Hospital
- vector<Hospital *> Departments:
- InsertPDemInf(Person*) : void
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ CreatePatient(Person*) : void
+ HireDoctor(Person*) : void
+ AddDept(Hospital*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTest(Person*) : Lab*
Abstract Product & Singleton
Client for command pattern
void
son*) : void
g*) : void
XRay
+ XRAY(s
Concrete Factroy Reciever for command pattern
Ab
Pr
Orthopedics
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
Endocrinology
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTest(Person*) : Lab*
Cardiology
- InsuranceInf(Person*) : void
- MedicalHistory(Person*) : void
- ConsentForm(Person*) : void
+ HireDoctor(Person*) : void
+ CreateRadiologyTest(Person*) : Radiology*
+ CreateLabTes(Person*) : Lab*

- PARTICIPANS -

Abstract Class: Hospital

Concrete Class: Orthopedics & Endocrinology & Cardiology

Abstract Class: Hospital: Hospital class is the base class of the Department. In the process of creating & admitting a patient to Hospital, patients are required to give the following pieces of information.

1- Demographic Information, (same for every department)
2- Insurance Information, (will change)
3- Past medical history, (will change)
4- Consent Form, (will change)

Except the demographic information of the patient, other three is changing from department to department. We have a member function CreatePatient which takes these 4 functions in it and performs the same order when it’s called (1,2,3,4).

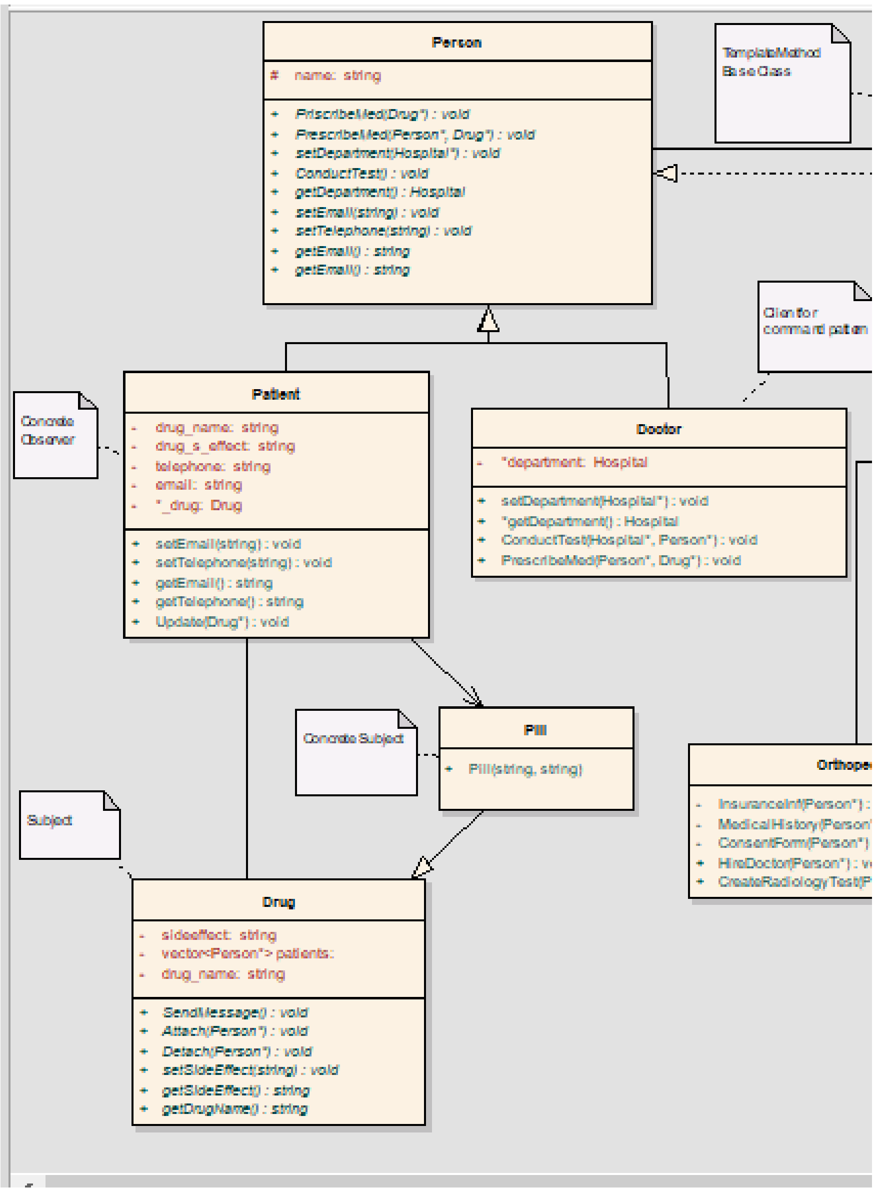
The major function for Template method here is CreatePatient(Person* patient) function here.

Concrete Class: Orthopedics & Endocrinology & Cardiology

-> In the concrete classes, the CreatePatient(Person* patient) already in the classes since these classes inherited from the Hospital. The only thing that we must do is override the member functions that varies from department to department. Varying information are (2,3,4).

P.S: We have not stored the insurance information, past medical history or consent form since it had no use for the future of the scenario here. The only purpose for us to use this was to show that we can use it without a problem.

V. Observer Pattern


Person
# name: string
+ PriscribeMed(Drug*) : void
+ PrescribeMed(Person*, Drug*) : void
+ setDepartment(Hospital*) : void
+ ConductTest() : void
+ getDepartment() : Hospital
+ setEmail(string) : void
+ setTelephone(string) : void
+ getEmail() : string
+ getEmail() : string
TemplateMethod BaseClass
Client for command pattern
Concrete Observer
Patient
- drug_name: string
- drug_s_effect: string
- telephone: string
- email: string
- *_drug: Drug
+ setEmail(string) : void
+ setTelephone(string) : void
+ getEmail() : string
+ getTelephone() : string
+ Update(Drug*) : void
Doctor
- *department: Hospital
+ setDepartment(Hospital*) : void
+ *getDepartment() : Hospital
+ ConductTest(Hospital*, Person*) : void
+ PrescribeMed(Person*, Drug*) : void
Concrete Subject
Pill
+ Pill(string, string)
Subject
Orthope
- InsuranceInf(Person*) :
- MedicalHistory(Person
- ConsentForm(Person*)
+ HireDoctor(Person*) : v
+ CreateRadiologyTest(P
Drug
- sideeffect: string
- vector<Person*> patients:
- drug_name: string
+ SendMessage() : void
+ Attach(Person*) : void
+ Detach(Person*) : void
+ setSideEffect(string) : void
+ getSideEffect() : string
+ getDrugName() : string

- PARTICIPANS –

Subject: Drug

Observer: Person

Concrete Subject: Pill

Concrete Observer: Patient

Subject: Drug

-> This is the interface for the subject. Drug class provides the following;

-Attaching and detaching observers to Subject(Drug&Pill). In order to attach an observer to a subject we use PrescribeMed(Person *p) function in Doctor class. We believe it makes the scenario more realistic.

-A notify function, in our case notify function is SendMessage() function.

-We use this function in setSideEffect(string seffect) after side effect(state) update.

Observer: Person

-> This is the interface for Observer. As you can imagine from the name of it, Person object observes the Subject (here it is Drug object.) This interface has the following important method in it;

-Update (Drug* drug): Function will work after a side effect(state) change in the subject. It will update Observer object's current state (side effect) to the incoming state.

Concrete Subject: Pill

-> Derived object from Drug. It has same member functions in it. Will be used as a Concrete Subject in our scenario.

Concrete Observer: Patient

-> Derived object from Person. It has also the same member functions and it will perform as a Concrete Observer.

Conclusion

In Hospital Management System, we tried to implement 5 patterns at once. First, we draw the UML and the hardest part was connection of 5 patterns in UML, we used Enterprise Architect to draw class diagram, then by adapting to structures, we started the implementation. We aimed to show that we understand design pattern.