Clinic Management System
Analysis and Design Document

Student:Grecu Razvan-Stefan
Group:30232

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

Use Swing/C# API to design and implement a client-server application for managing the consultations of doctors in a clinic. The application has three types of users: the clinic secretary, the doctors and an administrator.

- The clinic secretary can perform the following operations:
  - Add/update patients (patient information: name, identity card number, personal numerical code, date of birth, address).
  - CRUD on patients' consultations (e.g. scheduling a consultation, assigning a doctor to a patient based on the doctor's availability).
- The doctors can perform the following operations:
  - Add/view the details of a patient's (past) consultation.
- The administrator can perform the following operations:
  - CRUD on user accounts.

In addition, when a patient having a consultation has arrived at the clinic and checked in at the secretary desk, the application should inform the associated doctor by displaying a message.

The application should be client-server and the data will be stored in a database. Use the Observer design pattern for notifying the doctors when their patients have arrived.

1.2 Functional Requirements

This application is accessible via a valid combination of a username and a password. As such, it is meant for three types of users: secretaries, doctors and administrators. Each should have different privileges, fullfilling the following criteria:

- Secretaries can:
  - CRUD on patients
  - schedule consultations, update consultations, delete consultations
- Doctors can:
  - view consultations, update consultations
  - view notifications of schedules
- Administrators can:
  - CRUD on users

1.3 Non-functional Requirements

Non-functional requirements impose constraints on the services and functionalities of the system; such constraints could be : temporal constraints, development constraints, imposed standards etc. Depending on the situation, non-functional requirements may be more critical than functional ones. If these requirements are not met, the system will not be useful for the purpose it was developed. The conventional I/O mechanism for the application must have:

- Java compliant device, to run the application
- Basic I/O devices, such as a keyboard and/or mouse, as the user must offer data as input, and a monitor as he expects an outcome.

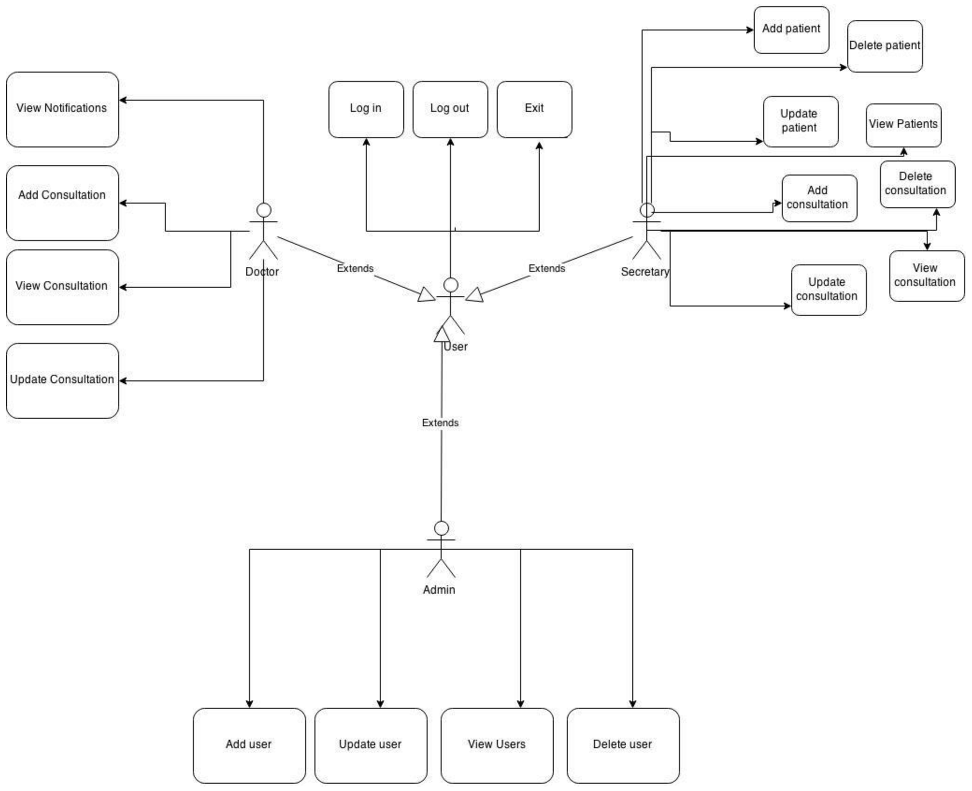
The availability of an application is tightly coupled to its fiability. If an application is not readily-available when it is needed, it is unlikely that it fulfills the role for which it was created.

2. Use-Case Model

Use case: Add a user
Level: User
Primary actor: Administrator
Main success scenario: to insert a new user into the system, one first has to be logged in as an administrator in the first place, to have such privileges. You will have to select the administrator menu and click on the insert user item. Redirected at the right page, you will have to enter the username, password and role of the user and click submit.
Extensions: A case of failure would be to introduce a wrong type of user, other than those specified in the user type enumeration.

Use case: View notifications
Level: User
Primary actor: Doctor
Main success scenario: You need to log in as a doctor, and a secretary, beforehand has added a consultation to the doctor that has signed in. He will then use the doctor menu and click on the notifications item and see them.

Use case: Update patient
Level: User
Primary actor: Secretary
Main success scenario: You need to be logged in as a Secretary; click on the secretary menu and update patient item and change a specified patient's (specified by its cnp) information. If all the data is alright, clicking submit will update the patient.
Extensions: You enter wrong data, maybe a wrong date format.



3. System Architectural Design

3.1 Architectural Pattern Description

An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. You can think of architecture styles and patterns as sets of principles that shape an application. Garlan and Shaw define an architectural style as "a family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition."

The client/server architectural style describes distributed systems that involve a separate client and server system, and a connecting network. The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style.

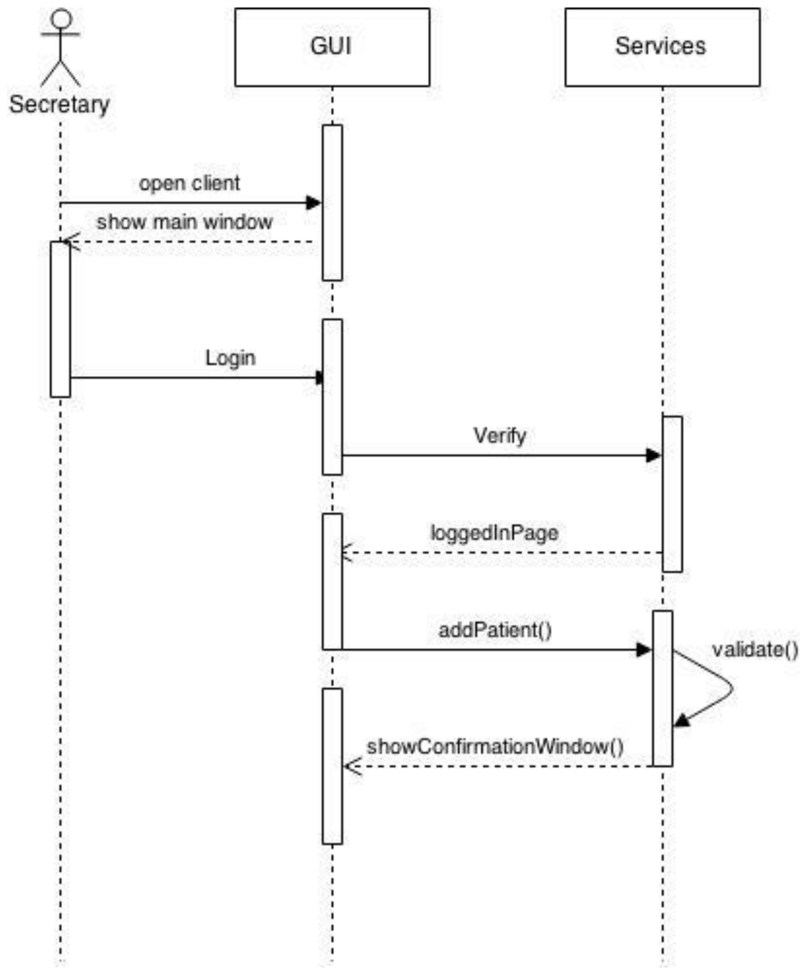
Historically, client/server architecture indicated a graphical desktop UI application that communicated with a database server containing much of the business logic in the form of stored procedures, or with a dedicated file server. More generally, however, the client/server architectural style describes the relationship between a client and one or more servers, where the client initiates one or more requests (perhaps using a graphical UI), waits for replies, and processes the replies on receipt. The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client. To achieve this architecture, I have used JAX-WS to create the server (namely endpoints - interfaces and their implementations, and a publisher), whereas in the client, I just needed to call the right URL to get the information needed.

Java API for XML Web Services (JAX-WS) is one of a set of Java technologies used to develop Web services. JAX-WS belongs to what Sun Microsystems calls the "core Web services" group. Like most of the core group, JAX-WS is typically used in conjunction with other technologies. Those other technologies may also come from the core Web services group (JAXB, for example), as well as from enhanced Web services (WSIT), secure Web services (WSIT, WS-Security), legacy Web services (JAX-RPC), and systems management services (WS-Management) groups.

JAX-WS is a fundamental technology for developing SOAP (Simple Object Access Protocol) and RESTful (Web services that use representational state transfer, or REST, tools) Java Web services, where JAX-WS is designed to take the place of the JAVA-RPC (Remote Procedure Call) interface in Web services and Web-based applications. JAX-WS is also used to build Web services and corresponding clients that communicate using XML to send messages or use remote procedure calls to exchange data between client and service provider.

JAX-WS represents remote procedure calls or messages using XML-based protocols such as SOAP, but hides SOAP's innate complexity behind a Java-based API. Developers use this API to define methods, then code one or more classes to implement those methods and leave the communication details to the underlying JAX-WS API. Clients create a local proxy to represent a service, then invoke methods on the proxy. The JAX-WS runtime system converts API calls and matching replies to and from SOAP messages.

4. UML Sequence Diagrams

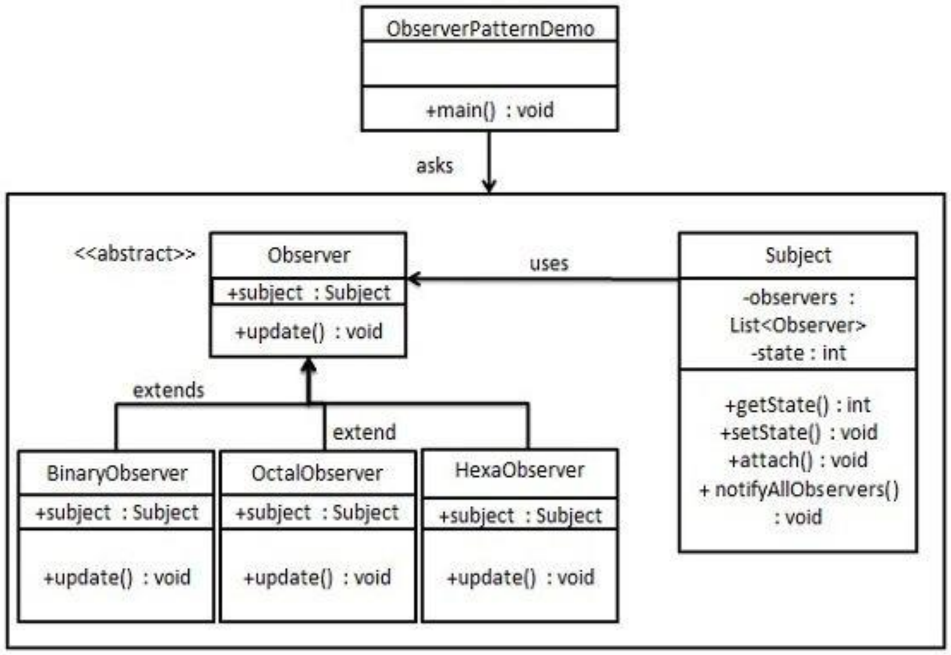


5. Class Design

5.1 Design Patterns Description

Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object. Bellow is an example of implementation, to better understand the pattern.

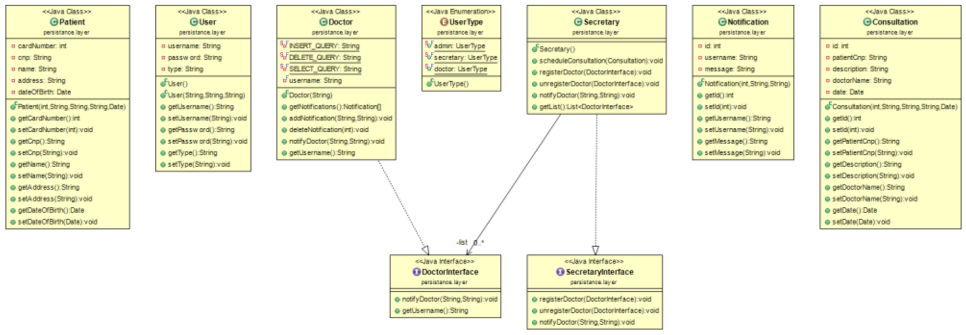


5.2 UML Class Diagram

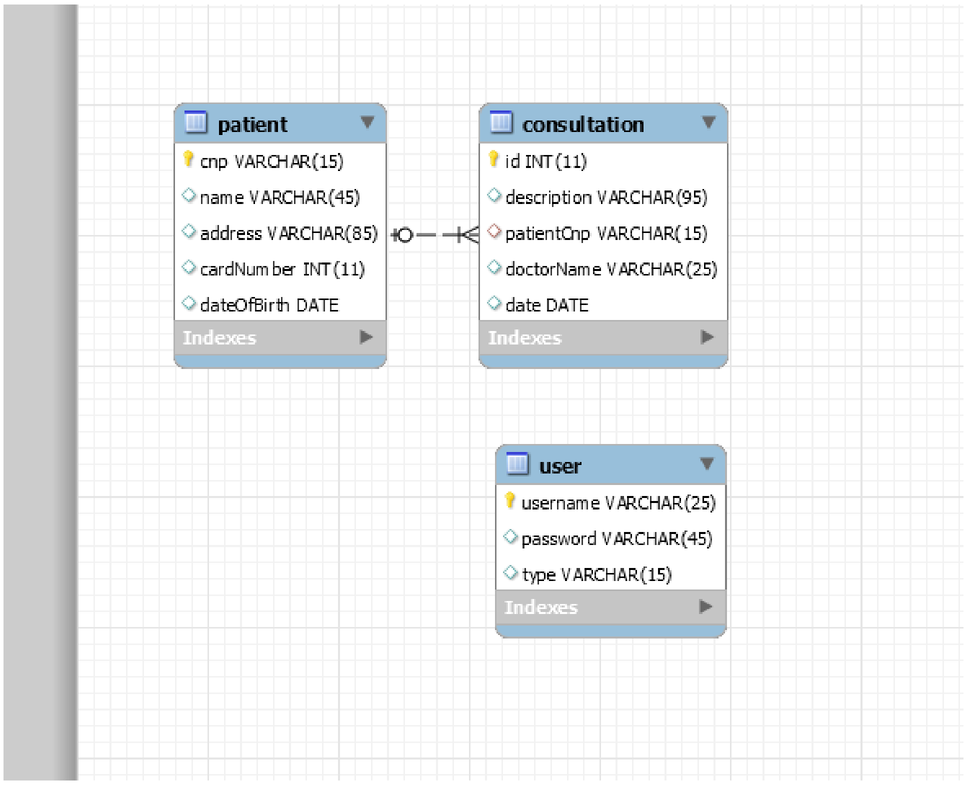
As the image is rather big, I have decided to add the diagram as a separate file.

6. Data Model

Furthermore I shall describe the models :



And in the database :



7. Bibliography

Martin Fowler - Patterns of Enterprise Application Architecture
Microsoft Library - Client-Server Architecture
http://java.sun.com/docs/books/tutorial/uiswing/
http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html
http://www.exampledepot.com/egs/?
http://download.oracle.com/javase/tutorial/networking/sockets/index.html
http://msdn.microsoft.com/en-us/library/b6xa24z5.aspx
http://msdn.microsoft.com/en-us/library/54xbah2z(VS.80).aspx
http://msdn.microsoft.com/en-us/library/e80y5yhx(VS.80).aspx