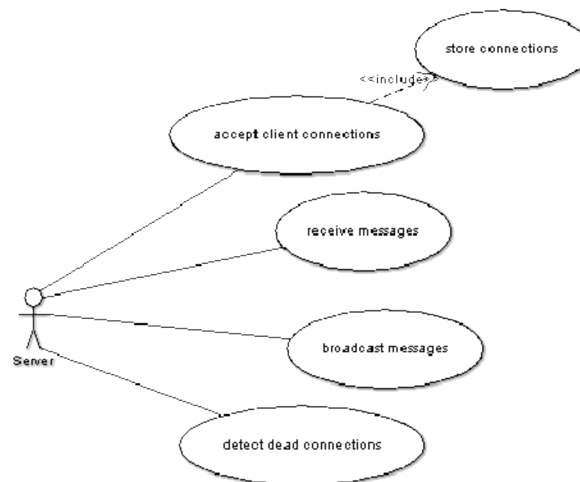
Exercise P1:
Internet Chat System (TCP)
“JChat”

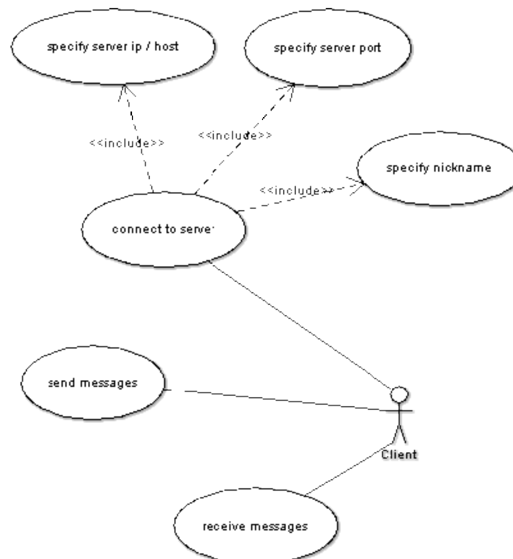
Use Cases

The following use case diagrams show the main functionalities of the Internet Chat System.

Server

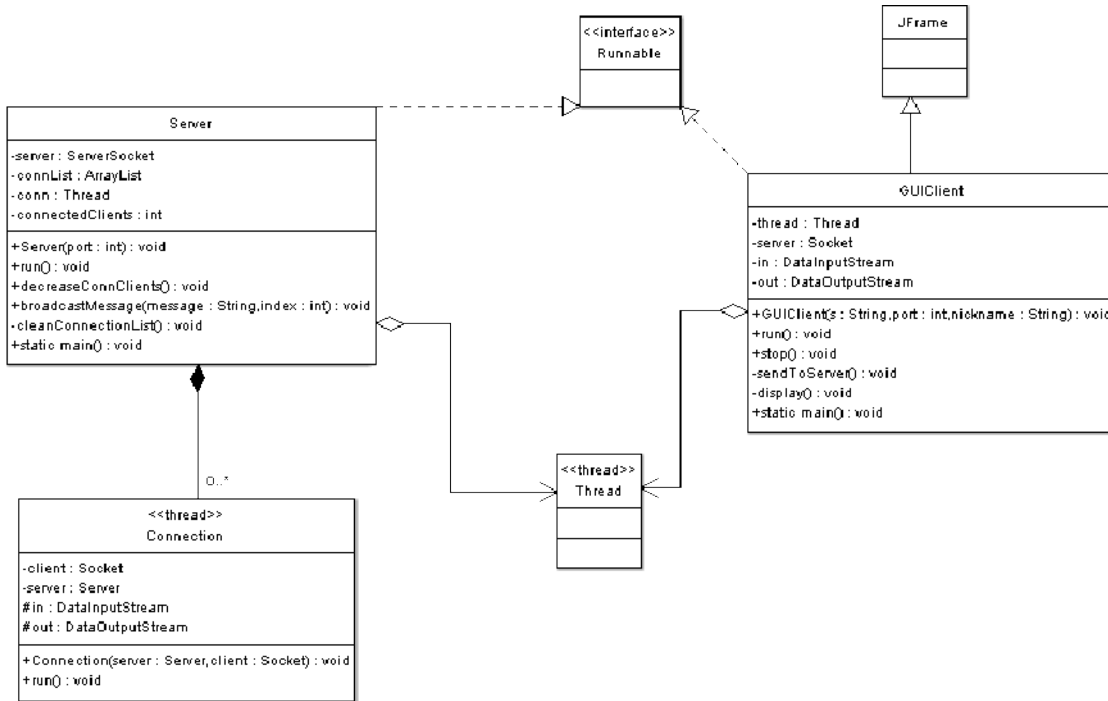


Client



Class Diagram

The structure of the application is shown by the class diagram below.



Class "Server":

The class *Server* disposes of a command-line interface which prompts the user to input the port on which the server has to listen for incoming connections. Furthermore it displays the IP-number of incoming and outgoing clients. The main responsibilities of the class *Server* are just to listen for incoming connections, to accept and to store them. It also has to broadcast the messages to all clients.
To avoid blocking, the server class creates a thread and implements the *Runnable* interface.

```
public class Server implements Runnable{

        private ServerSocket server;
        private ArrayList connList;
        private Thread conn;
        private int connectedClients = 0;
        ...
        public void run() {
                ...
                while(true){
                        Socket client = server.accept();
                        Connection connection = new Connection(this,client);
                        connList.add(connection);
                        ...
                }
                ...
        }

        public void broadcastMessage(String message, int index){
```

```
            Connection client;
            for (int i = 0; i < connList.size
                  client = (Connection)connLi
                  ...
                  client.out.writeUTF(message);
                  ...
            }
      }
}
```

It is possible to access directly the DataOutputStream **out** of the class *Client* because it is declared as protected and lies in the same package as the class *Server*!
This has been done for simplicity reasons!

The parameter index is needed because we wanted to notify the chat users of new incoming users. So we had to use something to avoid that the message of a new incoming client is not sent to that new client itself.
Another feature of the server is the method *cleanConnectionList()* which is used to remove connections which are dead. We created this method because otherwise if a chat-user leaves the chat and the server wants to broadcast the message to this "dead" connection, it gives an error. Therefore this method always checks the array-list of connections each time a message is broadcasted.

```
public class Server implements Runnable {
      ...
      private void cleanConnectionList(){
            Connection client;
            for(int i=0; i<connList.size(); i++){
                  client = (Connection)connList.get(i);
                  if(!client.isAlive()){
                        connList.remove(i);
                  }
            }
      }
      ...
}
```

Class "Connection":

The class *Connection* is responsible for listening for incoming messages and to notify the server to broadcast them to all connected clients. The class creates a *DataInputStream* and a *DataOutputStream* on the client socket. When messages arrive, the class "notifies" the server to broadcast it to all clients. This is done by accessing the *broadcastMessage(…)* method of the server through the local instance variable.
The class gets a *Server* object and the client as a *Socket*. The class itself is a thread which is achieved by directly inheriting from the Java class "Thread".

```
public class Connection extends Thread{
      private Socket client;
      private Server server;
      protected DataInputStream in;
      protected DataOutputStream out;

      public Connection(Server server, Socket client){
            ...
            in = new DataInputStream(client.getInputStream());
            out = new DataOutputStream(client.getOutputStream());
            ...
      }

      public void run(){
```

```
                String line;
                try{
                      while(true){
                            line=in.readUTF();
                            if(line!=null)
                                  server.broadcastMessage(line, -1);
                      }
                } catch (IOException e){
                      ...
                }
          }
    }
```

Class "GUIClient":

The class *GUIClient* disposes of a graphical user interface, done by inheriting from the *JFrame* class. The main components are a *JTextArea* to display the messages from the other clients, a *JTextField* where the user can write his messages and a *JButton* for sending the message to the *Server*. The sending can be done also by directly hitting the return-key.
The structure of the class *GUIClient* is similar to the one of the class *Server*. The class implements the *Runnable* interface and creates a local thread for listening on incoming messages. It also creates a *DataInputStream* and a *DataOutputStream* on the Socket, which is basically the connection to the server. Since the output-field and the input-field of messages are two different ones, there was no need to implement message buffering.
The nickname – handling is not done on the server side, which means also that there is no checking for duplicate nicknames. To visualize the nickname, the messages are simple sent in the form:

$$message = nickname + message$$

```
    public class GUIClient extends JFrame implements Runnable {
          private Thread thread;
          private Socket server;
          private DataInputStream in;
          private DataOutputStream out;
          private String nickname = "";

          public GUIClient(String s, int port, String nickname){
                ...
                server = new Socket(s, port);
                in = new DataInputStream(server.getInputStream());
                out = new DataOutputStream(server.getOutputStream());
                ...
                thread = new Thread(this);
                thread.start();
          }
          ...
    }
```

Since the graphical user interface runs itself on (many) threads, it was not necessary to have two threads, one that listens for incoming messages and one that lets the user contemporaneously write another message without being blocked. Below there is the run method of the thread that listens for incoming messages.

```
          public void run() {
```

```
        String line = "";
        try{
            while(true){
                line = in.readUTF();
                if(line!=null)
                    display(line);
            }
        }catch(...){...}
    }
```

The method below is called to send the message written in the JTextField to the server!

```
    private void sendToServer(){
        if(!jTextFieldUserInput.getText().equals("")){
            try {
                out.writeUTF(nickname.toUpperCase() + " wrote: " +
            jTextFieldUserInput.getText());
            } catch (...) {
                ...
            }
            jTextFieldUserInput.setText("");
        }
    }
```

Implementation problems

One problem that arises is when the connection catches an error when listening on the *DataInputStream*. This means, that the client has terminated the connection to the server. Our strategy was to notify the other clients, that a person has left the chat. This is done by simply broadcasting a new message to all connected clients. Since the message is sent to all connected clients, the server tries to send it also to the terminated client which gives an error. Normally such errors are prevented by calling the *cleanConnectionList()* method, which removes "dead" threads, but since this time the thread itself has invoked the method it appears as alive and won't be removed.

```
public class Connection extends Thread{
    ...
    public void run(){
        String line;
        try{
            while(true){
                line=in.readUTF();
                if(line!=null)
                    server.broadcastMessage(line, -1);
            }
        }catch (IOException e){
            System.out.println(">> Client '" + client.getInetAddress().toString() + "'
has terminated the connection!");
            server.decreaseConnClients();
            server.broadcastMessage("JCHAT SERVER: A user has left the chat!", -1);
        }
    }
    ...
}
```

Since this is only an optional feature, it could simply be removed to avoid the error. Also it isn't a big problem because it doesn't affect the correctness of the program because it is caught by a try-catch statement and it is simply displayed on the console-output of the server:

```
C:\WINDOWS\system32\cmd.exe

D:\Java\Projects\JChat\JChat\bin>java -jar JChatServer.jar
   ************************************************************
   **                  JCHAT SERVER STARTUP                  **
   **                        Welcome!                        **
   ************************************************************

Please enter the port where the server has to listen [Press 'enter' for default]
:
Ok!
The server has been started successfully and is listening now at port: 6789

-------------------------------------------------------------------------------

LOG:

>> incoming client: /192.168.0.4
>> incoming client: /192.168.0.3
>> incoming client: /192.168.0.4
>> Client '/192.168.0.4' has terminated the connection!
Error while broadcasting messages to clients!
Errordescription: java.net.SocketException: Connection reset by peer: socket wri
te error
```

Discuss a possible alternative implementation exploiting the UDP protocol and make a critical comparison between the two.

To program our chat application with the UDP protocol we would have to use the *DatagramSockets* and *DatagramPackets*.

The core part of the chat-client would look like that:

```
...
String m = "This is a test-message";
int port = "6789";
DatagramSocket socket;
try{
        socket = new DatagramSocket();
        byte[] message = m.getBytes();
        //retrieves ip of server given its hostname
        InetAddress host = InetAddress.getByName(<hostname>);
        DatagramPacket request = new DatagramPacket(message, m.length(), host, port);
        socket.send(request);
        byte[] buffer = new byte[1000];
        DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
        socket.receive(reply);
}catch(...){...}
```

The server on the other side would do practically the same as the client concerning the message sending and receiving, only that the server creates a *DatagramSocket* on the port it is listening:

```
...
DatagramSocket socket;
try{
        socket = new DatagramSocket(6789);
        byte[] buffer = new byte[1000];
        while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                socket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                        request.getAddress(), request.getPort());
                socket.send(reply);
        }
```

```
}catch(...){...}
```

The main difference between the two implementations is that in the TCP protocol we have a standing connection between the server and the client. Given this, we can implement, like in our implementation of the chat, functionalities like the detection of “living” or “dead” clients and so on. In the UDP protocol we simply send data packets of a fixed length to the server and wait for a reply, without having a direct connection. Another problem is that UDP doesn’t guarantee ordering as TCP does. Therefore if a user sends more messages consecutively it may be that the messages appear at the other client in a different order or they may also be lost. Such side effects are not acceptable in a chat application and would only cause confusion.