ASSIGNMENT A3
Analysis and Design Document

Student: Buzea Vlad-Calin
Group: 30432

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 secretary, the doctors and an administrator.

The secretary can perform the following operations:
- Add/update patients (patient information: name, 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.
- CRUD on patients.

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.

1.2 Functional Requirements

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.

1.3 Non-functional Requirements

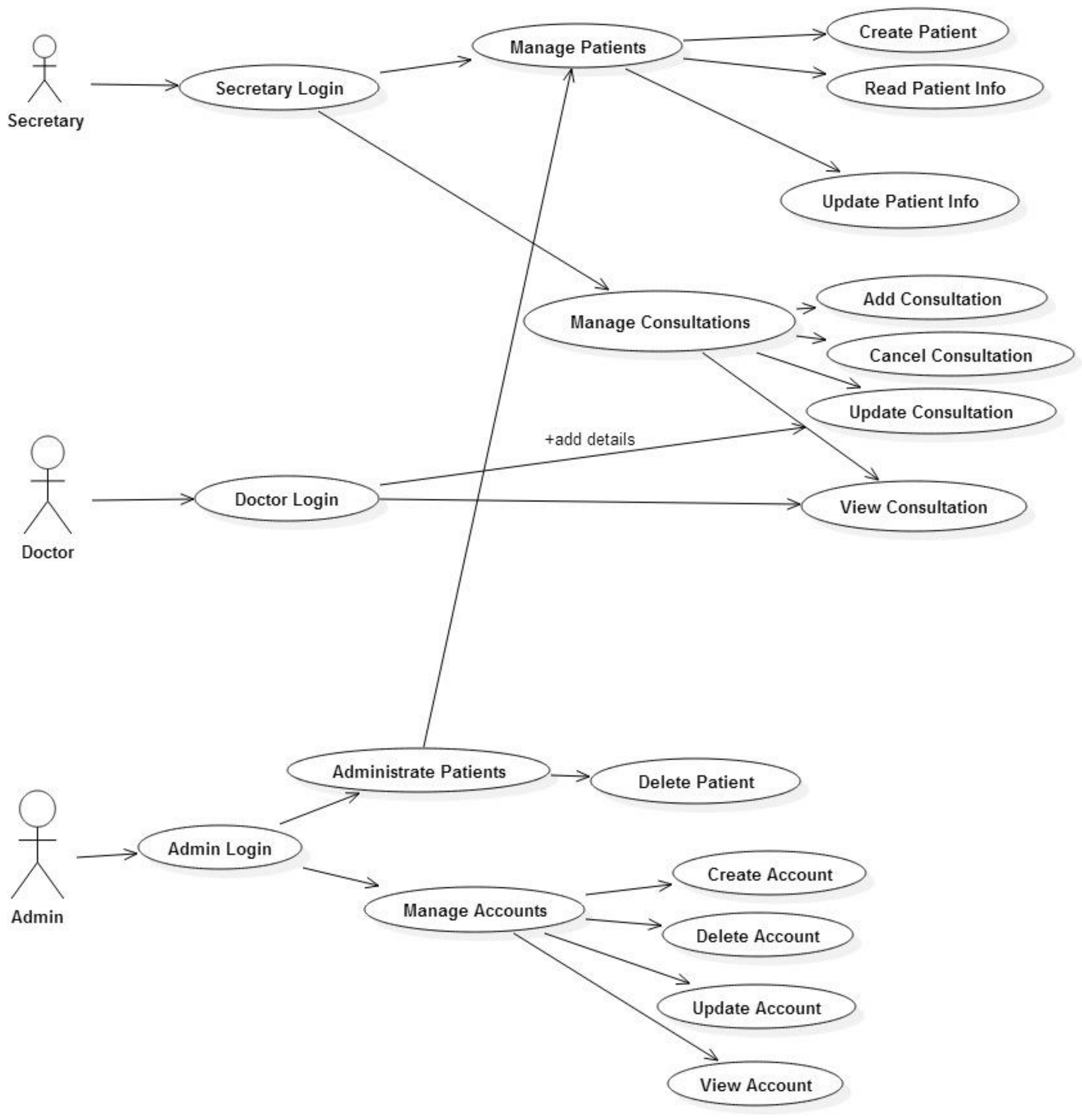
- The application should be accessible and easy to use.
- The application must be secure
- The user interface must update instantly (in under 1 second)
- The application must not jam

2. Use-Case Model

Use case: *Create new Consultation*
Level: *user-goal level*
Primary actor: *Secretary*
Main success scenario:

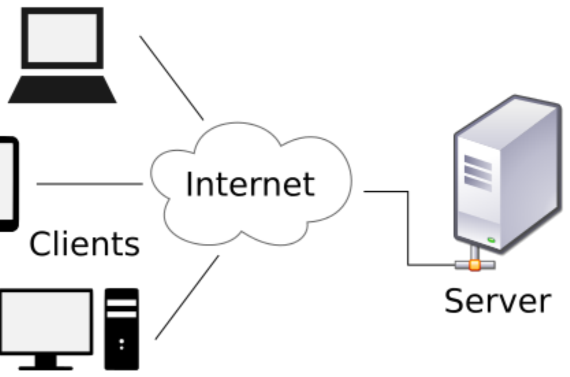
1. Open Hospital Application
2. Provide username and password
3. Press the "Login" button
4. Select the "Add Consult" Tab
5. Select the patient, the doctor and the time of the consultation
6. Press the "Add Consultation" button
7. The system confirms the creation of the consultation

Extensions: *If the time of the consultation does not fit the schedule, the secretary will be notified and the creation of the appointment does not take place.*


Secretary
Secretary Login
Manage Patients
Create Patient
Read Patient Info
Update Patient Info
Manage Consultations
Add Consultation
Cancel Consultation
Update Consultation
+add details
View Consultation
Doctor
Doctor Login
Admin
Admin Login
Administrate Patients
Delete Patient
Manage Accounts
Create Account
Delete Account
Update Account
View Account

3. System Architectural Design

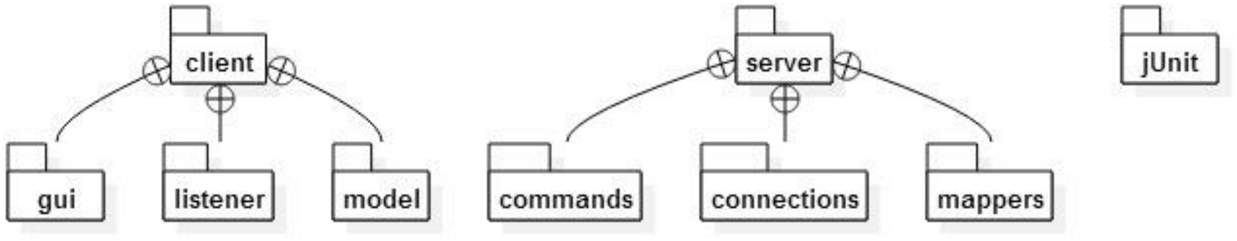
3.1 Architectural Pattern Description



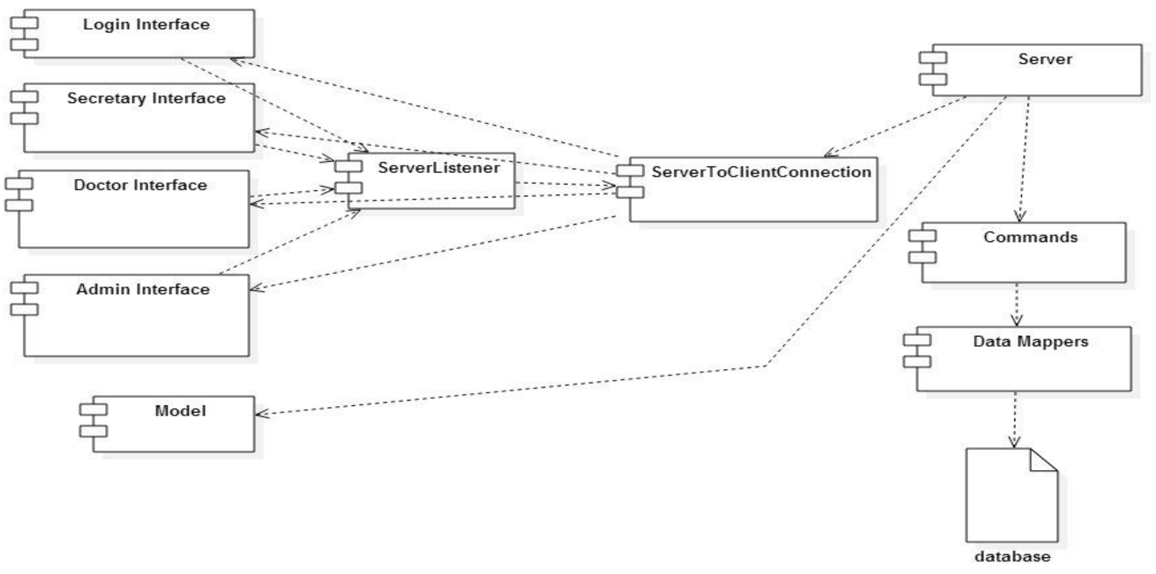
The **client–server** model of computing is a distributed computing structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs which share their resources with clients. A client does not share any of its resources, but requests a server's content or service function. Clients therefore initiate communication sessions with servers which await incoming requests. Examples of computer applications that use the client–server model are Email, network printing, and the World Wide Web.

3.2 Diagrams

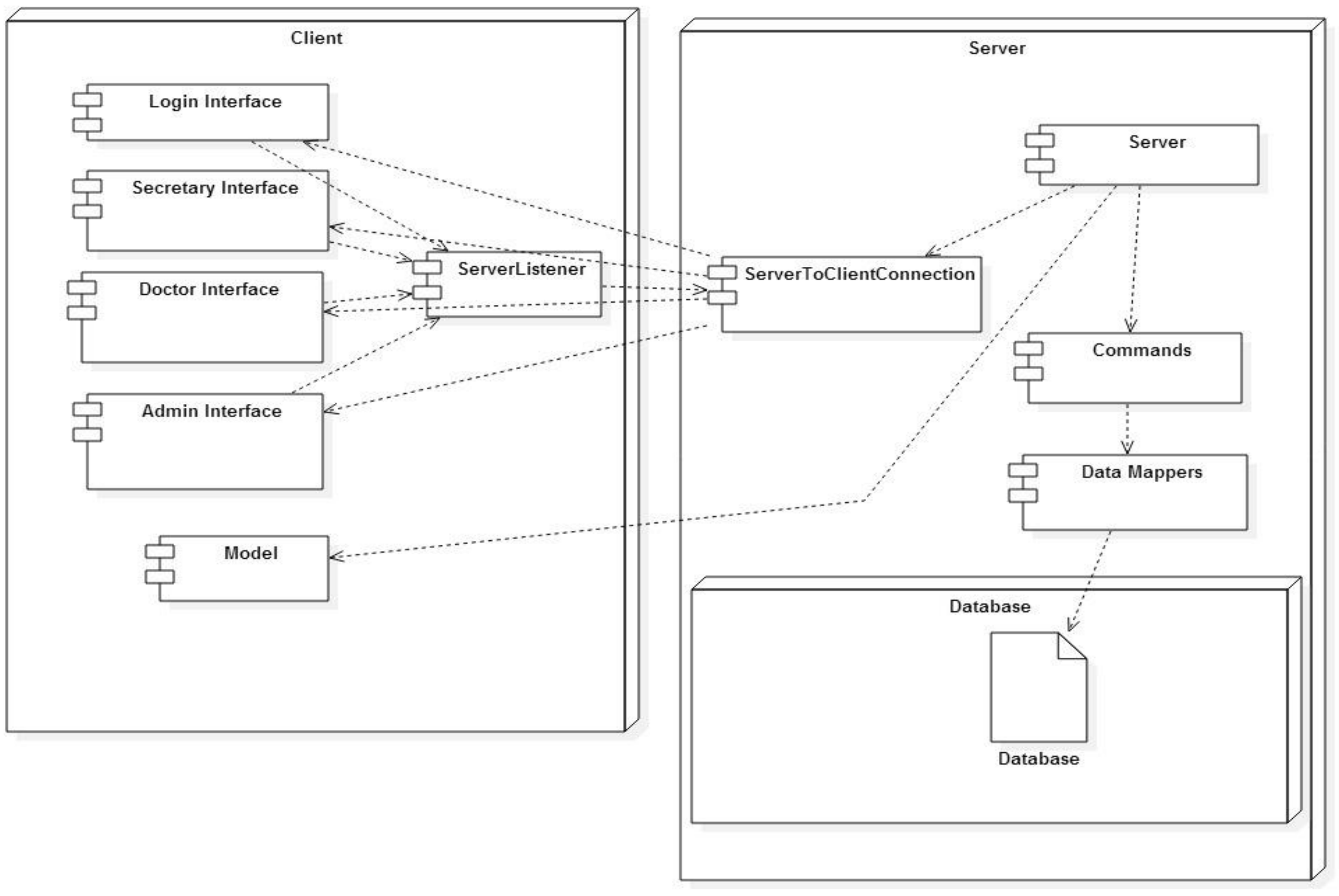
Package Diagram



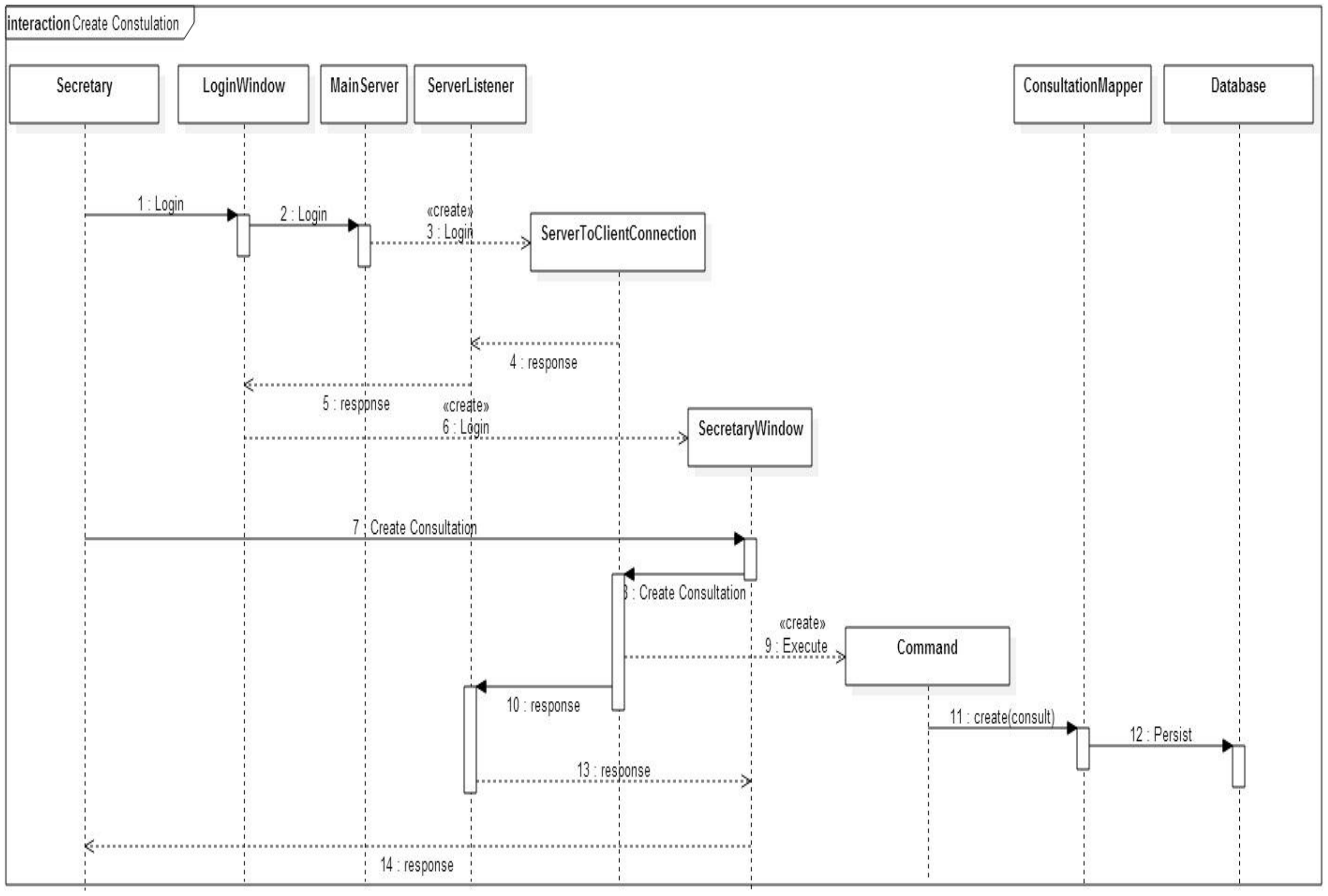
Component Diagram



Deployment Diagram



4. UML Sequence Diagrams

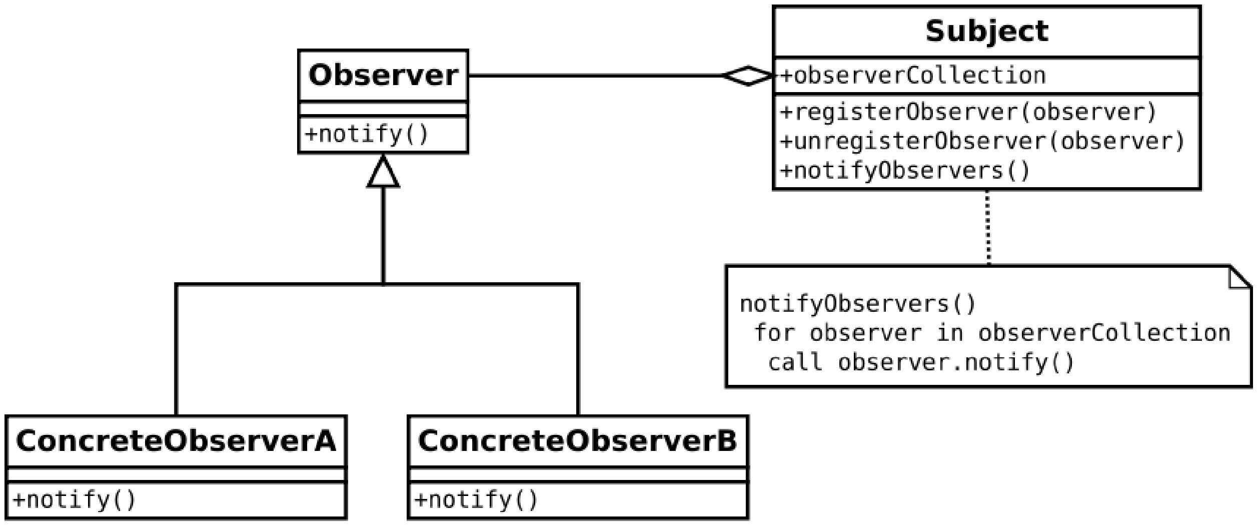


5. Class Design

5.1 Design Patterns Description

The **observer pattern** is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern. The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.

The observer pattern can cause memory leaks, known as the lapsed listener problem, because in basic implementation it requires both explicit registration and explicit deregistration, as in the dispose pattern, because the subject holds strong references to the observers, keeping them alive. This can be prevented by the subject holding weak references to the observers.



In object-oriented programming, the **command pattern** is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters.

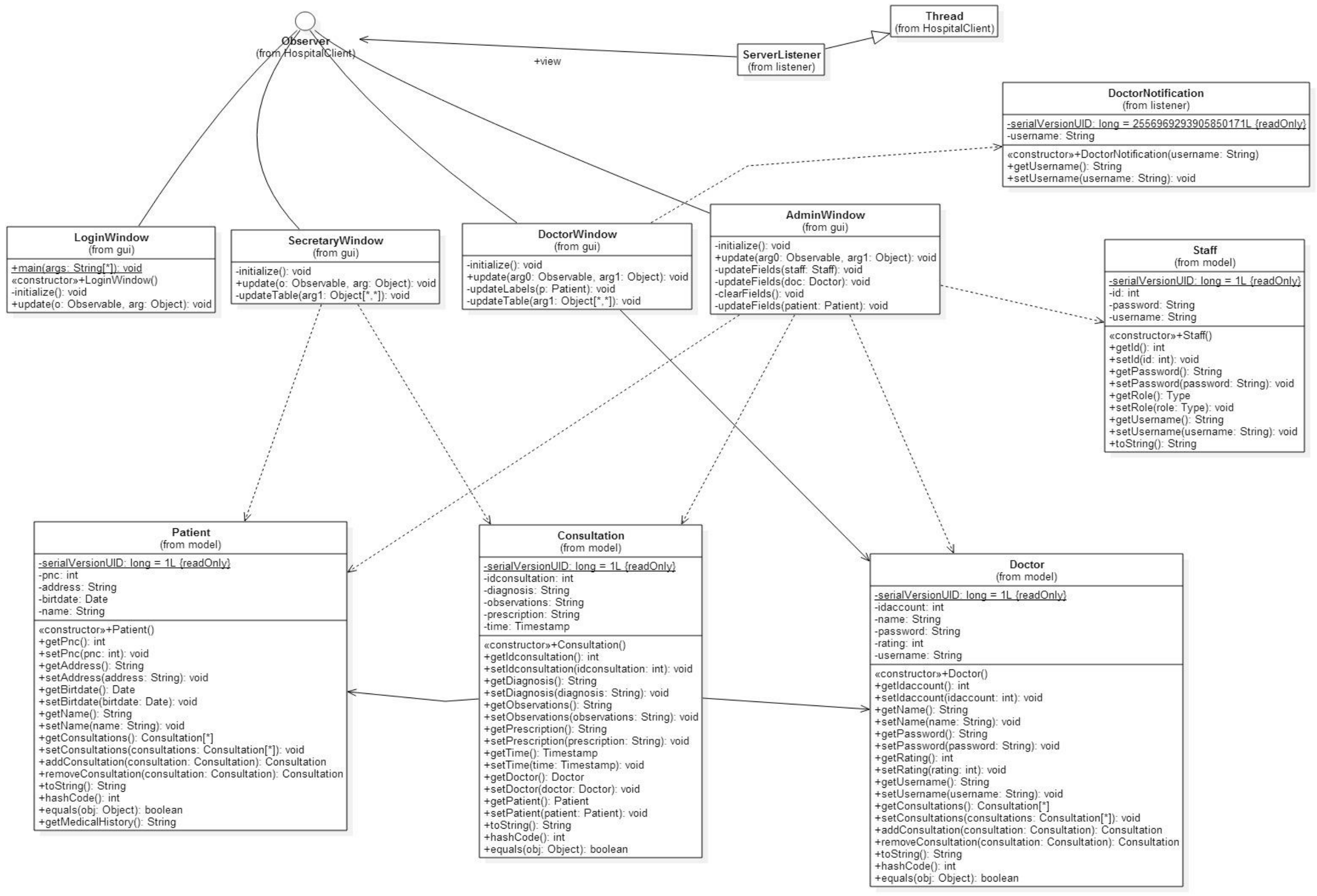
Four terms always associated with the command pattern are command, receiver, invoker and client. A command object knows about receiver and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command. The receiver then does the work. An invoker object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about command interface. Both an invoker object and several command objects are held by a client object. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.

Using command objects makes it easier to construct general components that need to delegate, sequence or execute method calls at a time of their choosing without the need to know the class of the method or the method parameters. Using an invoker object allows bookkeeping about command executions to be conveniently performed, as well as implementing different modes for commands, which are managed by the invoker object, without the need for the client to be aware of the existence of bookkeeping or modes.

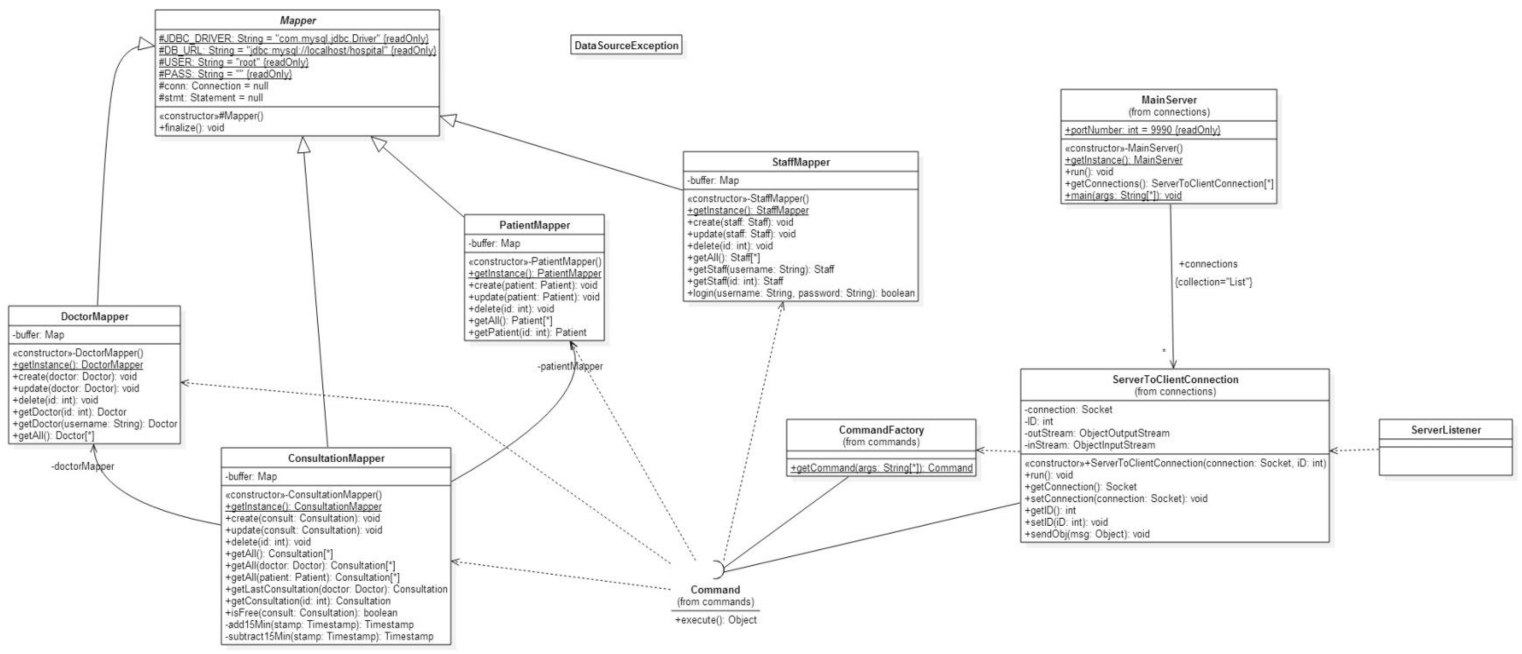
5.2 UML Class Diagram

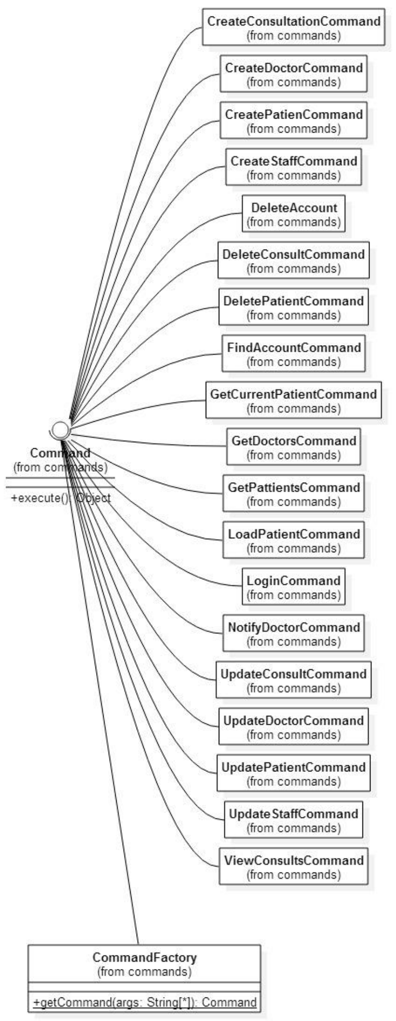
Due to dimension criteria, the class diagram will be divided into 3 parts:

1. The client
2. The server - connections
3. The server – commands

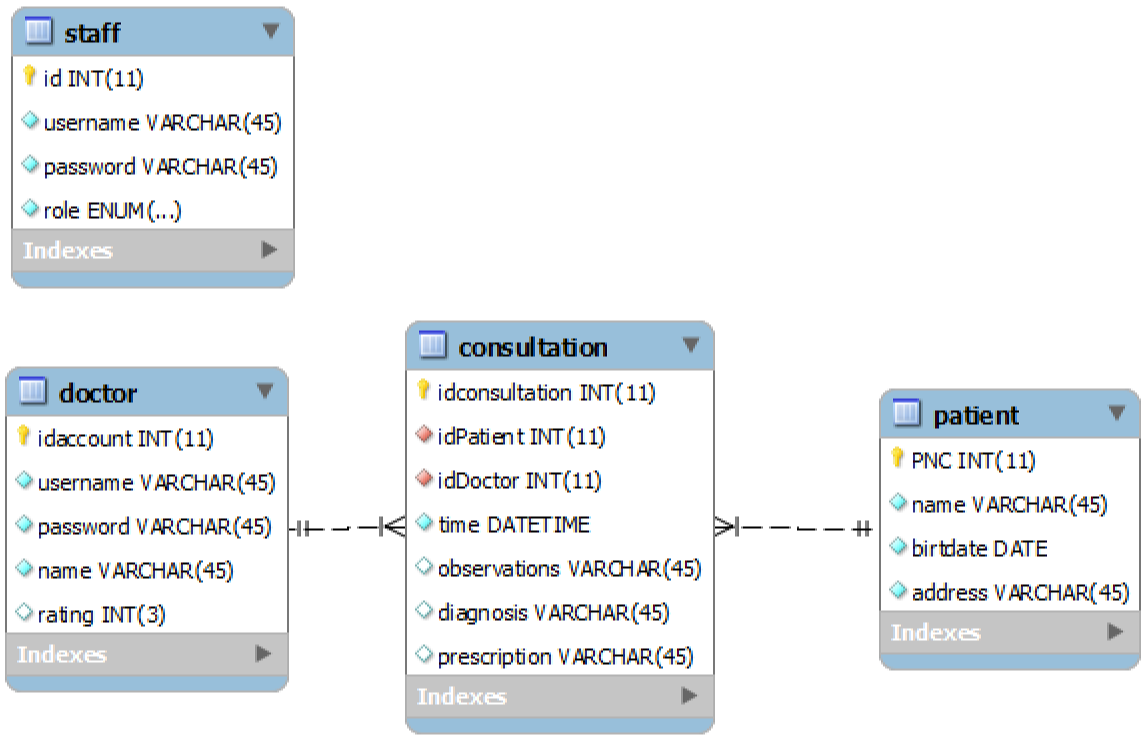


Observer
(from HospitalClient)
+view
ServerListener
(from listener)
Thread
(from HospitalClient)
DoctorNotification
(from listener)
-serialVersionUID: long = 2556969293905850171L {readOnly}
-username: String
«constructor»+DoctorNotification(username: String)
+getUsername(): String
+setUsername(username: String): void
LoginWindow
(from gui)
+main(args: String[*]): void
«constructor»+LoginWindow()
-initialize(): void
+update(o: Observable, arg: Object): void
SecretaryWindow
(from gui)
-initialize(): void
+update(o: Observable, arg: Object): void
-updateTable(arg1: Object[*,*]): void
DoctorWindow
(from gui)
-initialize(): void
+update(arg0: Observable, arg1: Object): void
-updateLabels(p: Patient): void
-updateTable(arg1: Object[*,*]): void
AdminWindow
(from gui)
-initialize(): void
+update(arg0: Observable, arg1: Object): void
-updateFields(staff: Staff): void
-updateFields(doc: Doctor): void
-clearFields(): void
-updateFields(patient: Patient): void
Staff
(from model)
-serialVersionUID: long = 1L {readOnly}
-id: int
-password: String
-username: String
«constructor»+Staff()
+getId(): int
+setId(id: int): void
+getPassword(): String
+setPassword(password: String): void
+getRole(): Type
+setRole(role: Type): void
+getUsername(): String
+setUsername(username: String): void
+toString(): String
Patient
(from model)
-serialVersionUID: long = 1L {readOnly}
-pnc: int
-address: String
-birtdate: Date
-name: String
«constructor»+Patient()
+getPnc(): int
+setPnc(pnc: int): void
+getAddress(): String
+setAddress(address: String): void
+getBirtdate(): Date
+setBirtdate(birtdate: Date): void
+getName(): String
+setName(name: String): void
+getConsultations(): Consultation[*]
+setConsultations(consultations: Consultation[*]): void
+addConsultation(consultation: Consultation): Consultation
+removeConsultation(consultation: Consultation): Consultation
+toString(): String
+hashCode(): int
+equals(obj: Object): boolean
+getMedicalHistory(): String
Consultation
(from model)
-serialVersionUID: long = 1L {readOnly}
-idconsultation: int
-diagnosis: String
-observations: String
-prescription: String
-time: Timestamp
«constructor»+Consultation()
+getIdconsultation(): int
+setIdconsultation(idconsultation: int): void
+getDiagnosis(): String
+setDiagnosis(diagnosis: String): void
+getObservations(): String
+setObservations(observations: String): void
+getPrescription(): String
+setPrescription(prescription: String): void
+getTime(): Timestamp
+setTime(time: Timestamp): void
+getDoctor(): Doctor
+setDoctor(doctor: Doctor): void
+getPatient(): Patient
+setPatient(patient: Patient): void
+toString(): String
+hashCode(): int
+equals(obj: Object): boolean
Doctor
(from model)
-serialVersionUID: long = 1L {readOnly}
-idaccount: int
-name: String
-password: String
-rating: int
-username: String
«constructor»+Doctor()
+getIdaccount(): int
+setIdaccount(idaccount: int): void
+getName(): String
+setName(name: String): void
+getPassword(): String
+setPassword(password: String): void
+getRating(): int
+setRating(rating: int): void
+getUsername(): String
+setUsername(username: String): void
+getConsultations(): Consultation[*]
+setConsultations(consultations: Consultation[*]): void
+addConsultation(consultation: Consultation): Consultation
+removeConsultation(consultation: Consultation): Consultation
+toString(): String
+hashCode(): int
+equals(obj: Object): boolean


Mapper
#JDBC_DRIVER: String = "com.mysql.jdbc.Driver" {readOnly}
#DB_URL: String = "jdbc:mysql://localhost/hospital" {readOnly}
#USER: String = "root" {readOnly}
#PASS: String = "" {readOnly}
#conn: Connection = null
#stmt: Statement = null
«constructor»#Mapper()
+finalize(): void
DataSourceException
MainServer
(from connections)
+portNumber: int = 9990 {readOnly}
«constructor»-MainServer()
+getInstance(): MainServer
+run(): void
+getConnections(): ServerToClientConnection[*]
+main(args: String[*]): void
+connections
{collection="List"}
*
StaffMapper
-buffer: Map
«constructor»-StaffMapper()
+getInstance(): StaffMapper
+create(staff: Staff): void
+update(staff: Staff): void
+delete(id: int): void
+getAll(): Staff[*]
+getStaff(username: String): Staff
+getStaff(id: int): Staff
+login(username: String, password: String): boolean
PatientMapper
-buffer: Map
«constructor»-PatientMapper()
+getInstance(): PatientMapper
+create(patient: Patient): void
+update(patient: Patient): void
+delete(id: int): void
+getAll(): Patient[*]
+getPatient(id: int): Patient
DoctorMapper
-buffer: Map
«constructor»-DoctorMapper()
+getInstance(): DoctorMapper
+create(doctor: Doctor): void
+update(doctor: Doctor): void
+delete(id: int): void
+getDoctor(id: int): Doctor
+getDoctor(username: String): Doctor
+getAll(): Doctor[*]
-patientMapper
-doctorMapper
ConsultationMapper
-buffer: Map
«constructor»-ConsultationMapper()
+getInstance(): ConsultationMapper
+create(consult: Consultation): void
+update(consult: Consultation): void
+delete(id: int): void
+getAll(): Consultation[*]
+getAll(doctor: Doctor): Consultation[*]
+getAll(patient: Patient): Consultation[*]
+getLastConsultation(doctor: Doctor): Consultation
+getConsultation(id: int): Consultation
+isFree(consult: Consultation): boolean
-add15Min(stamp: Timestamp): Timestamp
-subtract15Min(stamp: Timestamp): Timestamp
ServerToClientConnection
(from connections)
-connection: Socket
-ID: int
-outStream: ObjectOutputStream
-inStream: ObjectInputStream
«constructor»+ServerToClientConnection(connection: Socket, iD: int)
+run(): void
+getConnection(): Socket
+setConnection(connection: Socket): void
+getID(): int
+setID(iD: int): void
+sendObj(msg: Object): void
CommandFactory
(from commands)
+getCommand(args: String[*]): Command
ServerListener
Command
(from commands)
+execute(): Object


CreateConsultationCommand
(from commands)
CreateDoctorCommand
(from commands)
CreatePatienCommand
(from commands)
CreateStaffCommand
(from commands)
DeleteAccount
(from commands)
DeleteConsultCommand
(from commands)
DeletePatientCommand
(from commands)
FindAccountCommand
(from commands)
GetCurrentPatientCommand
(from commands)
Command
(from commands)
+execute(): Object
GetDoctorsCommand
(from commands)
GetPattientsCommand
(from commands)
LoadPatientCommand
(from commands)
LoginCommand
(from commands)
NotifyDoctorCommand
(from commands)
UpdateConsultCommand
(from commands)
UpdateDoctorCommand
(from commands)
UpdatePatientCommand
(from commands)
UpdateStaffCommand
(from commands)
ViewConsultsCommand
(from commands)
CommandFactory
(from commands)
+getCommand(args: String[*]): Command

6. Data Model



7. System Testing

The system testing was performed using unit testing for the Data Mappers, integration testing for the integration of the client with the server, and validation testing for the finished system. The unit testing was performed using jUnit in the TestCases class. Integration testing was performed using the graphical interface by testing each functionality in turn for main success and fail scenarios (data-flow strategy).

Each Use Case has been tested using the data-flow strategy. We have tested for the main success scenario and one fail scenario.

8. Bibliography

- http://en.wikipedia.org/wiki/Command_pattern
- http://en.wikipedia.org/wiki/Observer_pattern
- http://en.wikipedia.org/wiki/Observer_pattern
- http://docs.oracle.com/javase/tutorial/networking/sockets/index.html
- http://www.javaworld.com/article/2077322/core-java/sockets-programming-in-java-a-tutorial.html
- http://docs.oracle.com/javase/tutorial/jndi/objects/serial.html