3

Student: Czako Zoltan

**Group: 30235**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 4

3. System Architectural Design 6

4. UML Sequence Diagrams 8

5. Class Design 10

6. Data Model 13

7. System Testing 13

8. Bibliography 14

1. Requirements Analysis

# Assignment Specification

In this assignment we implemented a client-server application. The subject of this application is a medical app, which helps the employees in their daily work, helps them to manipulate the patient’s data, to make appointments between doctor and patient and so on. Besides helping the employees, it also helps the administrator of the hospital, who can add new users, view users and he can modify user’s data. This app will notify all the doctors who has an appointment with a patient, if the patient has arrived.

# Functional Requirements

Our application will have three types of users. The first type is the Secretary. An secretary has access to the data of the patiens in the hospital. He can insert new patient data, he can modify the information of an existing patient and if he must, he can delete a the information of a patient. A secretary can see all the data of a patient, he can view the name of a patient, the mobile, the mail etc.

The second type of user is the Administrator. This type of user has the ability to view the list of the hospital’s app users, he can select a user and view his personal information like username, first name, last name, mobile and mail. He can modify the user data of a user and he can also delete a user if needed.

The third type of user is the Doctor. The doctor can view a patient’s data, he can view the patient’s treatment history and he can create a new treatment.

The app is also responsible for notifying all the doctors that has an appointment with a patient if the patient has arrived.

# Non-functional Requirements

In order to obtain all this functionality we created an ASP.NET WebApi project as a server for this app and as the client we will have Html, CSS and the Http requests will be made using Ajax calls. Talking about accessibility, to differentiate the two type of user we created a login page. After login there are two different views, one for the admin, one for the doctor and one for the secretary. We used list to show all the users and all the treatments in an ordered way. For creating and updating users/appointments data we created another two views with text boxes in which we can introduce or modify data.

To obtain availability we created a GitHub repository and after each modification that we made or new features that we created, we committed out changes to this repository and in this way we prevented data loss, wrong modifications or accidental code deletion. This repository helps us in backup, because we can backup our system striating with a well specified date(last time when our system was stabile).

Talking about dependency, on the server side we have a layered architecture, we have three separate project one for the Database, one for the Services and one for the WeApi. Besides these functional layers we have another project for testing our system.

We work really much on failure management and we tried to cover all the situations when a user can enter wrong data, or tries to make some illegal operation.

Talking about portability, our application is platform dependent, it work only on Windows operation systems because it is a Windows native application.

In the view of usability, this software is intended to be used in hospitals, it can help in patient data management and for create appointments between doctors and patients.

To grant a well working and bugs free application, to keep business needs well satisfied we created Unit Tests for the Services, in which we have data validations or conditional statements. We tried to have a 100% of covering, we test corner cases as well as happy flow situations.

2. Use-Case Model

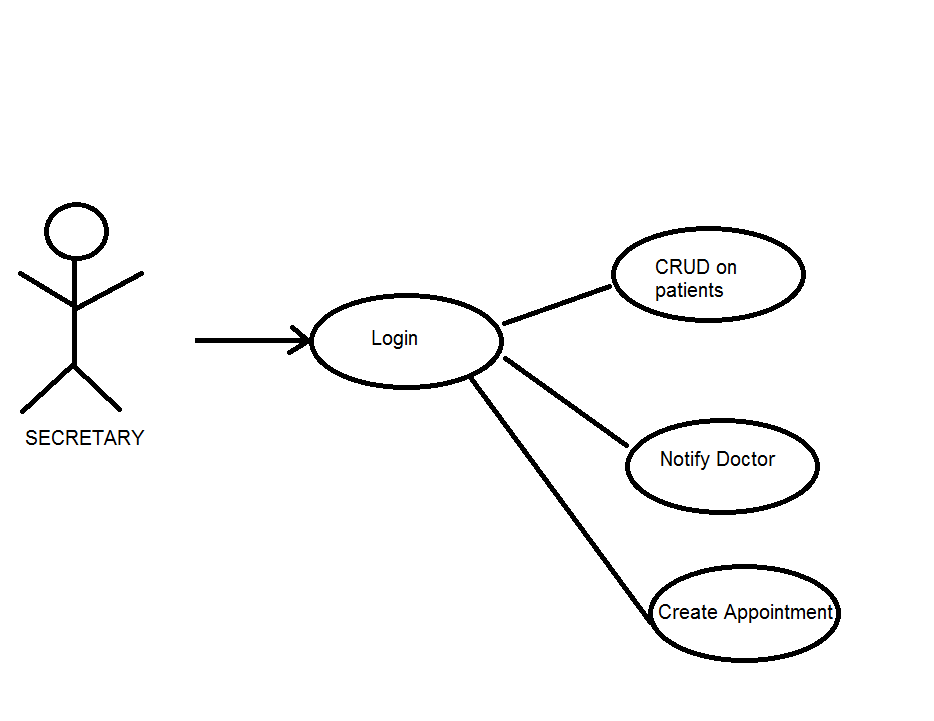
Use Case: Describe the possible actions of the secretary

Level: user-goal level

Primary actor: Secretary

Main success scenario:

1. Login
2. CRUD on patients data
3. Create appointment between patient and doctor



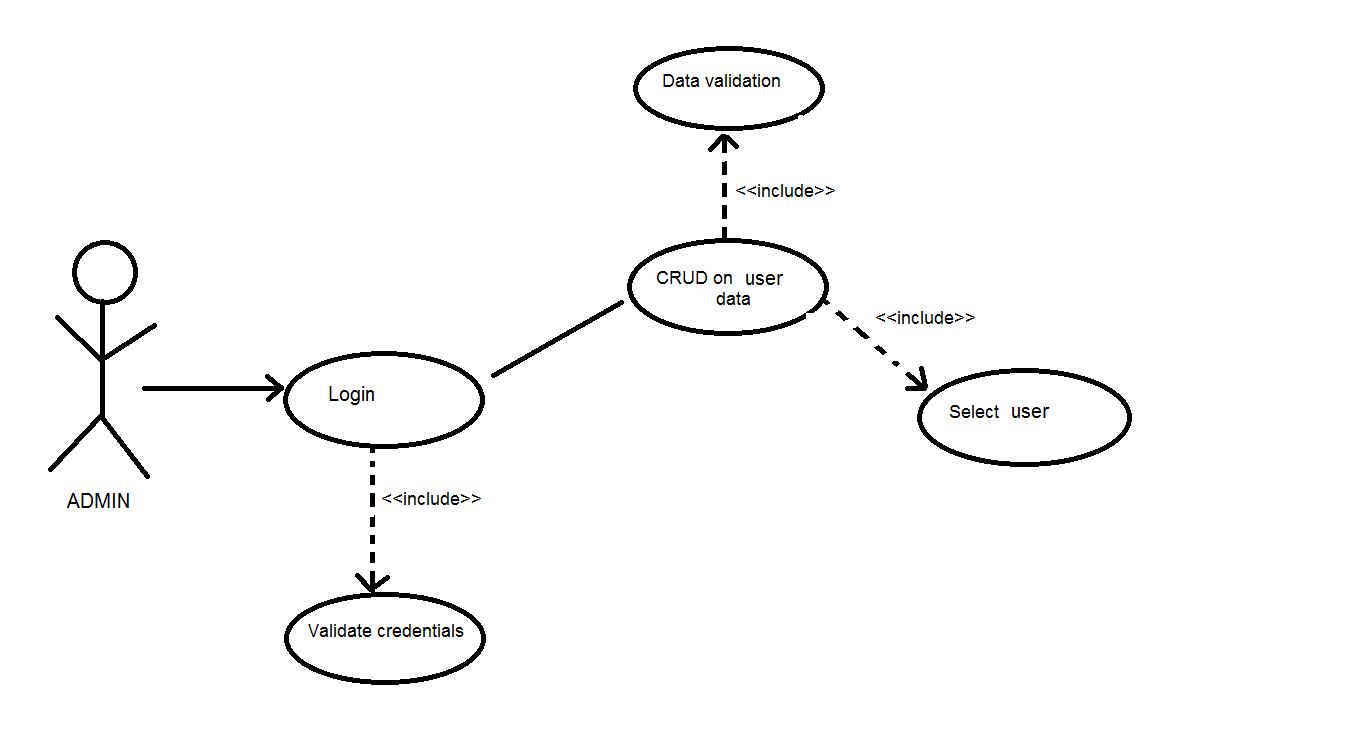
Use Case: Describe the possible actions of the aministrator

Level: sub-function level

Primary actor: Administrator

Main success scenario:

1. Login 🡪 Validate credentials
2. CRUD on Users 🡪 Select user 🡪 Validate data



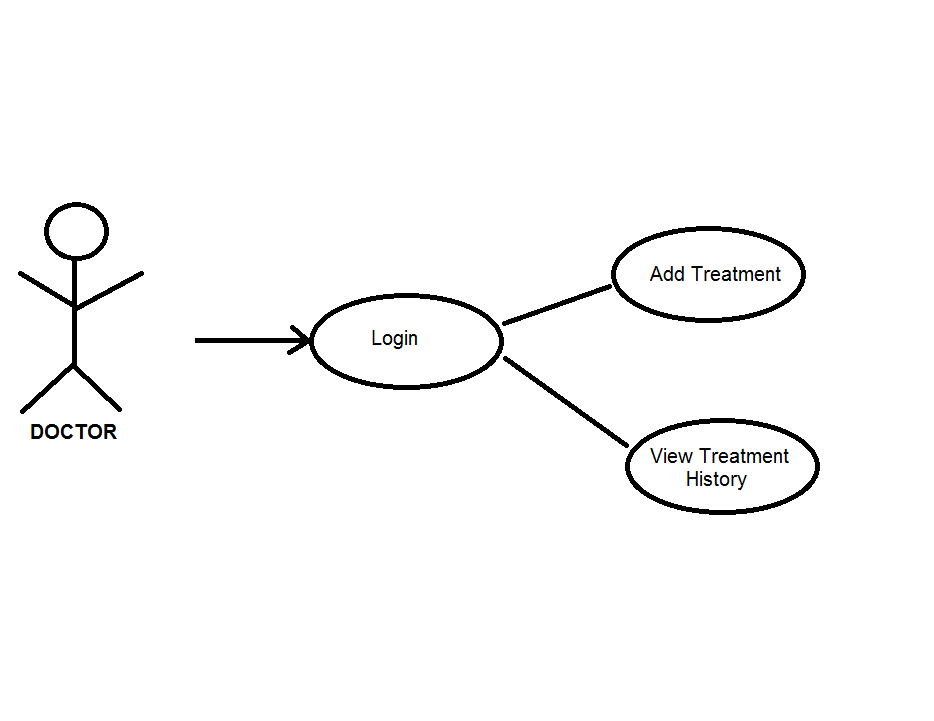
Use Case: Describe the possible actions of the doctors

Level: user-goal level

Primary actor: Doctor

Main success scenario:

1. Login
2. Create Treatment
3. View Treatment history for a patient



3. System Architectural Design

**3.1 Architectural Pattern Description**

In [Computer science](https://simple.wikipedia.org/wiki/Computer_science) client-server is a [software architecture](https://simple.wikipedia.org/wiki/Computer_architecture) model consisting of two parts, [client](https://simple.wikipedia.org/wiki/Client) systems and [server](https://simple.wikipedia.org/wiki/Server_(computing)) systems, both communicating over a [computer network](https://simple.wikipedia.org/wiki/Computer_network) or on the same [computer](https://simple.wikipedia.org/wiki/Computer). A client-server application is a [distributed system](https://simple.wikipedia.org/wiki/Distributed_computing) made up of both client and server software. Client server application provide a better way to share the workload.The client [process](https://simple.wikipedia.org/wiki/Process_(computing)) always initiates a connection to the server, while the server [process](https://simple.wikipedia.org/wiki/Process_(computing)) always waits for requests from any client. When both the client [process](https://simple.wikipedia.org/wiki/Process_(computing)) and server [process](https://simple.wikipedia.org/wiki/Process_(computing)) are running on the same [computer](https://simple.wikipedia.org/wiki/Computer), this is called a single seat setup.

Another type of related [software architecture](https://simple.wikipedia.org/w/index.php?title=Software_architecture&action=edit&redlink=1) is known as peer-to-peer, because each host or [application](https://simple.wikipedia.org/wiki/Process_(computing)) [instance](https://simple.wikipedia.org/wiki/Object_(computer_science)) can simultaneously act as both a client and a server (unlike centralized servers of the client-server model) and because each has equivalent responsibilities and status. Peer-to-peer architectures are often [abbreviated](https://simple.wikipedia.org/wiki/Abbreviation) using the [acronym](https://simple.wikipedia.org/wiki/Acronym) P2P.

The client-server relationship describes the relation between the client and how it makes a service request to the server, and how the server can accept these requests, process them, and return the requested information to the client. The interaction between client and server is often described using [sequence diagrams](https://simple.wikipedia.org/w/index.php?title=Sequence_diagram&action=edit&redlink=1). Sequence diagrams are standardized in the [Unified Modeling Language](https://simple.wikipedia.org/w/index.php?title=Unified_Modeling_Language&action=edit&redlink=1).

Both client-server and P2P architectures are in wide usage today.

The basic type of client-server [architecture](https://simple.wikipedia.org/w/index.php?title=Software_architecture&action=edit&redlink=1) employs only two types of hosts: [clients](https://simple.wikipedia.org/wiki/Client) and [servers](https://simple.wikipedia.org/wiki/Server). This type of architecture is sometimes referred to as two-tier. The two-tier architecture means that the [client](https://simple.wikipedia.org/wiki/Client) acts as one tier and[server](https://simple.wikipedia.org/wiki/Server) [process](https://simple.wikipedia.org/wiki/Process_(computing)) acts as the other tier.

The client-server [architecture](https://simple.wikipedia.org/w/index.php?title=Software_architecture&action=edit&redlink=1) has become one of the basic models of [network computing](https://simple.wikipedia.org/wiki/Network_computing). Many types of applications have been written using the client-server model. Standard networked functions such as [E-mail](https://simple.wikipedia.org/wiki/E-mail) exchange,[web](https://simple.wikipedia.org/wiki/World_Wide_Web) access and [database](https://simple.wikipedia.org/wiki/Database) access, are based on the client-server model. For example, a [web browser](https://simple.wikipedia.org/wiki/Web_browser) is a client program at the user computer that may access information at any web server in the world.

**Clients characteristics:**

* Always initiates requests to [servers](https://simple.wikipedia.org/wiki/Server).
* Waits for replies.
* Receives replies.
* Usually connects to a small number of [servers](https://simple.wikipedia.org/wiki/Server) at one time.
* Usually interacts directly with end-users using any [user interface](https://simple.wikipedia.org/wiki/User_interface) such as [graphical user interface](https://simple.wikipedia.org/wiki/Graphical_user_interface).

**Servers characteristics:**

* Always wait for a request from one of the clients.
* Serve [clients](https://simple.wikipedia.org/wiki/Client) requests then replies with requested data to the clients.
* A [server](https://simple.wikipedia.org/wiki/Server) may communicate with other servers in order to serve a client request.
* If additional information is required to process a request (or security is implemented), a server may request additional data (passwords) from a client before processing a request.
* End users typically do not interact directly with a server, but use a client.

## [Support for Test-Driven Development](javascript:void(0))

The server side is easy to test, because it is using .NET Technologies, it is a WebAPI written in C#. It can be easily tested using Unit Tests. For testing just a layer and not the whole application we can use the Mock Framework to create mock data and to simulate functionality of dependent resources.

**3.2 Diagrams**

**Package Diagram**

HTML

CSS

JavaScript

Ajax

**Client Side**

**Server Side**

OnlineClinic

WebAPI

ServiceTests

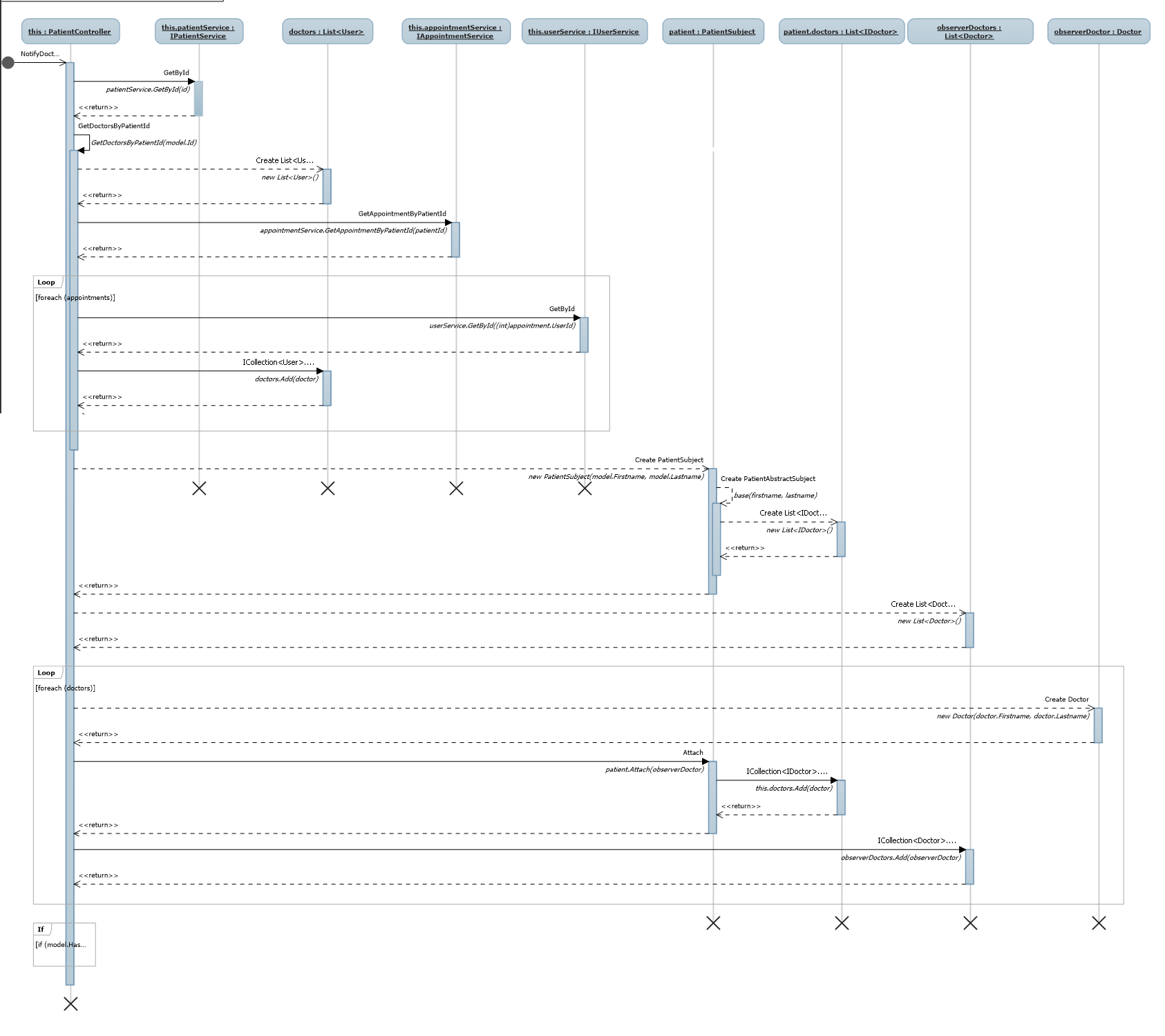
Services

Repository using Entity

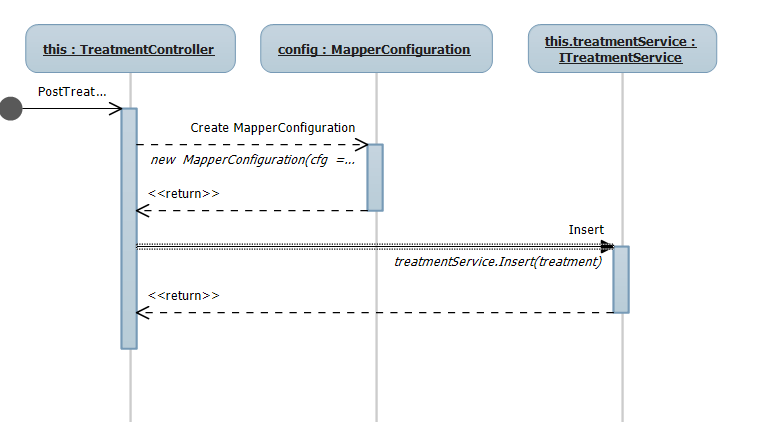
Database

4. UML Sequence Diagrams

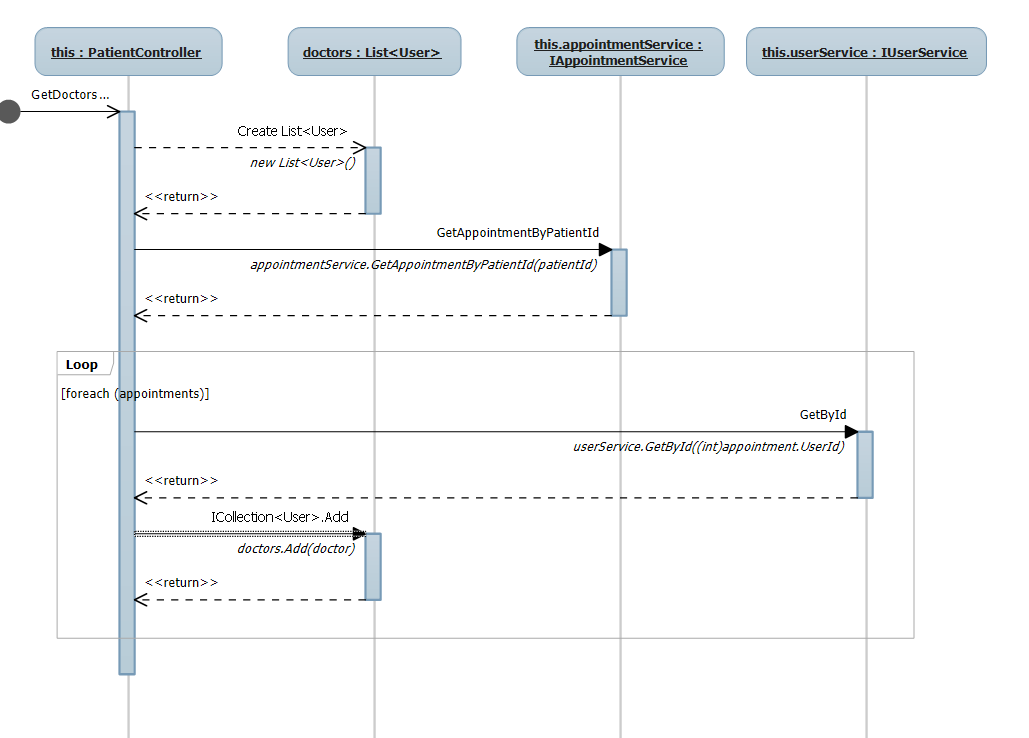
In the next image we will present a sequence diagram in which we describe the flow of how we notify a doctor about a patient using the Observer Design Pattern:



The next sequence diagram shows the way a doctor creates a treatment



The next sequence diagram shows how we can find for a patient with which doctor has appointment:



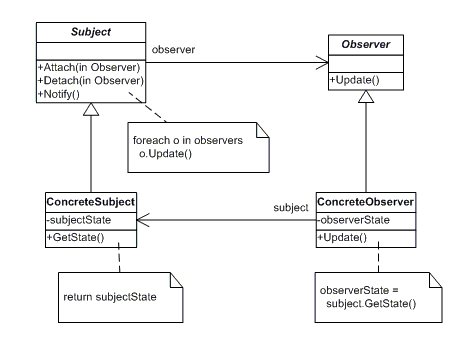
5. Class Design

**5.1 Design Patterns Description**

### **Participants**

    The classes and objects participating in this pattern are:

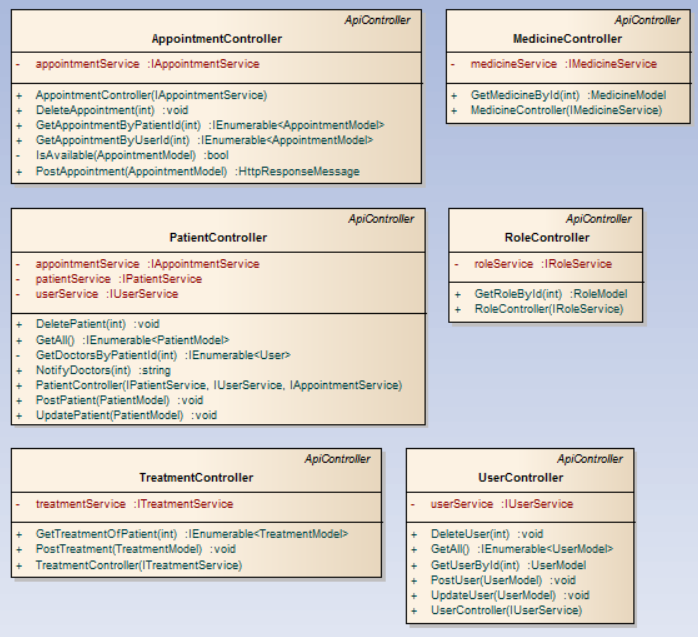
* **Subject**  **(Stock)**
  + knows its observers. Any number of Observer objects may observe a subject
  + provides an interface for attaching and detaching Observer objects.
* **ConcreteSubject**  **(IBM)**
  + stores state of interest to ConcreteObserver
  + sends a notification to its observers when its state changes
* **Observer**  **(IInvestor)**
  + defines an updating interface for objects that should be notified of changes in a subject.
* **ConcreteObserver**  **(Investor)**
  + maintains a reference to a ConcreteSubject object
  + stores state that should stay consistent with the subject's
  + implements the Observer updating interface to keep its state consistent with the subject's



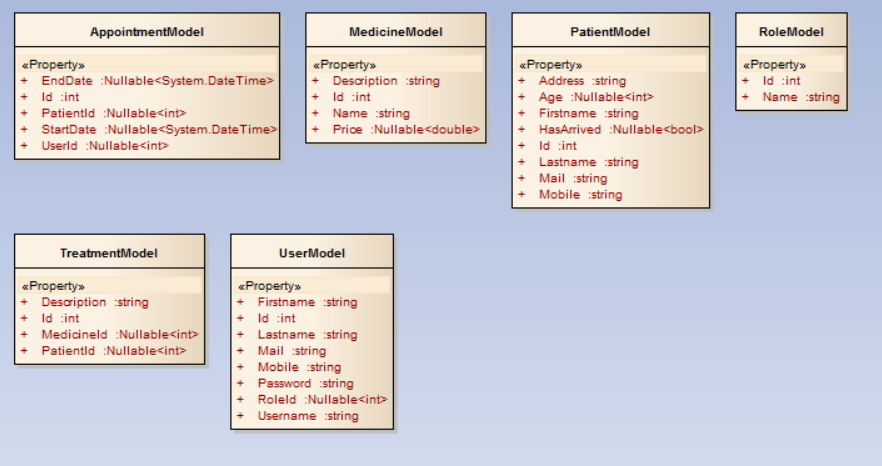
**5.2 UML Class Diagram**

**Class diagram for Bookstroe (MVC)**

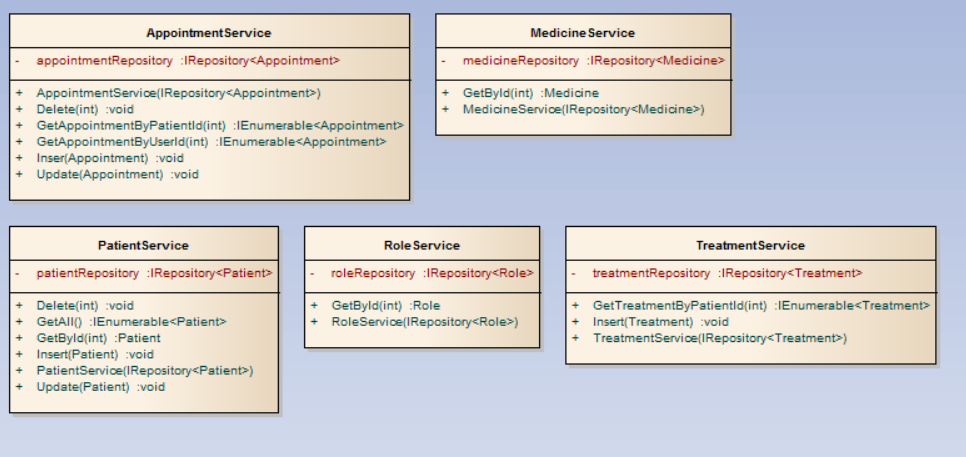
*Controllers*

**

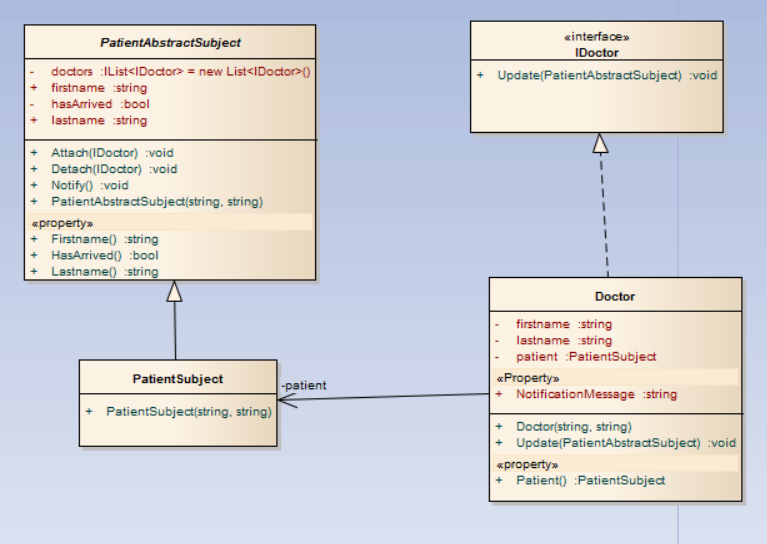
*Models*

**

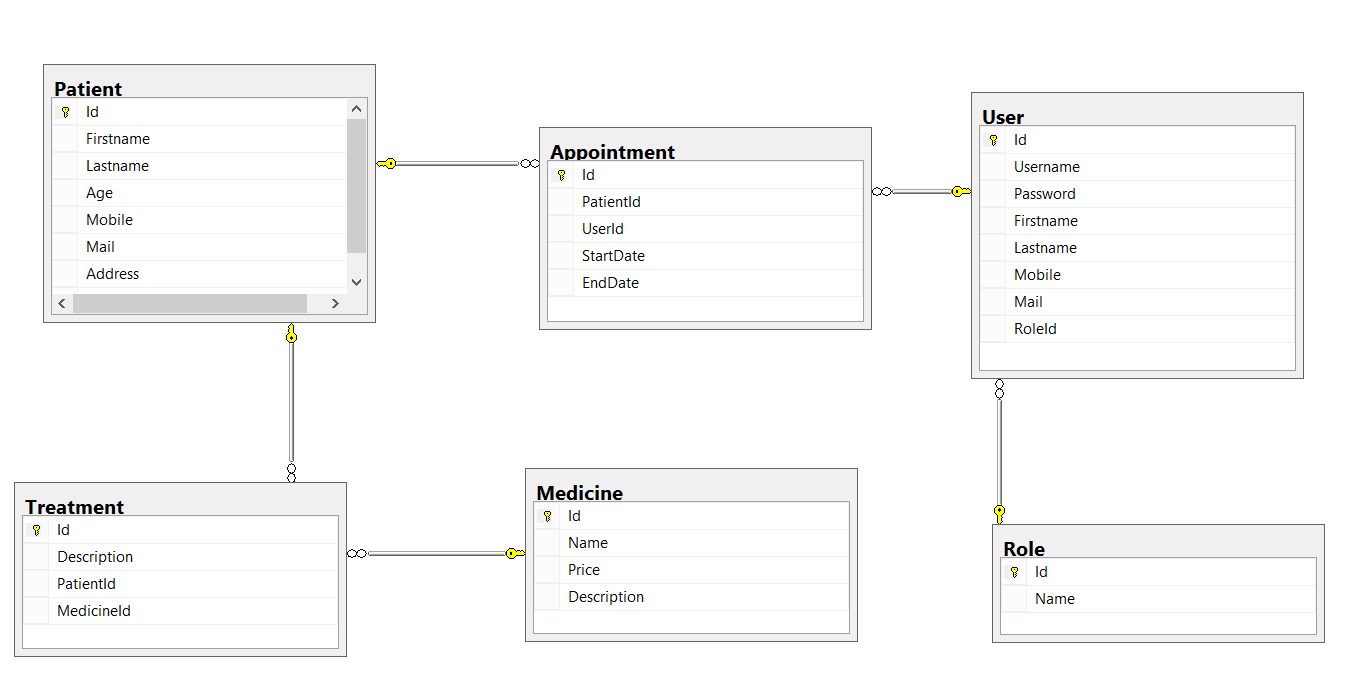
**Class diagram for Services**



**Class diagram for Observer Pattern**

****

6. Data Model



7. System Testing

We wanted to create a reliable and functionally correct project so we created test for each functionality, for each data validation and conditional statements. We used the Unit test framework and we tested the Services, because in this part of the project we had a lot of logic. We tested the corner cases as well as the happy flow situations.

8. Bibliography

[1] <http://www.dofactory.com/net/observer-design-pattern>

[2] <https://msdn.microsoft.com/en-us/library/ff506346(v=vs.110).aspx>

[3] <http://www.asp.net/web-api/overview/getting-started-with-aspnet-web-api/tutorial-your-first-web-api>

[4] <http://www.w3schools.com/ajax/>

[5] <https://developer.mozilla.org/en-US/Learn/CSS>

[6] <http://www.w3schools.com/html/>

[7] <https://simple.wikipedia.org/wiki/Client-server>