A chat program

1 Introduction

1.1 Overview

This documentation is split into a **theoretical part** in which the use of **design patterns** is described. The second part deals with the implementation itself.

1.2 Objectives

1.2.1 Theoretic part

The goal was to develop a model of a chat program, taking into account the functional characters of a chat server and its appropriate chat clients. Furthermore, a complete **UML diagram** for documentation purposes was created.

1.2.2 Practical part

The goal of the practical part was to develop a **working chat client and server**. For this purpose the program should be as close to the theory as possible.

1.3 Brief description

1.3.1 Theoretic part

The following functions are critical for a chat program :

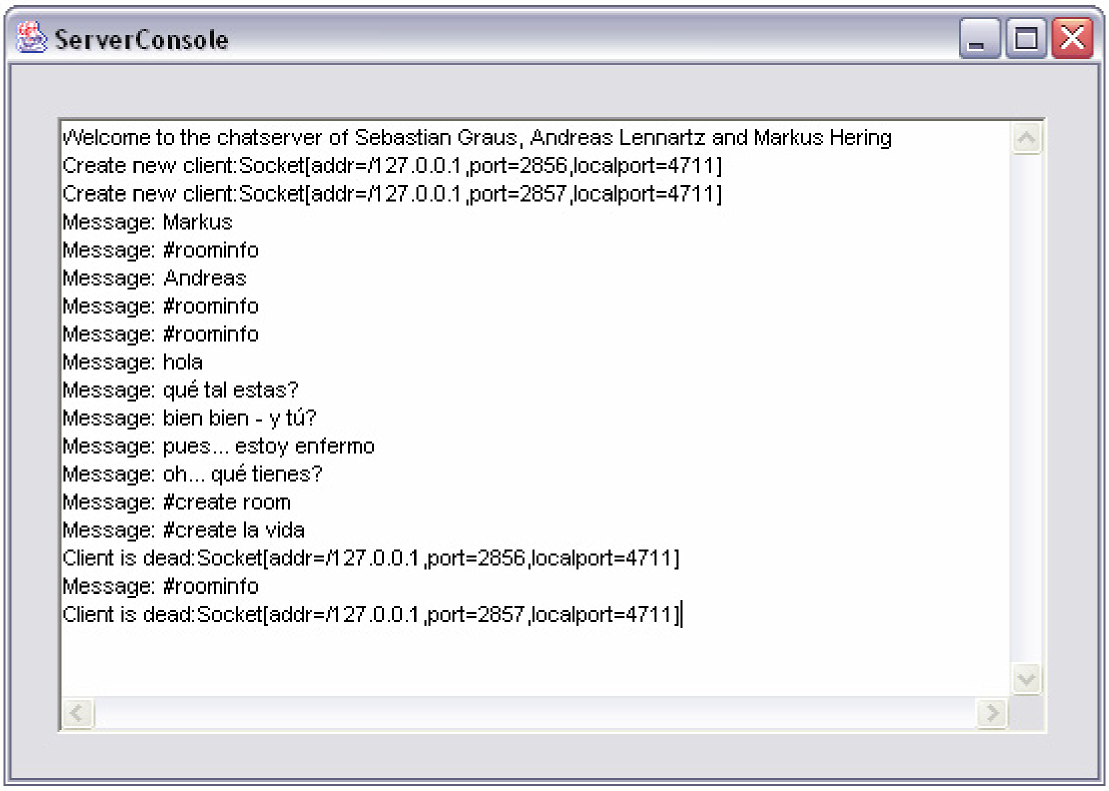
- **chat server** (handles the administration of the clients)
- a **default room** in which a user enters if he logs on
- every member has the possibility to **open new rooms** and to talk to a selected circle, sent
- messenges can be read by the respectively present persons only
- it is displayed **who** is in the chat and in **which room**

In order to fulfill these requirements the components and necessary object classes must be found and analyzed with the use of suitable **design patterns**.

1.3.2 Practical part

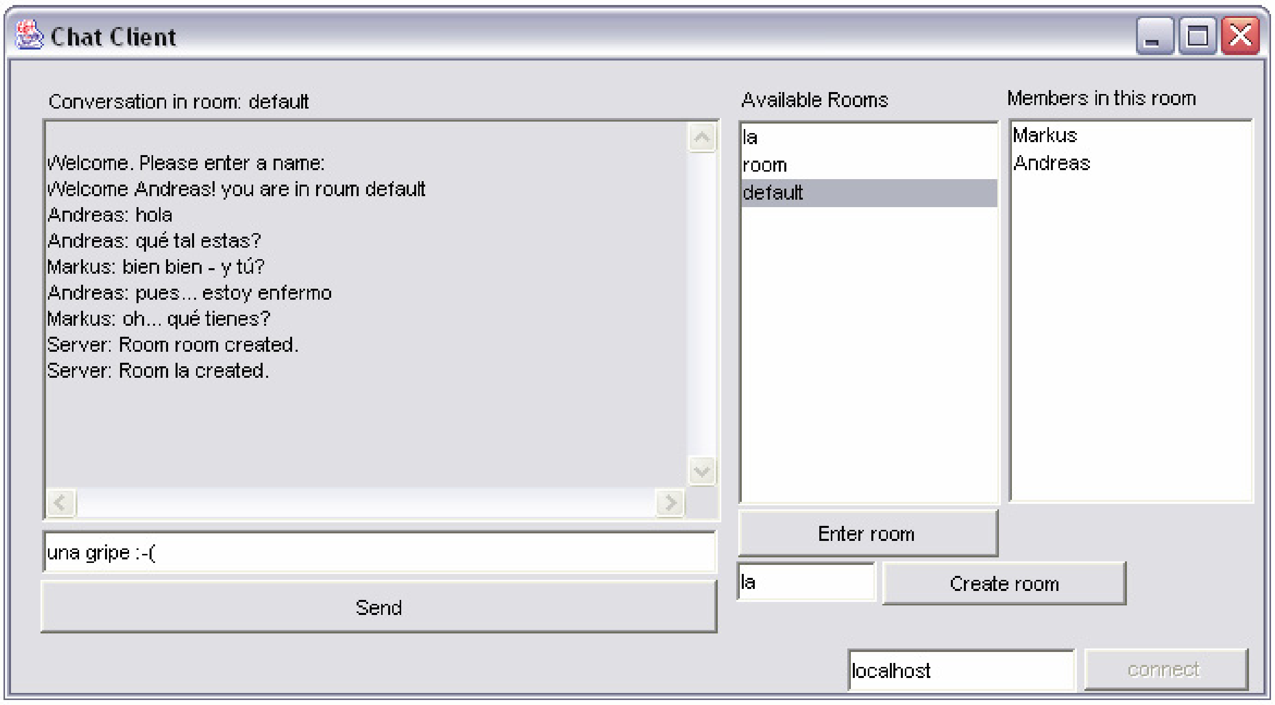
To realize the project the program is **divide into a server and a client.**

The ***ServerConsole*** displays everything which happens during the server is online.



```
Welcome to the chatserver of Sebastian Graus, Andreas Lennartz and Markus Hering
Create new client:Socket[addr=/127.0.0.1,port=2856,localport=4711]
Create new client:Socket[addr=/127.0.0.1,port=2857,localport=4711]
Message: Markus
Message: #roominfo
Message: Andreas
Message: #roominfo
Message: #roominfo
Message: hola
Message: qué tal estas?
Message: bien bien - y tú?
Message: pues... estoy enfermo
Message: oh... qué tienes?
Message: #create room
Message: #create la vida
Client is dead:Socket[addr=/127.0.0.1,port=2856,localport=4711]
Message: #roominfo
Client is dead:Socket[addr=/127.0.0.1,port=2857,localport=4711]
```

The user interface of the chat client displays the current conversation, the rooms which were created and the clients who are in the selected room. Before you are able to take part in the conversation you have to connect with the “***connect- Button***”.



```
Welcome. Please enter a name:
Welcome Andreas! you are in roum default
Andreas: hola
Andreas: qué tal estas?
Markus: bien bien - y tú?
Andreas: pues... estoy enfermo
Markus: oh... qué tienes?
Server: Room room created.
Server: Room la created.
```

una gripe :-(

la

localhost

2 Applied design patterns

For the **server** the following design patterns are used:

- ***Singleton*** (guarantees that there exists just (exactly) one authority of the server)
- ***Mediator*** (cares for the communication (cooperation) of the separate chat clients with the server)

For the client are these design pattern used:

- ***Decorator*** (serve for creating the scrollbars, buttons, etc. in the user interfaces)
- ***Observer*** (is used for the clients perceive a condition change of dependent objects (the list with rooms) and actualize appropriately)

2.1 The Server in Theory

2.1.1 Singleton

As aforementioned the chat program is divided into two parts, in a server program and a client program.
It has to be considered that there could be **many different clients**, but only **one server**.
In principle the application is developed like that: The individual clients are not directly connected, but they are able to communicate with each other over the server.
The server is able send

- messages
- instructions and
- information

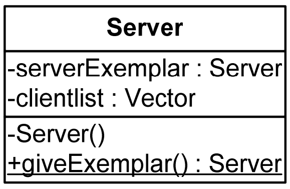
to the clients. The clients can run on other computers than the server. Thus communication takes place over a **network connection** (e.g. the Internet). As soon as a new client will be created, this will start a connection with the server - over an **information channel**. The server informs the client about

- other existing clients
- rooms and
- other important information

The client again continues to lead this information to its **graphic user interface** (named GUI in the following) and **waits for user inputs** afterwards. As soon as the user types e.g. a message line for chat, this message is sent by the client to the server for further processing. Then the server passes this message on to the other clients.
First let´s look at the server, its requirements and with which design pattern the above mentioned can be transferred into a **draft model**.
Requirements of the server:

- It may exist only one server, because the clients must be administered **central**.
- All clients are **assigned** to the server. (From the view of the client: it is assigned to each clientexactly **one server**.)
- It administers communication between the clients.

Around to ensure that the server exists only once, the design pattern "***Singleton***" as our **production design pattern is used**.
The purpose of the "***Singleton***" agrees with the purpose of the first requirements to the server: Singleton secures that a class possesses exactly just one copy and a **global point of access** after it is made available.
This global point of access is used in the further process by the clients to communicate over the server with each other. It is ensured by the uniqueness of the server that the clients have a central "**partner**", over whom the information can be get and send by messages in the form of strings.
An UML-draft of the server class follows, sketched with the "***Singleton***" pattern:

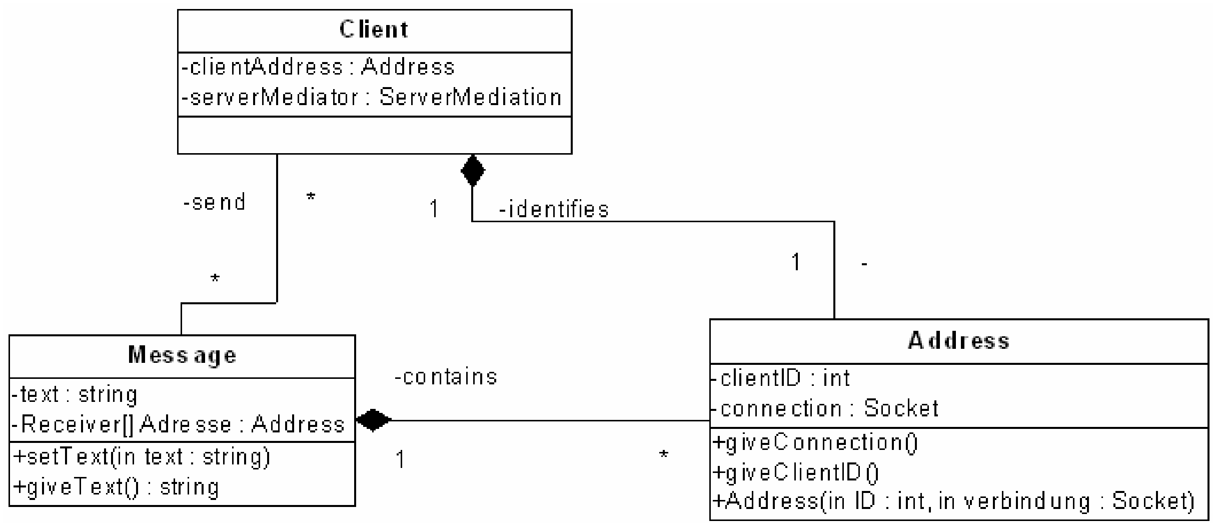


"-" means thereby of the Java keyword private, "+" indicates the keyword public. Underlining a function means the java keyword static. (this is just the theoretic attempt – the model of all classes shows the practical implementation)

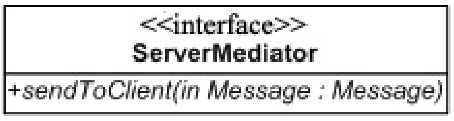
Thus the server gets a private protected constructor, causes that our server can be instanced by no other class except the server himself. The server saves the only instance which it may pass on in the variable ***serverExemplar***. This variable will only be created if it is requested with the static method ***giveExemplar()*** by a client.
If a client connects with the server, the client has to get an instance of the server over this connection. The client can address all other clients (over the server) via this server copy. The server notes with the structure of the connection the client as well as the associated connecting information likewise and stores these in the vector ***clientList***, in order to be able to address it further.

2.1.2 Mediator

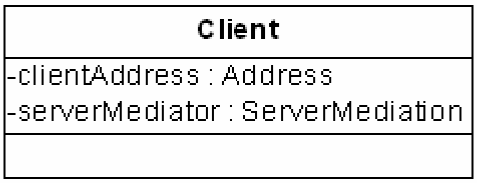
As already partly described in the above part, the server works as a "***Mediator***" between the clients. The server must administer the connection of the individual clients, because they are not able to communicate by themselves.
Therefore there is a need to create a function for the server which makes it possible to send messages to a certain client. The returned value is a message which contains the message itself and as well an address identification of the client and if necessary further information about the message. The name of this function will be ***sendToclient***.
In order to keep the addressing as general as possible, an object ***address*** *is introduced*. This represents the address of one client and contains all **important connection information** for the server. Basis for this is a clear ID, with which each client is identified, and a corresponding Java connection of the class ***socket***. For the representation of the message likewise a class ***message*** *is* introduced. Each message has a field with all receiving addresses of the type address.
Because each message must contain at least one address of the receiver, this results in an **entity-relationship between the message and the address** (One message contains 1 or several addresses).
Besides that each client has its own address, with which he can be identified clearly.
In the following you see an **UML structure** of the new classes ***address*** and ***message***, as well as the **connection of the *address*** with the clients:



A requirement of our chat application was that the clients can **communicate over the server** with each other. In order to represent this circumstance, the **behaviour design pattern** "***Mediator***" is used. This design pattern has the purpose to **encase the interaction** of a quantity of objects in one object.
In our case the server encases the quantity of all clients in itself. This brings the advantage that the clients are coupled among themselves loosely. The **missing of clients** will not be considered by the other clients, because now the server object "**cares**" for it. The clients have to know only the server and have no direct relations among themselves. Everything is passed on by the server to the appropriate clients. The server serves as a "***Mediator***".
Therefore introducing an interface ***Serverclient***. This interface implements the above described function to send messages to the individual clients.



Besides this the above introduced client class is extended with this interface:



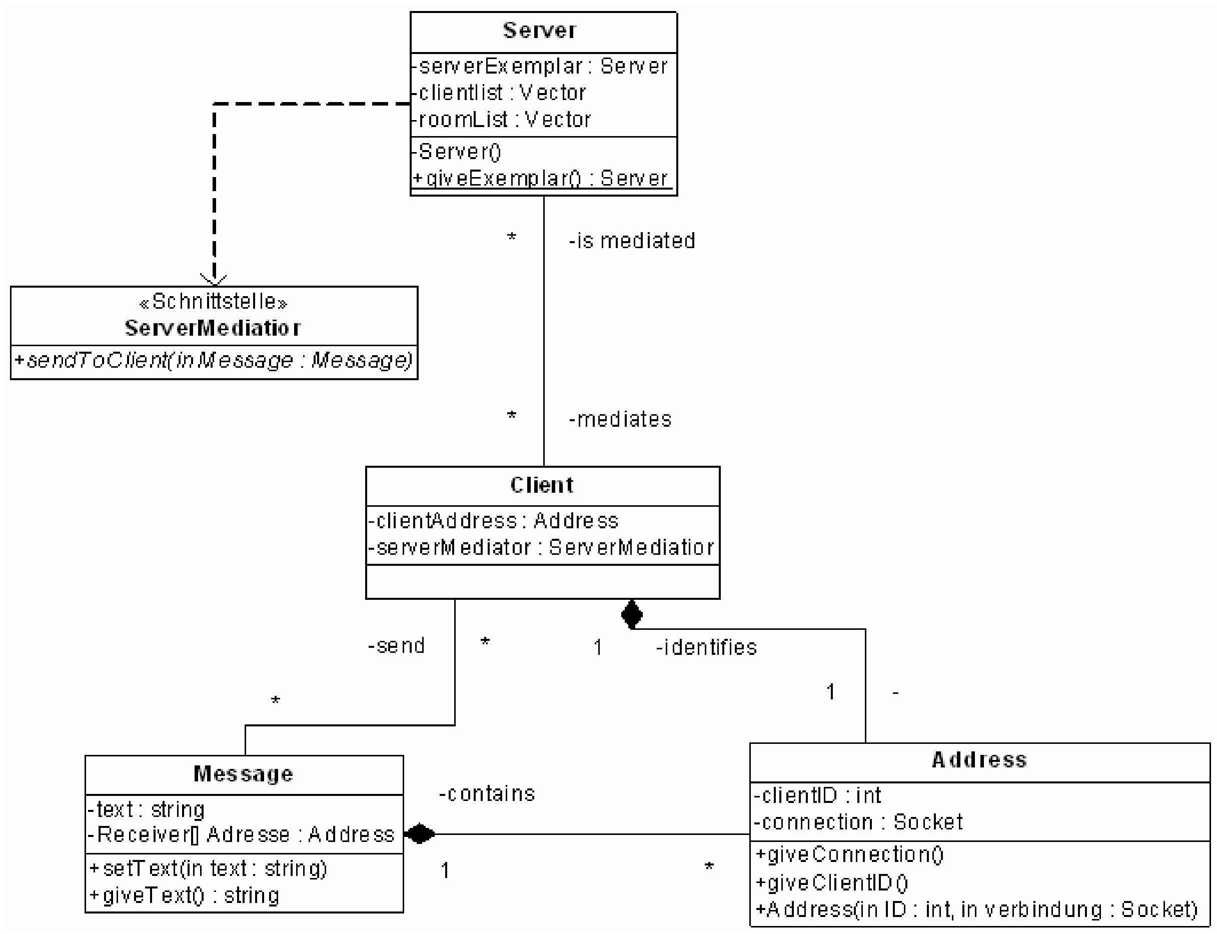
Because the client will be the centre of our view later, assigning him here with no further functionality.

According to the design pattern of the client now having the following requirements to our **classes and concrete objects**:
Server requirements:
- The server implements an interface for the **interaction with the client objects**
- The servers implements the total behaviour by **coordination of the client objects**
- The servers **knows and administers** the client objects

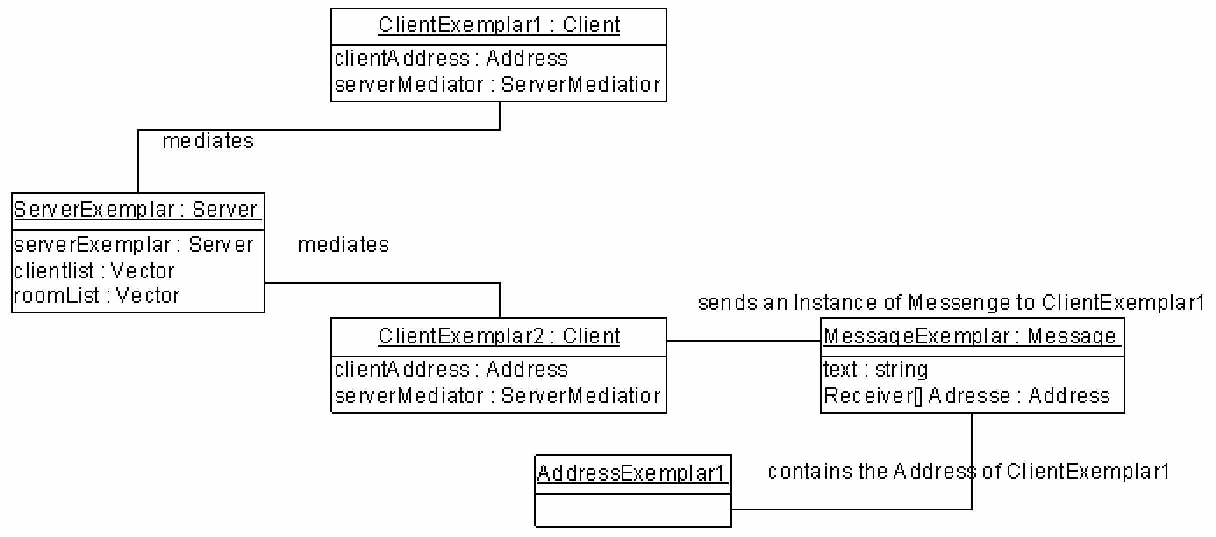
Client requirements:
- Each client class **knows** its ***ServerClient*** interface
- Each client object **communicates** with the other clients over the ***ServerClient***

Below now the **UML structures** which are describing these requirements. This is a **class diagram** and a structure of the **concrete objects**. From these structures it is evident that the message objects are now transferred by one client to the next client.

Class diagram:



UML-structure of the concrete objects:

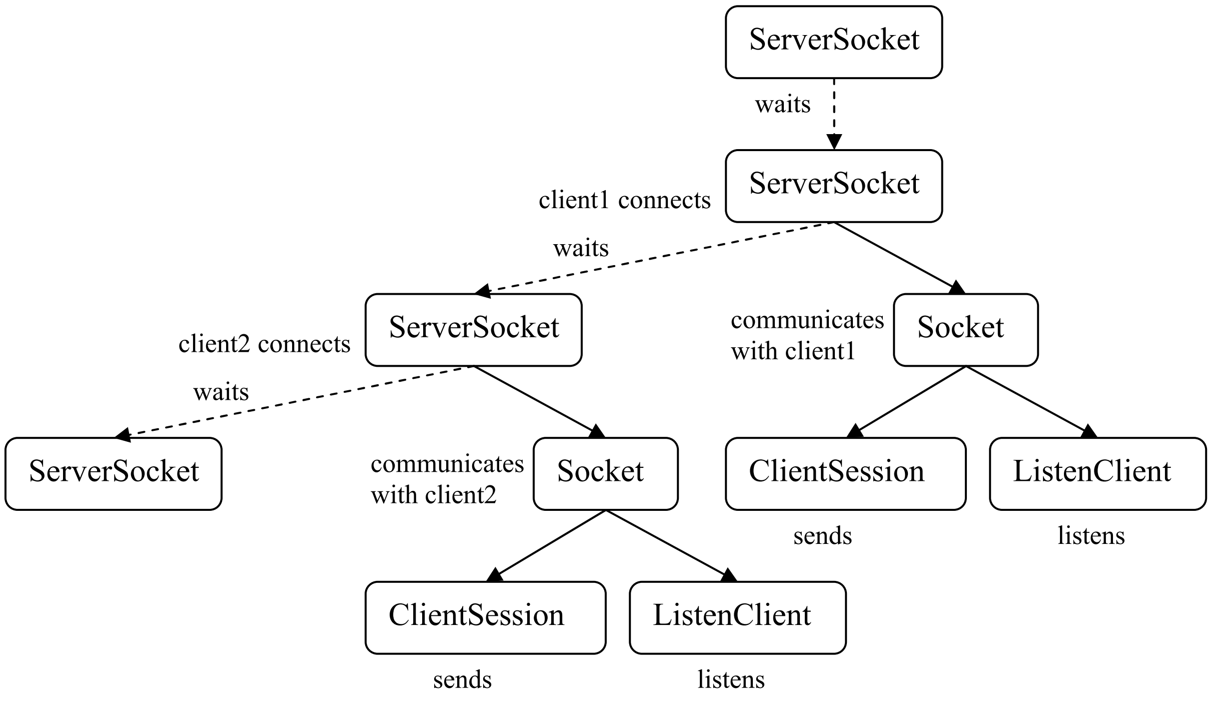


2.2 The Server in practical

The server program includes 4 classes:

- ServerMain
- ClientSession
- ListenClient
- ServerMainGUI

The class ***ServerMain*** is the administrative centre of the whole server. It includes the **main method** on the one hand and contains the class ***ServerSocket*** which belongs to the java.net imports on the other hand. This class is responsible for **listening on the ports** (waiting for clients to logon). If a client connects to the server the ***ServerSocket*** accepts the connection, creates a new ***Socket*** which is passed to the ***ClientSession*** and **keeps listening**. The ***ClientSession*** uses this ***Socket*** in order to keep the connection with the client.
Moreover the class ***ServerMain*** includes lists of the rooms and of the clients in each room. It is able to **create** and **delete** rooms and to **move** the clients between the rooms.



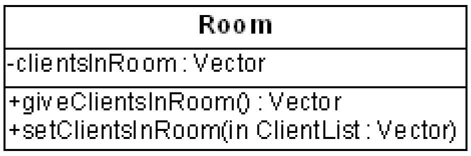
ClientSession is the class which manages the **main communication** with the clients. It is responsible for

sending messages to the client. Therefore it **starts the thread *ListenClient***, which is listing constantly, if the client is sending messages.
Because sending and listening have to run parallel two independent threads were realized(***ListenClient*** and ***ClientSession***).
The class ***ServerMainGUI*** only includes the user interface and manages the displaying of status messages.
The address translation and administration is completely included in the imported default java class ***java.net.socket*** Therefore it is no need to work with the ***address object***, like in the theoretic part.

2.3 The client in theory

2.3.1 Observer

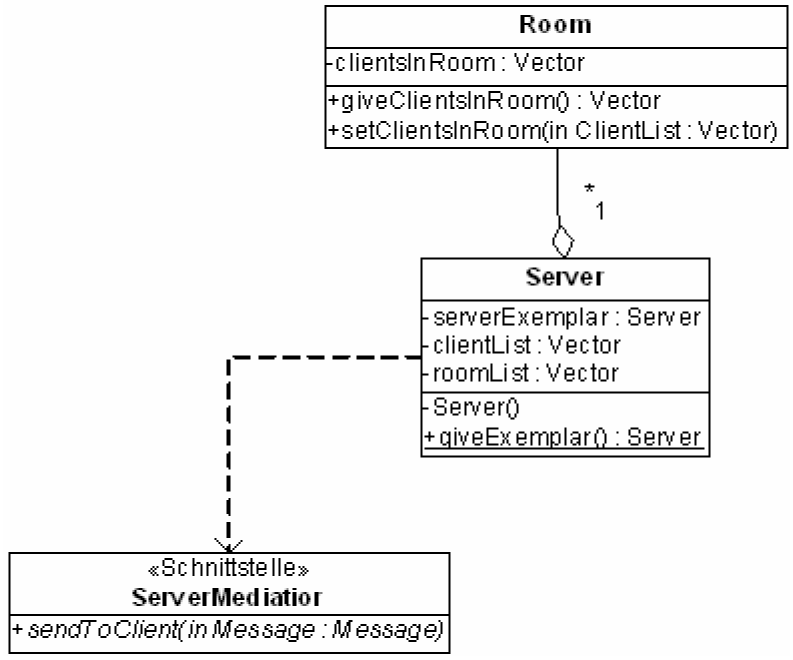
Now the individual clients can send messages to one or more clients. However there isn't the possibility of sending messages to a selected circle of clients in a so-called ***room*** yet.
Therefore an object ***room*** is introduced, which leads a list with all clients in it.


Room
-clientsInRoom : Vector
+giveClientsInRoom() : Vector
+setClientsInRoom(in ClientList : Vector)

These ***rooms*** are **administered by the server**. This leads to the following requirements for the server:

- The server must handle a list with all ***rooms***
- the server has to administer the ***rooms*** and return a list of all **clients which are in the room at the moment** on request.

The following diagram shows the interaction between the ***rooms*** and the server:



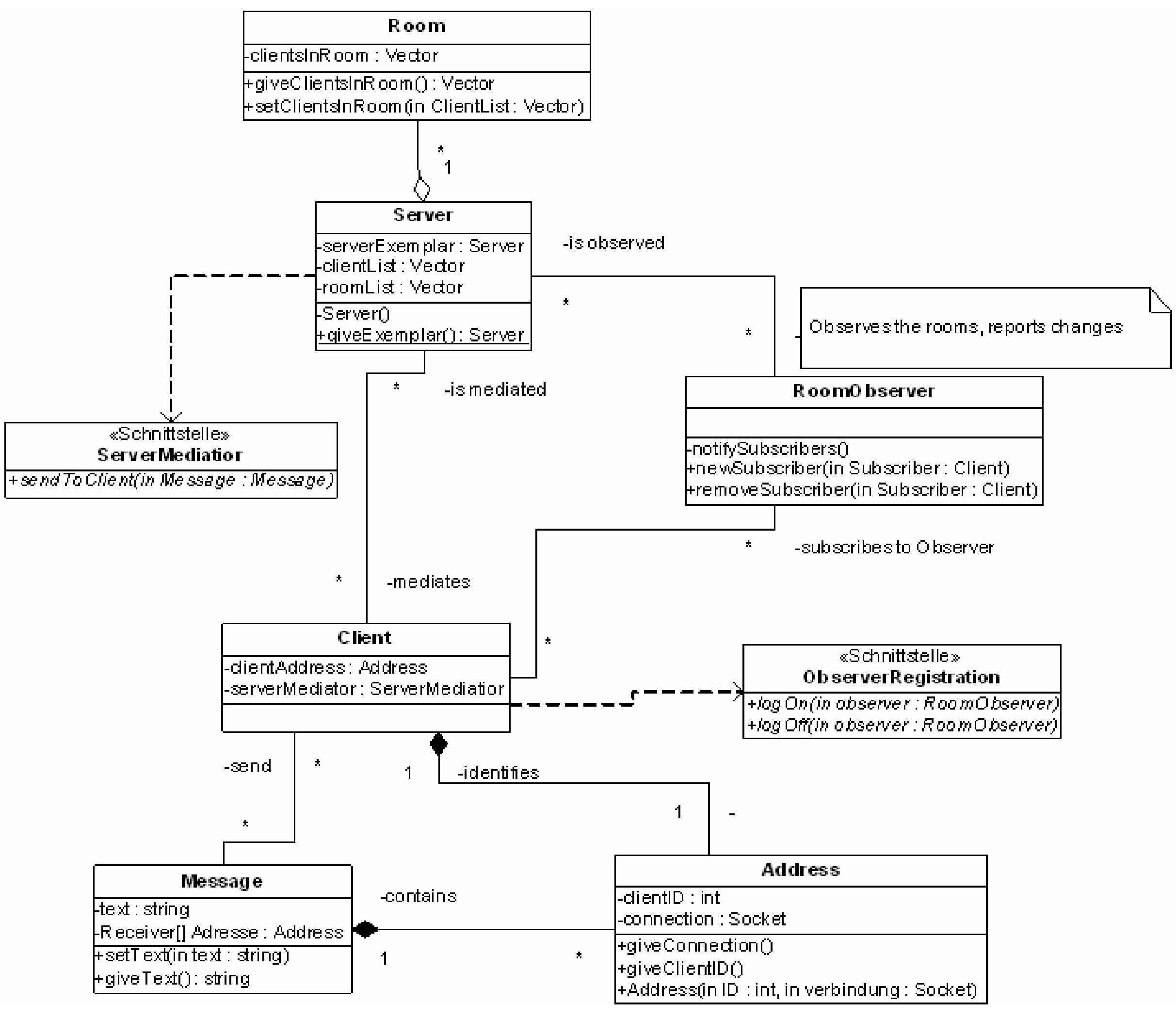
With this structure it becomes clear that the server administers the ***rooms*** now. The clients can access information about these ***rooms*** about the client interface ***ServerMediator***.
Each client can open a ***room*** and is able to talk to the other clients in this ***room***. Therefore a list with all in this room located clients has to be sent to the other, so that the user can directly address other clients from this list. Thus the following requirement for a client results:

- the client has to indicate a list with all **active rooms** and the **clients in this rooms**
- the client has to update this list **constantly**

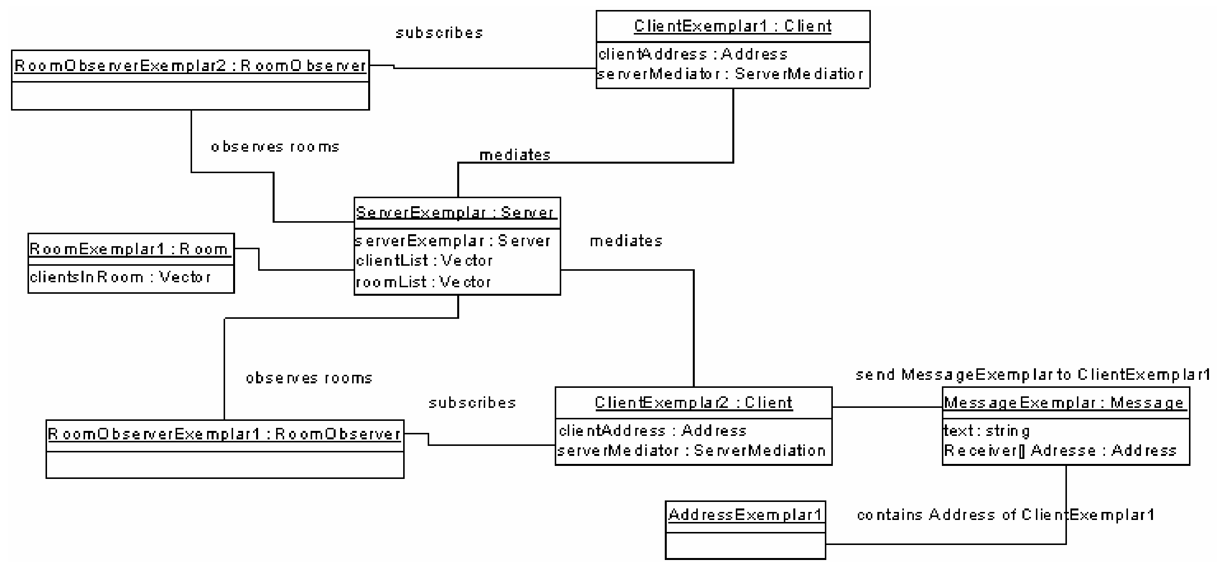
In order to become fair this circumstances, the behaviour pattern ***Observer*** is used. A class ***Observer*** is introduced, which observes the individual ***rooms*** in each case. The clients are subjects, which are informed about changes about the ***Observer***. In our case that would be the report over change at and in the ***rooms***, that means the creation and locking of a ***room*** and the coming and going of clients.
The ***Observer*** has the purpose to form a 1:n dependence between the server and the clients. The change of the condition of a clients causes that all other dependent clients are informed and updated automatically if necessary.

This happens, when the ***observer*** observes the server, and informs his respective client, as soon as there is a change with the condition of the servers (e.g. a client logs off).
Thus the server administers all central information about the **announced clients** and the **existing *rooms*** in the variables ***clientList*** and ***roomList***. As soon as a ***room*** is closed or a client logs out, the contents of those variables are changed. For the reason other clients have to get aware of this change and be able to update their own data and the announcement for the user they need to have ordered a so-called "***Abbonement***" from the "***Observer***". The task of the "***Observer***" is it to supervise the server constantly and to inform every registered client as soon as a change of the ***clientList*** or the ***roomList*** is made. The clients, which receive such a message, can evaluate and process these accordingly then.
The "***Observer***" has the public functions ***newSubscriber(Subscriber: client)*** and ***deleteSubscribert (Subscriber: client)***, over which a client can register on the "***Observer***". The closed function ***tellSubscribers()*** is called by the "***Observer***" whenever the server makes a change of its ***clientList*** or ***roomList***.
Each client must implement the interface ***ObserverRegistration***, in order to have a coupling between the client and the "***Observer***" as loose as possible. Over this interface the client can login and logout itself with its "***Observer***" over the functions ***logIn(observer: Observer)*** and ***logOut(observer: Observer)*** respectively.
In the following you see two **structure diagrams**, which integrate the pattern of the "***Observer***" into our existing model. These two diagrams show the **class diagram** and the **connection between the concrete objects**.

Class diagram:

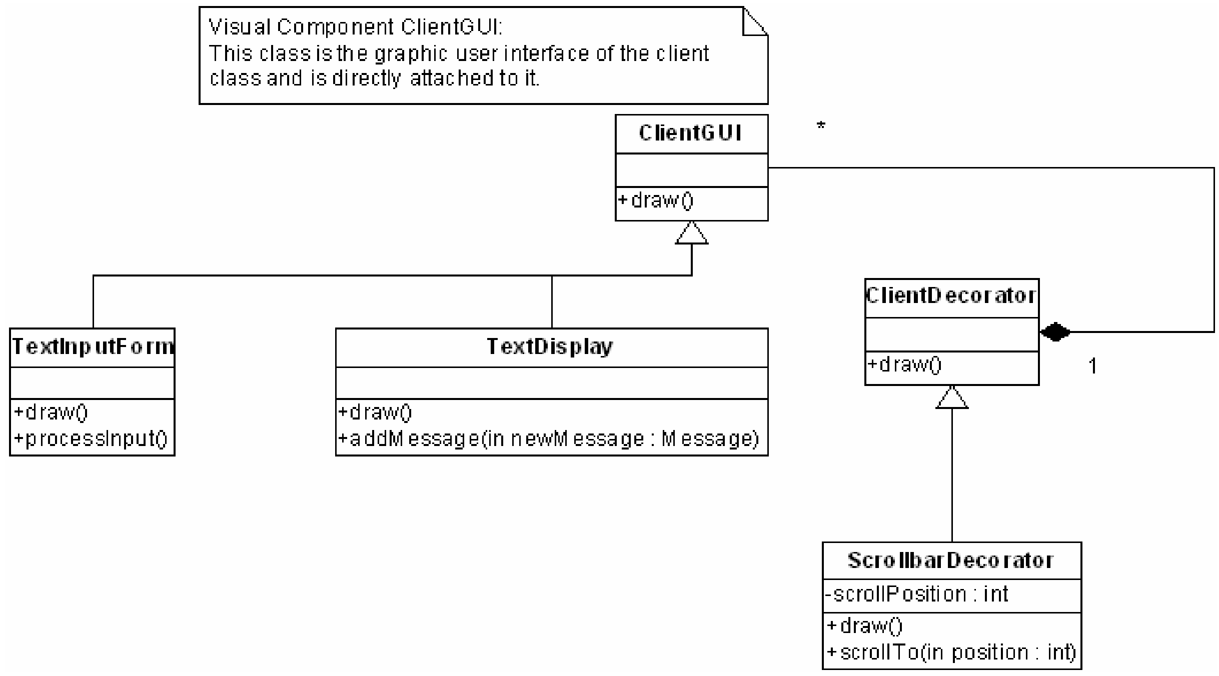


UML-structure of the concrete objects:



2.3.2 Decorator

Because it is explained now how communication functions between the clients, this can be turned into the visual level of our chat program.
The graphic user interface of our clients contains of three resembling **text windows** with Scrollbars (at the sides) and a **single-line text field**.
First proceeding from a class ***clientGUI***, which implements the graphic user interface of the client program and over which the user can **access to the client**.
The following **UML-class-diagram** shows the pattern of the "***Decorator***":


Visual Component ClientGUI:
This class is the graphic user interface of the client class and is directly attached to it.
ClientGUI
+draw()
*
ClientDecorator
+draw()
1
TextInputForm
+draw()
+processInput()
TextDisplay
+draw()
+addMessage(in newMessage : Message)
ScrollbarDecorator
-scrollPosition : int
+draw()
+scrollTo(in position : int)

2.4 The client in practical

The client program includes 3 classes:

- ClientMain
- ServerCom
- ClientGUI

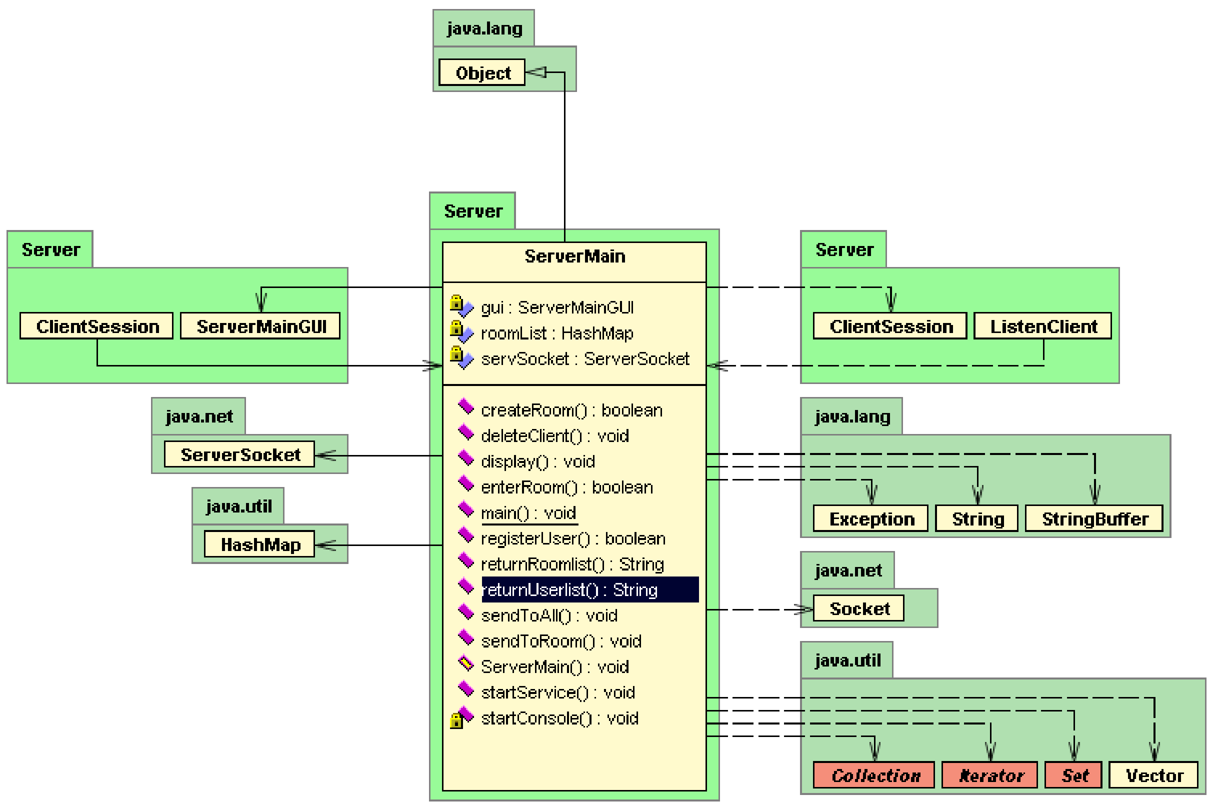
The class ***ClientMain*** is the administrative centre of the client. It includes the **main method** on the one hand and **handles the connection to the server** on the other hand. It creates the thread ***ServerCom*** which is always listening to the server. Moreover the ***ClientMain*** is responsible for sending the messages.
The class ***ClientGUI*** implements the user interface. That means it displays the current conversation, the rooms which were created and the clients who are in the current room. It receives all inputs of the users, namely keyboard entries and button clicks.

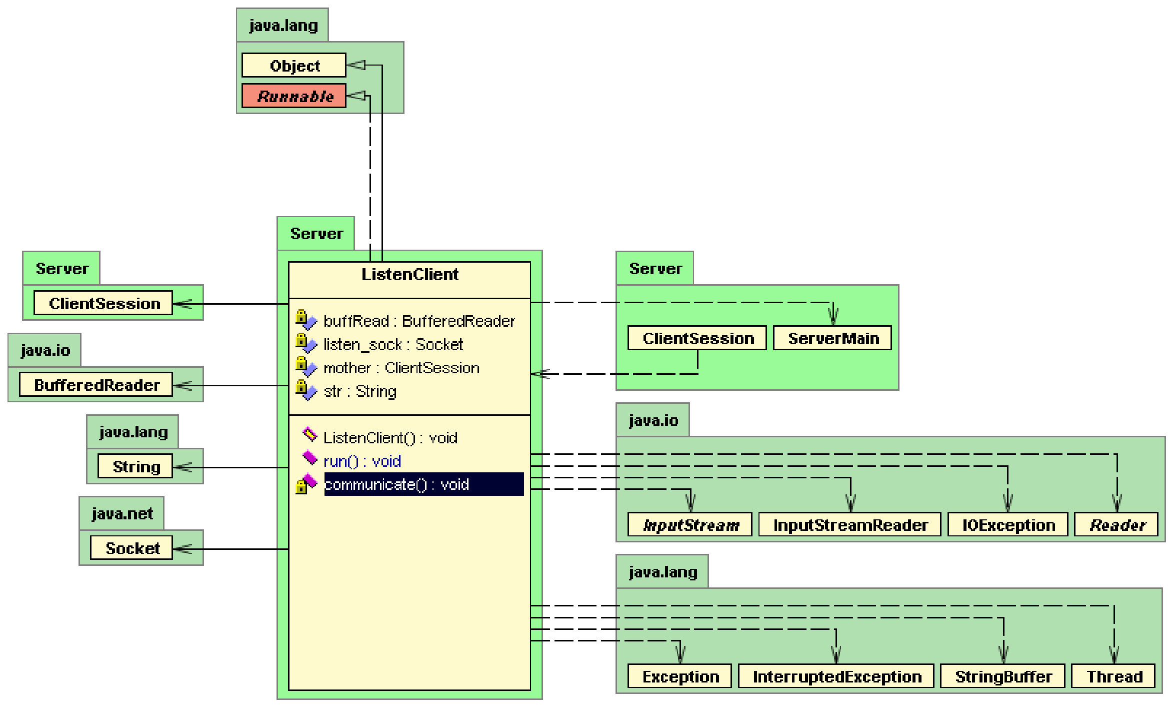
2.5 The communication in practical

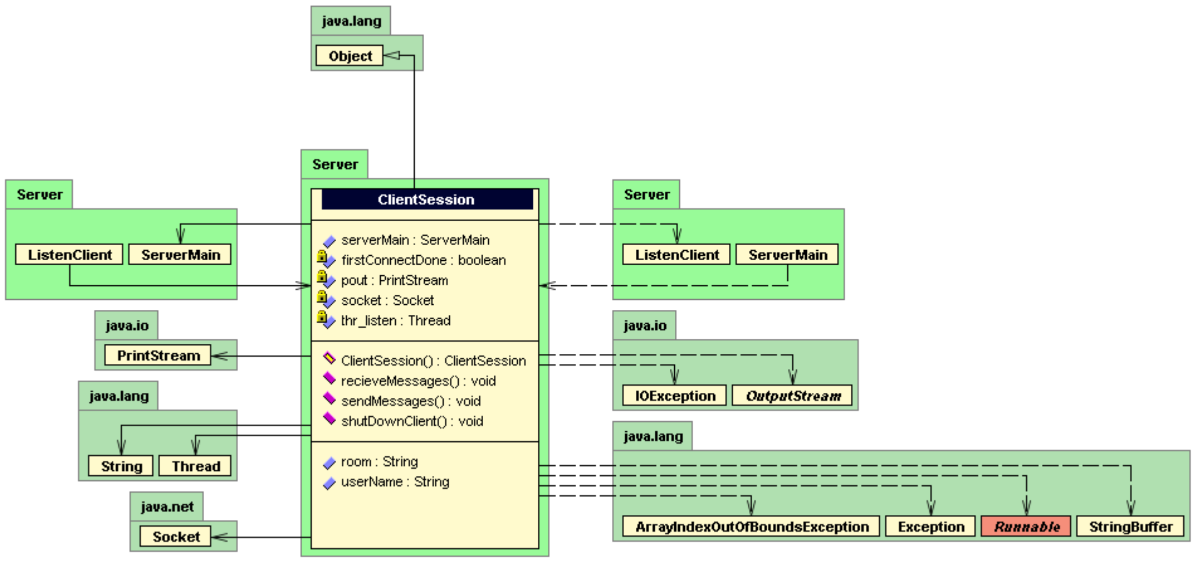
Deviating to the theory there is only a **socket** which is corresponding to the **address** in the theoretic part. Furthermore the class ***Message*** doesn't exist too. The messages are transferred as **strings** instead. An order to the server is realized with a **diamond** (#) in front of the string.

3 Model of all classes

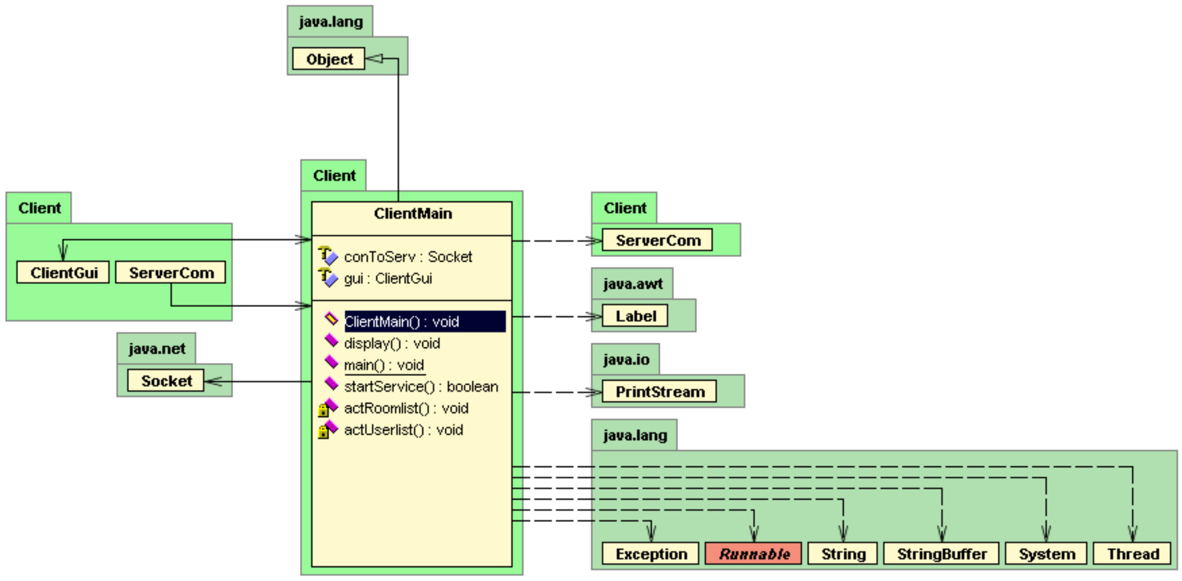
Classes of the server:
Referring to our part “**The Server in practical**” the following schematic diagrams show all classes:

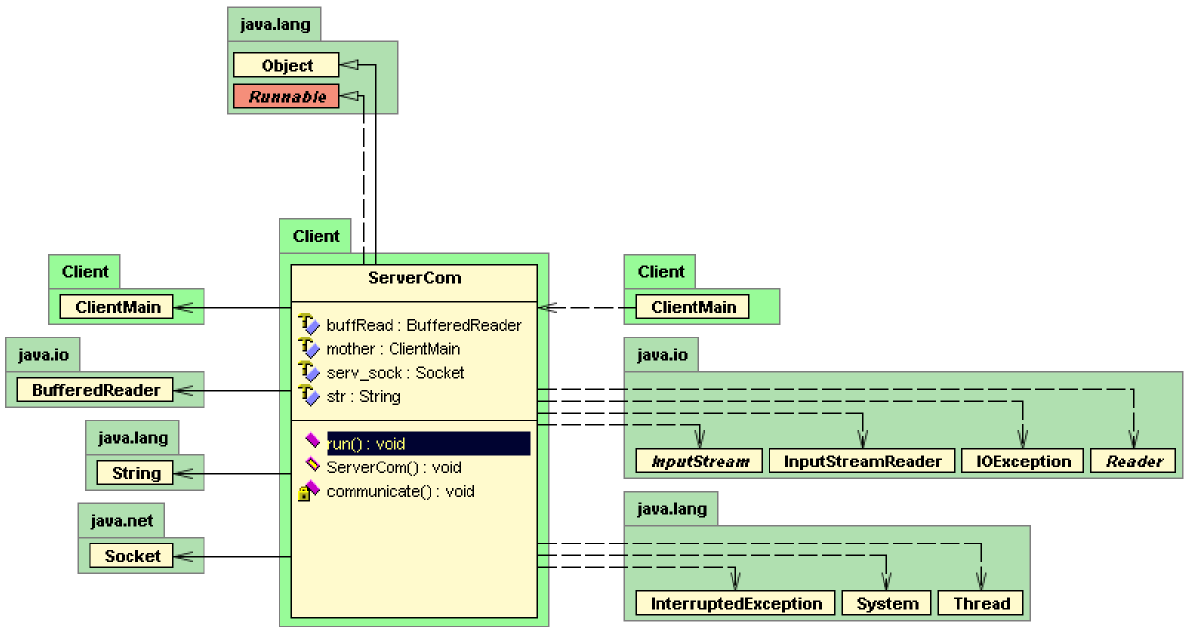






Classes of the client - The schematic view of the client classes:





The schematic views of the GUI-classes were left out because of there unimportance referring to the program.

4 Conclusion

At the end can be said, that the use of design patterns is a good assistance for the problem of class modelling. Particularly with more complex problem solving, solutions in indexing steps for individual ranges of the problem can be developed. However design patterns don't remove the task of the actual modelling – it only offers the possibility of solving occurring new or similar problems again and again as elegantly as possible (or at least without rough design errors).

As a conclusion of the practical part, it can be stated that the implementation of a chat in Java leads to many different problems. As it can be seen in the difference between the practical and the theoretic part, the solution sometimes looks very different to the draft. Because of the fact that there must different threads that handle the sockets and communication (In-and outgoing), the design of the classes varies in practical.

5 Sources / Used Software

Literature:

“Entwurfsmuster: Elemente wiederverwendbarer ojektorientierter Software” / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides /Verlag: Addison-Wesley / 1. Auflage 1996