

Projektrapport
Chattapplikation
för Objektorienterad programutveckling, trådar och
datakommunikation

Rasmus Andersson
Emil Sandgren
Erik Sandgren
Jimmy Maksymiw
Lorenz Puskas
Kalle Bornemark

11 mars 2015

Innehåll

1	Arbetsbeskrivning	3
1.1	Rasmus Andersson	3
1.2	Emil Sandgren	3
1.3	Erik Sandgren	3
1.4	Jimmy Maksymiw	3
1.5	Lorenz Puskas	3
1.6	Kalle Bornemark	3
2	Instruktioner för programstart	3
3	Systembeskrivning	3
4	Klassdiagram	4
4.1	Server	4
4.2	Klient	5
5	Kommunikationsdiagram	6
5.1	Connect and login	6
5.2	Client send Message	6
6	Sekvensdiagram	7
6.1	Connect and login	7
6.2	Send message	8
7	Källkod	8
7.1	Server	8
7.1.1	Server.java, Server.ConnectedClient.java	8
7.1.2	Startserver.java	17
7.2	Klient	21
7.2.1	ChatWindow.java	21
7.2.2	Client.java	23
7.2.3	ClientController.java	27
7.2.4	ClientUI.java	30
7.2.5	ImageScaleHandler.java	42
7.2.6	StartClient.java	43
7.3	Delade klasser	47
7.3.1	ChatLog	47
7.3.2	Message	48
7.3.3	User	50
7.3.4	Conversation	51

1 Arbetsbeskrivning

1.1 Rasmus Andersson

Arbetade med kommunikation mellan servern och klienten med Kalle Bornemark, och Jimmy Maksymiw. Formgav projektrapporten samt skrev ImageScaleHandler.java samt Chatlog.java. Jobbade inte med UI-klasserna.

1.2 Emil Sandgren

1.3 Erik Sandgren

Arbetat med generell grundläggande kommunikation mellan server och klient i början. Jobbat sedan med UI och hoppat in lite därefter på det som behövdes. Har ritat upp strukturen mycket och buggfixat.

1.4 Jimmy Maksymiw

1.5 Lorenz Puskas

1.6 Kalle Bornemark

2 Instruktioner för programstart

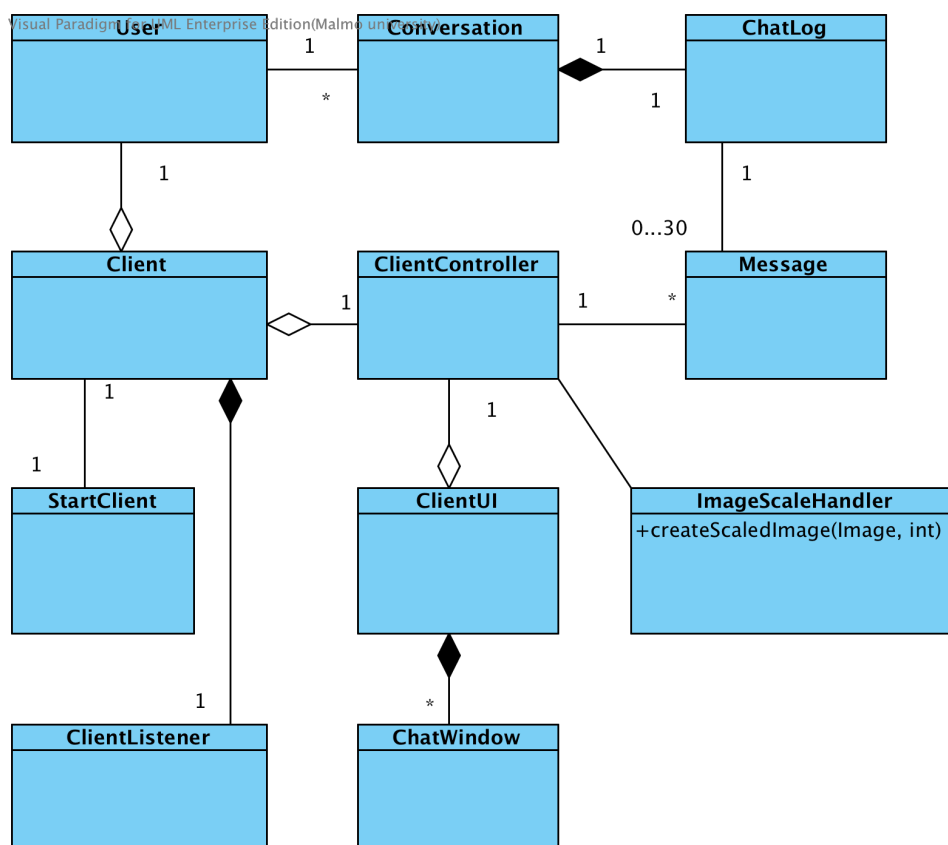
För att köra programmet så krävs det att man startar en server och minst en klient. Main-metoden för att starta servern finns i StartServer.java och main-metoden för att starta klienter finns i StartClient.java. Alla filvägar är relativa till det workspace som används och behöver inte ändras.

3 Systembeskrivning

Vårt system förser en Chatt-tjänst. I systemet finns det klienter och en server. Klienterna har ett grafiskt användargränssnitt som han eller hon kan använda för att skicka meddelanden till alla andra anslutna klienter, enskilda klienter, eller till en grupp av klienter. Meddelanden består av text eller av bilder. Alla dessa meddelanden går via en server som ser till att meddelanden kommer fram till rätt gruppchat eller till lobbyn. Servern lagrar alla textmeddelanden som användarna skickar och loggar även namnet på de bilder som skickas via bildmeddelanden. Det loggas även när användare ansluter eller stänger ner anslutningen mot servern.

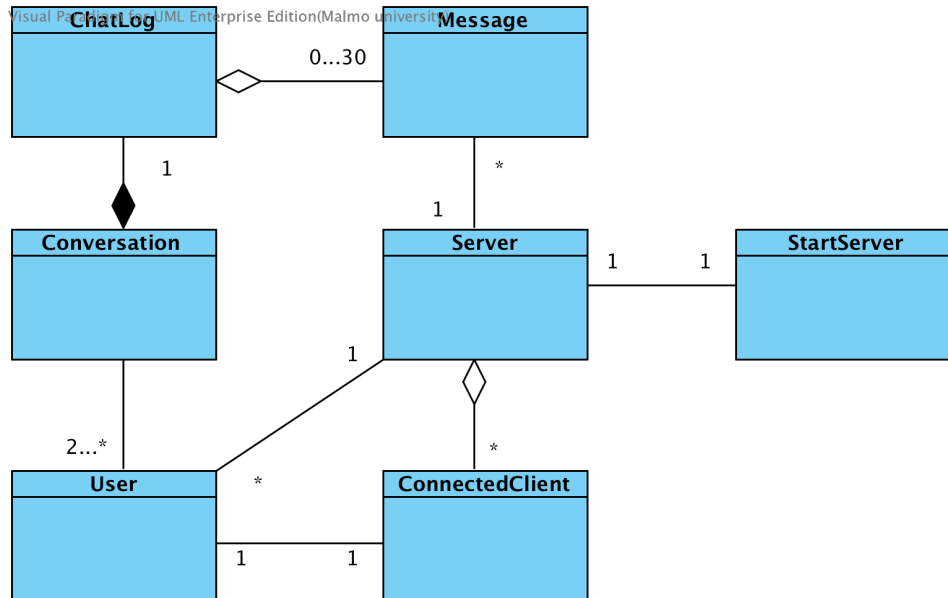
4 Klassdiagram

4.1 Server



Figur 1: Server

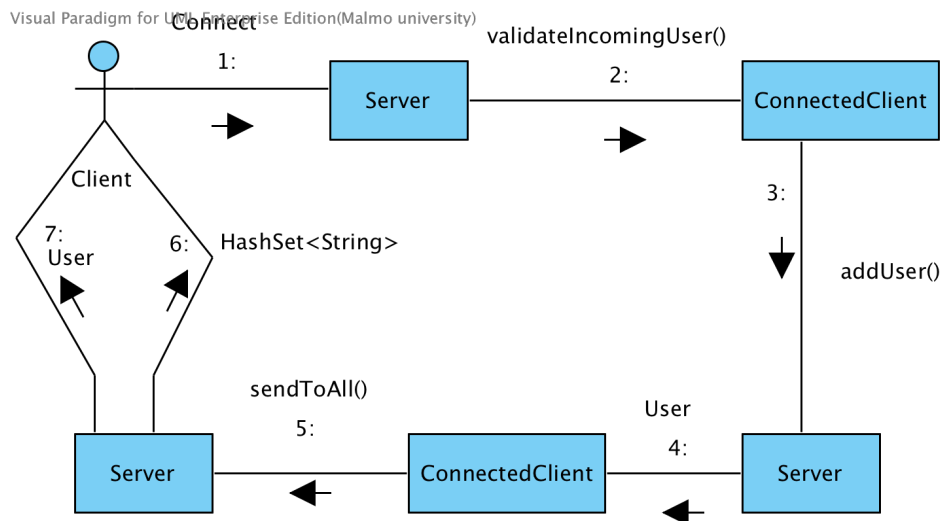
4.2 Klient



Figur 2: Klient

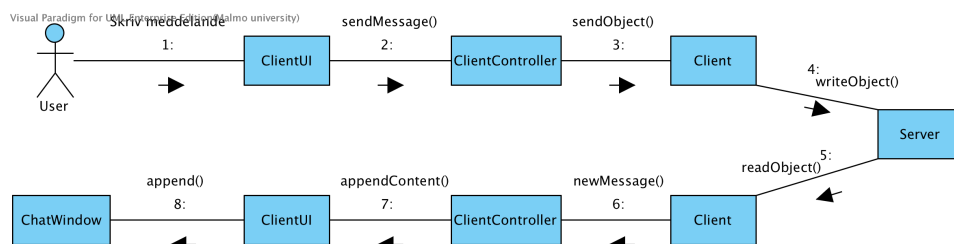
5 Kommunikationsdiagram

5.1 Connect and login



Figur 3: Client connecting and logging in

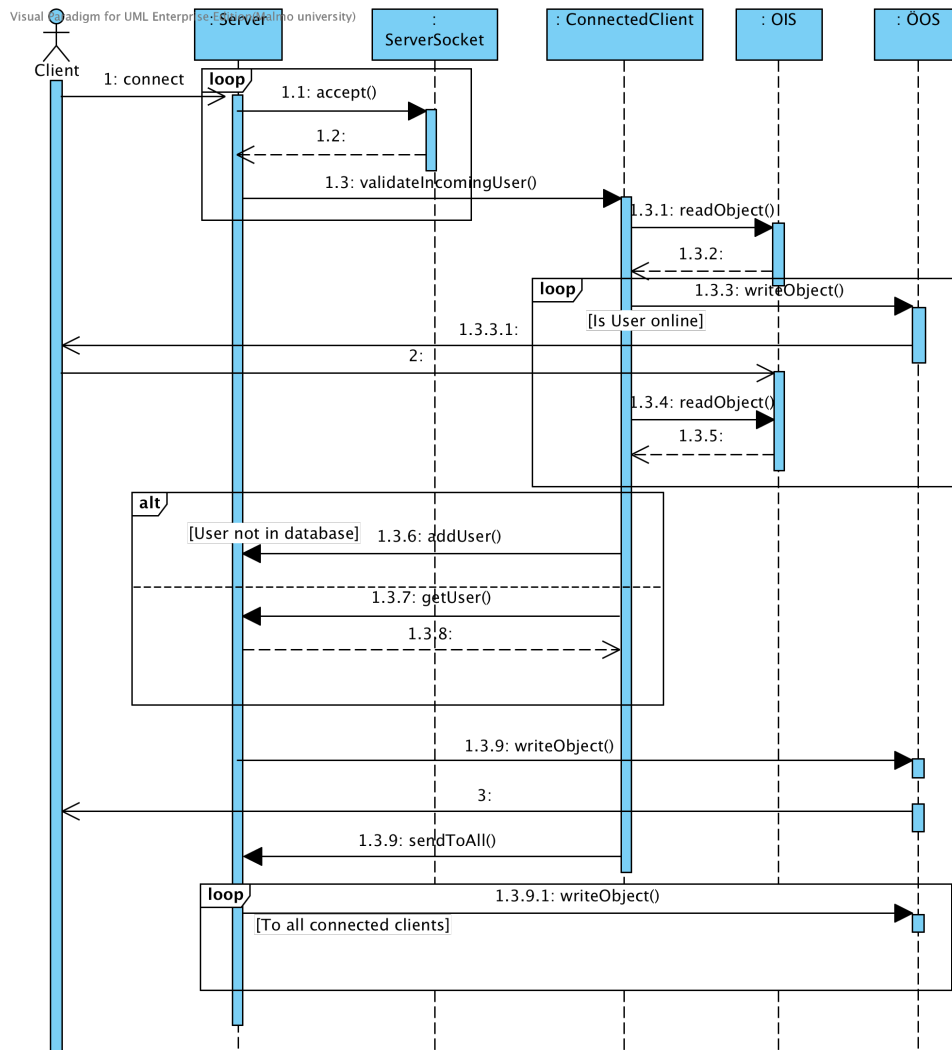
5.2 Client send Message



Figur 4: Client sending a message

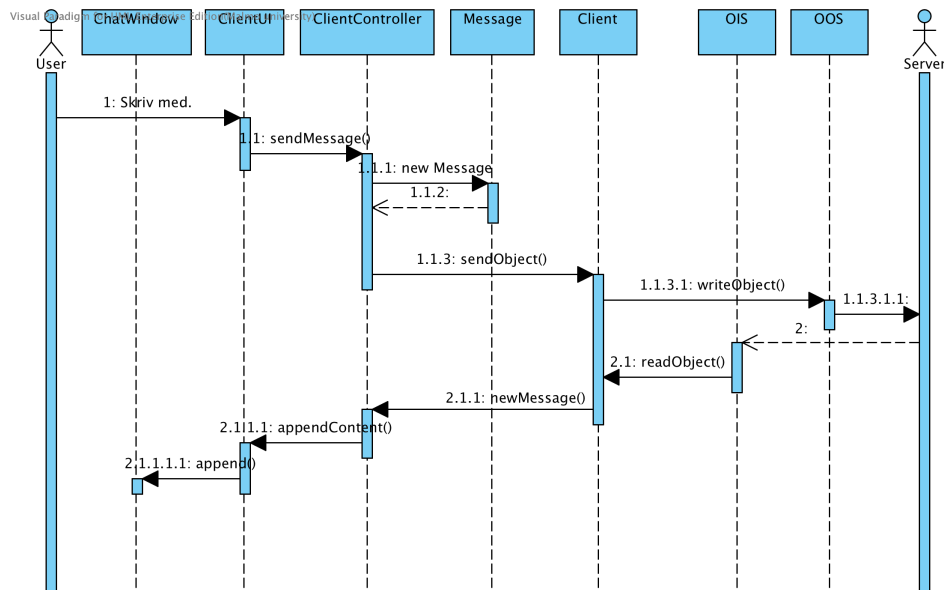
6 Sekvensdiagram

6.1 Connect and login



Figur 5: Client connecting and logging in

6.2 Send message



Figur 6: Client sending a message

7 Källkod

7.1 Server

7.1.1 Server.java, Server.ConnectedClient.java

```

1 package chat;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6 import java.net.ServerSocket;
7 import java.net.Socket;
8 import java.util.ArrayList;
9 import java.util.HashSet;
10 import java.util.logging.*;
11
12 /**
13  * Model class for the server.
14  *
15  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
16  *          Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
17  */
18 public class Server implements Runnable {
19     private ServerSocket serverSocket;
20     private ArrayList<ConnectedClient> connectedClients;
  
```



```
21     private ArrayList<User> registeredUsers;
22     private static final Logger LOGGER = Logger.getLogger(Server
23         .class.getName());
24
25     public Server(int port) {
26         initLogger();
27         registeredUsers = new ArrayList<>();
28         connectedClients = new ArrayList<>();
29         try {
30             serverSocket = new ServerSocket(port);
31             new Thread(this).start();
32         } catch (IOException e) {
33             e.printStackTrace();
34         }
35
36         /**
37          * Initiates the Logger
38          */
39         private void initLogger() {
40             Handler fh;
41             try {
42                 fh = new FileHandler("./src/log/Server.log");
43                 LOGGER.addHandler(fh);
44                 SimpleFormatter formatter = new SimpleFormatter();
45                 fh.setFormatter(formatter);
46                 LOGGER.setLevel(Level.FINE);
47             } catch (IOException e) {}
48         }
49
50         /**
51          * Returns the User which ID matches the given ID.
52          * Returns null if it doesn't exist.
53          *
54          * @param id The ID of the User that is to be found.
55          * @return The matching User object, or null.
56          */
57         public User getUser(String id) {
58             for (User user : registeredUsers) {
59                 if (user.getId().equals(id)) {
60                     return user;
61                 }
62             }
63             return null;
64         }
65
66         /**
67          * Sends an object to all currently connected clients.
68          *
69          * @param object The object to be sent.
70          */
71         public synchronized void sendObjectToAll(Object object) {
72             for (ConnectedClient client : connectedClients) {
73                 client.sendObject(object);
```

```

74     }
75 }
76
77 /**
78  * Checks who the message shall be sent to, then sends it.
79  *
80  * @param message The message to be sent.
81  */
82 public void sendMessage(Message message) {
83     Conversation conversation = null;
84     String to = "";
85
86     // Lobby message
87     if (message.getConversationID() == -1) {
88         sendObjectToAll(message);
89         to += "lobby";
90     } else {
91         User senderUser = null;
92
93         // Finds the sender user
94         for (ConnectedClient cClient : connectedClients) {
95             if (cClient.getUser().getId().equals(message.
96                 getFromUserID())) {
97                 senderUser = cClient.getUser();
98
99                 // Finds the conversation the message shall
100                 // be sent to
101                 for (Conversation con : senderUser.
102                     getConversations()) {
103                     if (con.getId() == message.
104                         getConversationID()) {
105                         conversation = con;
106                         to += conversation.getInvolvedUsers
107                             ().toString();
108
109                         // Finds the message's recipient
110                         // users, then sends the message
111                         for (String s : con.getInvolvedUsers
112                             ()) {
113                             for (ConnectedClient conClient :
114                                 connectedClients) {
115                                 if (conClient.getUser().
116                                     getId().equals(s)) {
117                                     conClient.sendObject(
118                                         message);
119                                 }
120                             }
121                         }
122                     }
123                 }
124                 conversation.addMessage(message);
125             }
126         }
127     }
128 }

```

```
118         LOGGER.info("— NEW MESSAGE SENT —\n" +
119                     "From: " + message.getFromUserID() + "\n" +
120                     "To: " + to + "\n" +
121                     "Message: " + message.getContent().toString());
122     }
123
124     /**
125     * Sends a Conversation object to its involved users
126     *
127     * @param conversation The Conversation object to be sent.
128     */
129     public void sendConversation(Conversation conversation) {
130         HashSet<String> users = conversation.getInvolvedUsers();
131         for (String s : users) {
132             for (ConnectedClient c : connectedClients) {
133                 if (c.getUser().getId().equals(s)) {
134                     c.sendObject(conversation);
135                 }
136             }
137         }
138     }
139
140     /**
141     * Sends an ArrayList with all connected user's IDs.
142     */
143     public void sendConnectedClients() {
144         ArrayList<String> connectedUsers = new ArrayList<>();
145         for (ConnectedClient client : connectedClients) {
146             connectedUsers.add(client.getUser().getId());
147         }
148         sendObjectToAll(connectedUsers);
149     }
150
151     /**
152     * Waits for client to connect.
153     * Creates a new instance of ConnectedClient upon client
154     * connection.
155     * Adds client to list of connected clients.
156     */
157     public void run() {
158         LOGGER.info("Server started.");
159         while (true) {
160             try {
161                 Socket socket = serverSocket.accept();
162                 ConnectedClient client = new ConnectedClient(
163                     socket, this);
164                 connectedClients.add(client);
165             } catch (IOException e) {
166                 e.printStackTrace();
167             }
168         }
169     }
170
171     /**
```

```
170     * Class to handle the communication between server and
171     * connected clients.
172     */
173     private class ConnectedClient implements Runnable {
174         private Thread client = new Thread(this);
175         private ObjectOutputStream oos;
176         private ObjectInputStream ois;
177         private Server server;
178         private User user;
179         private Socket socket;
180
181         public ConnectedClient(Socket socket, Server server) {
182             LOGGER.info("Client connected: " + socket.
183                 getInetAddress());
184             this.socket = socket;
185             this.server = server;
186             try {
187                 oos = new ObjectOutputStream(socket.
188                     getOutputStream());
189                 ois = new ObjectInputStream(socket.
190                     getInputStream());
191             } catch (IOException e) {
192                 e.printStackTrace();
193             }
194             client.start();
195         }
196
197         /**
198          * Returns the connected clients current User.
199          *
200          * @return The connected clients current User
201          */
202         public User getUser() {
203             return user;
204         }
205
206         /**
207          * Sends an object to the client.
208          *
209          * @param object The object to be sent.
210          */
211         public synchronized void sendObject(Object object) {
212             try {
213                 oos.writeObject(object);
214             } catch (IOException e) {
215                 e.printStackTrace();
216             }
217         }
218
219         /**
220          * Removes the user from the list of connected clients.
221          */
222         public void removeConnectedClient() {
223             for (int i = 0; i < connectedClients.size(); i++) {
```

```
220         if (connectedClients.get(i).getUser().getId().
221             equals(this.getUser().getId())) {
222             connectedClients.remove(i);
223             System.out.println("Client removed from
224                                 connectedClients");
225         }
226     }
227 }
228
229 /**
230  * Removes the connected client ,
231  * sends an updated list of connected clients to other
232  * connected clients ,
233  * sends a server message with information of who
234  * disconnected
235  * and closes the client's socket .
236  */
237 public void disconnectClient() {
238     removeConnectedClient();
239     sendConnectedClients();
240     sendObjectToAll("Client disconnected: " + user.getId
241                     ());
242     LOGGER.info("Client disconnected: " + user.getId());
243     try {
244         socket.close();
245     } catch (Exception e) {
246         e.printStackTrace();
247     }
248 }
249
250 /**
251  * Checks if given user exists among already registered
252  * users .
253  *
254  * @return Whether given user already exists or not .
255  */
256 public boolean isUserInDatabase(User user) {
257     for (User u : registeredUsers) {
258         if (u.getId().equals(user.getId())) {
259             return true;
260         }
261     }
262     return false;
263 }
264
265 public User getUser(String ID) {
266     for (User user : registeredUsers) {
267         if (user.getId().equals(ID)) {
268             return user;
269         }
270     }
271     return null;
272 }
```

```
268      /**
269       * Compare given user ID with connected client's IDs and
          check if the user is online.
270       *
271       * @param id User ID to check online status.
272       * @return Whether given user is online or not.
273       */
274     public boolean isUserOnline(String id) {
275         for (ConnectedClient client : connectedClients) {
276
277             if (client.getUser().getId().equals(id) &&
                client != this) {
278                 return true;
279             }
280         }
281         return false;
282     }
283
284     /**
285      * Checks if given set of User IDs already has an open
          conversation.
286      * If it does, it sends the conversation to its
          participants.
287      * If it doesn't, it creates a new conversation, adds it
          to the current users
288      * conversation list, and sends the conversation to its
          participants.
289      *
290      * @param participants A HashSet of user-IDs.
291      */
292     public void updateConversation(HashSet<String>
          participants) {
293         boolean exists = false;
294         Conversation conversation = null;
295         for (Conversation con : user.getConversations()) {
296             if (con.getInvolvedUsers().equals(participants))
                {
297                 conversation = con;
298                 exists = true;
299             }
300         }
301         if (!exists) {
302             conversation = new Conversation(participants);
303             addConversation(conversation);
304         }
305         sendConversation(conversation);
306     }
307
308     /**
309      * Adds given conversation to all its participants' User
          objects.
310      *
311      * @param con The conversation to be added.
312      */
```

```
313     */
314     public void addConversation(Conversation con) {
315         for (User user : registeredUsers) {
316             for (String ID : con.getInvolvedUsers()) {
317                 if (ID.equals(user.getId())) {
318                     user.addConversation(con);
319                 }
320             }
321         }
322     }
323
324     /**
325     * Check if given message is part of an already existing
326     * conversation.
327     *
328     * @param message The message to be checked.
329     * @return Whether given message is part of a
330     *         conversation or not.
331     */
332     public Conversation isPartOfConversation(Message message) {
333         for (Conversation con : user.getConversations()) {
334             if (con.getId() == message.getConversationID()) {
335                 return con;
336             }
337         }
338         return null;
339     }
340
341     /**
342     * Forces connecting users to pick a user that's not
343     * already logged in,
344     * and updates user database if needed.
345     * Announces connected to other connected users.
346     */
347     public void validateIncomingUser() {
348         Object object;
349         try {
350             object = ois.readObject();
351             user = (User) object;
352             while (isUserOnline(user.getId())) {
353                 while (isUserOnline(user.getId())) {
354                     LOG.info("Checking online status for user: "
355                             + user.getId());
356                     LOG.info("User " + user.getId() + "
357                             already connected. Asking for new name.");
358                     sendObject("Client named " + user.getId() + "
359                             already connected, try again!");
360                     // Wait for new user
361                     object = ois.readObject();
362                     user = (User) object;
363                     LOG.info("Checking online status for user
364                             : " + user.getId());
365                 }
366             }
367         } catch (IOException | ClassNotFoundException e) {
368             LOG.error("Error while validating incoming user: "
369                     + e.getMessage());
370         }
371     }
```

```
357         }
358         if (!isUserInDatabase(user)) {
359             registeredUsers.add(user);
360         } else {
361             user = getUser(user.getId());
362         }
363         oos.writeObject(user);
364         server.sendObjectToAll("Client connected: " +
365             user.getId());
366         LOGGER.info("Client connected: " + user.getId());
367         ;
368         sendConnectedClients();
369     } catch (Exception e) {
370         e.printStackTrace();
371     }
372 }
373
374 /**
375  * Listens to incoming Messages, Conversations, HashSets
376  * of User IDs or server messages.
377  */
378 public void startCommunication() {
379     Object object;
380     Message message;
381     try {
382         while (!Thread.interrupted()) {
383             object = ois.readObject();
384             if (object instanceof Message) {
385                 message = (Message) object;
386                 server.sendMessage(message);
387             } else if (object instanceof Conversation) {
388                 Conversation con = (Conversation) object
389                 ;
390                 oos.writeObject(con);
391             } else if (object instanceof HashSet) {
392                 @SuppressWarnings("unchecked")
393                 HashSet<String> participants = (HashSet<
394                     String>) object;
395                 updateConversation(participants);
396             } else {
397                 server.sendObjectToAll(object);
398             }
399         }
400     } catch (IOException e) {
401         disconnectClient();
402         e.printStackTrace();
403     } catch (ClassNotFoundException e2) {
404         e2.printStackTrace();
405     }
406 }
407
408 public void run() {
409     validateIncomingUser();
410     startCommunication();
411 }
```



```
406     }  
407 }  
408 }
```

Listing 1: Server

7.1.2 Startserver.java

```
1 package chat;  
2  
3 import java.awt.BorderLayout;  
4 import java.awt.Color;  
5 import java.awt.Dimension;  
6 import java.awt.FlowLayout;  
7 import java.awt.Font;  
8 import java.awt.GridLayout;  
9 import java.awt.event.ActionEvent;  
10 import java.awt.event.ActionListener;  
11 import java.awt.event.KeyEvent;  
12 import java.awt.event.KeyListener;  
13 import java.net.InetAddress;  
14 import java.net.UnknownHostException;  
15  
16 import javax.swing.JButton;  
17 import javax.swing.JFrame;  
18 import javax.swing.JLabel;  
19 import javax.swing.JOptionPane;  
20 import javax.swing.JPanel;  
21 import javax.swing.JTextField;  
22 import javax.swing.UIManager;  
23 import javax.swing.UnsupportedLookAndFeelException;  
24  
25 /**  
26  * Create an server-panel class.  
27  */  
28 public class StartServer extends JPanel{  
29     private JPanel pnlServerCenterFlow = new JPanel(new  
30         FlowLayout());  
31     private JPanel pnlServerCenterGrid = new JPanel(new  
32         GridLayout(1,2,5,5));  
33     private JPanel pnlServerGrid = new JPanel(new GridLayout  
34         (2,1,5,5));  
35     private JPanel pnlServerRunning = new JPanel(new  
36         BorderLayout());  
37  
38     private JTextField txtServerPort = new JTextField("3450");  
39     private JLabel lblServerPort = new JLabel("Port:");  
40     private JLabel lblServerShowServerIp = new JLabel();  
41     private JLabel lblWelcome = new JLabel("Create a bIRC server  
42         ");  
43     private JLabel lblServerRunning = new JLabel("Server is  
44         running...");
```

```
39     private JButton btnServerCreateServer = new JButton("Create
        Server");
40
41     private Font fontIpPort = new Font("Sans-Serif",Font.PLAIN
        ,17);
42     private Font fontInfo = new Font("Sans-Serif",Font.BOLD|Font
        .ITALIC,20);
43     private Font fontWelcome = new Font("Sans-Serif", Font.BOLD
        ,25);
44     private Font fontButton = new Font("Sans-Serif", Font.BOLD
        ,18);
45     private Server server;
46
47     private BorderLayout br = new BorderLayout();
48
49     public StartServer() {
50         lookAndFeel();
51         initPanels();
52         initLabels();
53         setlblServerShowServerIp();
54         initListeners();
55     }
56
57     /**
58      * Initiate Server-Panels.
59      */
60     public void initPanels() {
61         setPreferredSize(new Dimension(350,150));
62         setOpaque(true);
63         setLayout(br);
64         setBackground(Color.WHITE);
65         add(pnlServerGrid, BorderLayout.CENTER);
66         pnlServerGrid.add(pnlServerCenterGrid);
67         add(lblServerShowServerIp, BorderLayout.SOUTH);
68
69         pnlServerCenterFlow.setOpaque(true);
70         pnlServerCenterFlow.setBackground(Color.WHITE);
71         pnlServerCenterGrid.setOpaque(true);
72         pnlServerCenterGrid.setBackground(Color.WHITE);
73         pnlServerGrid.setOpaque(true);
74         pnlServerGrid.setBackground(Color.WHITE);
75
76         pnlServerCenterGrid.add(lblServerPort);
77         pnlServerCenterGrid.add(txtServerPort);
78         btnServerCreateServer.setFont(fontButton);
79         pnlServerGrid.add(btnServerCreateServer);
80         pnlServerRunning.add(lblServerRunning, BorderLayout
            .CENTER);
81     }
82
83     /**
84      * Initiate Server-Labels.
85      */
86     public void initLabels() {
```

```
87         lblServerPort.setHorizontalAlignment(JLabel.CENTER);
88         lblWelcome.setHorizontalAlignment(JLabel.CENTER);
89         lblServerShowServerIp.setFont(fontInfo);
90         lblServerShowServerIp.setForeground(new Color(146,1,1));
91         lblServerShowServerIp.setHorizontalAlignment(JLabel.
            CENTER);
92         lblServerPort.setFont(fontIpPort);
93         lblServerPort.setOpaque(true);
94         lblServerPort.setBackground(Color.WHITE);
95         lblWelcome.setFont(fontWelcome);
96         add(lblWelcome, BorderLayout.NORTH);
97         txtServerPort.setFont(fontIpPort);
98         lblServerRunning.setFont(fontInfo);
99     }
100
101     /**
102      * Method that shows the user that the server is running.
103      */
104     public void setServerRunning() {
105         remove(br.getLayoutComponent(BorderLayout.CENTER));
106         add(lblServerRunning, BorderLayout.CENTER);
107         lblServerRunning.setHorizontalAlignment(JLabel.CENTER);
108         validate();
109         repaint();
110     }
111
112     /**
113      * Initiate Listeners.
114      */
115     public void initListeners() {
116         CreateStopServerListener create = new
            CreateStopServerListener();
117         EnterListener enter = new EnterListener();
118         btnServerCreateServer.addActionListener(create);
119         txtServerPort.addKeyListener(enter);
120     }
121
122     /**
123      * Sets the ip-label to the local ip of your own computer.
124      */
125     public void setlblServerShowServerIp() {
126         try {
127             String message = ""+ InetAddress.getLocalHost();
128             String realmessage[] = message.split("/");
129             lblServerShowServerIp.setText("Server ip is: " +
                realmessage[1]);
130         } catch (UnknownHostException e) {
131             JOptionPane.showMessageDialog(null, "An error
                occurred.");
132         }
133     }
134
135     /**
136      * Main method for create a server-frame.
```

```
137     * @param args
138     */
139     public static void main(String[] args) {
140         StartServer server = new StartServer();
141         JFrame frame = new JFrame("bIRC Server");
142         frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
143         frame.add(server);
144         frame.pack();
145         frame.setVisible(true);
146         frame.setLocationRelativeTo(null);
147         frame.setResizable(false);
148     }
149
150     /**
151     * Returns the port from the textfield.
152     *
153     * @return Port for creating a server.
154     */
155     public int getPort() {
156         return Integer.parseInt(this.txtServerPort.getText());
157     }
158
159     /**
160     * Set the "Look and Feel".
161     */
162     public void lookAndFeel() {
163         try {
164             UIManager.setLookAndFeel(UIManager.
165                 getSystemLookAndFeelClassName());
166         } catch (ClassNotFoundException e) {
167             e.printStackTrace();
168         } catch (InstantiationException e) {
169             e.printStackTrace();
170         } catch (IllegalAccessException e) {
171             e.printStackTrace();
172         } catch (UnsupportedLookAndFeelException e) {
173             e.printStackTrace();
174         }
175     }
176
177     /**
178     * Listener for create server. Starts a new server with the
179     * port of the textfield.
180     */
181     private class CreateStopServerListener implements
182         ActionListener {
183         public void actionPerformed(ActionEvent e) {
184             if (btnServerCreateServer==e.getSource()) {
185                 server = new Server(getPort());
186                 setServerRunning();
187             }
188         }
189     }
```

```
188      /**
189       * Enter Listener for creating a server.
190       */
191     private class EnterListener implements KeyListener {
192         public void keyPressed(KeyEvent e) {
193             if (e.getKeyCode() == KeyEvent.VK_ENTER) {
194                 server = new Server(getPort());
195                 setServerRunning();
196             }
197         }
198
199         public void keyReleased(KeyEvent arg0) {}
200
201         public void keyTyped(KeyEvent arg0) {}
202     }
203 }
```

Listing 2: StartServer

7.2 Klient

7.2.1 ChatWindow.java

```
1 package chat;
2
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5
6 import javax.swing.*;
7 import javax.swing.text.*;
8
9 /**
10  * Class used to present content in the main window.
11  *
12  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
13  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
14  */
15 public class ChatWindow extends JPanel {
16     private int ID;
17     private JScrollPane scrollPane;
18     private JTextPane textPane;
19
20     private SimpleAttributeSet chatFont = new SimpleAttributeSet
21         ();
22     private SimpleAttributeSet nameFont = new SimpleAttributeSet
23         ();
24
25     /**
26      * Constructor that takes an ID from a Conversation, and
27      * creates a window to display it.
28      *
29      * @param ID The Conversation object's ID.
30      */
31 }
```

```
28     public ChatWindow(int ID) {
29         setLayout(new BorderLayout());
30         this.ID = ID;
31         textPane = new JTextPane();
32         scrollPane = new JScrollPane(textPane);
33
34         scrollPane.setVerticalScrollBarPolicy(JScrollPane.
35             VERTICAL_SCROLLBAR_AS_NEEDED);
36         scrollPane.setHorizontalScrollBarPolicy(JScrollPane.
37             HORIZONTAL_SCROLLBAR_NEVER);
38
39         StyleConstants.setForeground(chatFont, Color.BLACK);
40         StyleConstants.setFontSize(chatFont, 20);
41
42         StyleConstants.setForeground(nameFont, Color.BLACK);
43         StyleConstants.setFontSize(nameFont, 20);
44         StyleConstants.setBold(nameFont, true);
45
46         add(scrollPane, BorderLayout.CENTER);
47         textPane.setEditable(false);
48     }
49
50     /**
51     * Appends a new message into the panel window.
52     * The message can either contain a String or an ImageIcon.
53     *
54     * @param message The message object which content will be
55     * displayed.
56     */
57     public void append(final Message message) {
58         SwingUtilities.invokeLater(new Runnable() {
59             @Override
60             public void run() {
61                 StyledDocument doc = textPane.getStyledDocument
62                     ();
63                 try {
64                     doc.insertString(doc.getLength(), message.
65                         getTimestamp() + " - ", chatFont);
66                     doc.insertString(doc.getLength(), message.
67                         getFromUserID() + ": ", nameFont);
68                     if (message.getContent() instanceof String)
69                     {
70                         doc.insertString(doc.getLength(), (
71                             String)message.getContent(), chatFont
72                         );
73                     } else {
74                         ImageIcon icon = (ImageIcon)message.
75                             getContent();
76                         StyleContext context = new StyleContext
77                             ();
78                         Style labelStyle = context.getStyle(
79                             StyleContext.DEFAULT_STYLE);
80                         JLabel label = new JLabel(icon);
```

```
69         StyleConstants.setComponent(labelStyle ,
70             label);
71         doc.insertString(doc.getLength() , "
72             Ignored" , labelStyle);
73     }
74     doc.insertString(doc.getLength() , "\n" ,
75         chatFont);
76     textPane.setCaretPosition(textPane.
77         getDocument().getLength());
78     } catch (BadLocationException e) {
79         e.printStackTrace();
80     }
81 }
82 /**
83  * Appends a string into the panel window.
84  *
85  * @param stringMessage The string to be appended.
86  */
87 public void append(String stringMessage) {
88     StyledDocument doc = textPane.getStyledDocument();
89     try {
90         doc.insertString(doc.getLength() , "[Server: " +
91             stringMessage + "\n" , chatFont);
92     } catch (BadLocationException e) {
93         e.printStackTrace();
94     }
95 }
96 /**
97  * Returns the ChatWindow's ID.
98  *
99  * @return The ChatWindow's ID.
100 */
101 public int getID() {
102     return ID;
103 }
104 }
```

Listing 3: ChatWindow

7.2.2 Client.java

```
1 package chat;
2
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6 import java.net.Socket;
```

```
7 import java.net.SocketTimeoutException;
8 import java.util.ArrayList;
9
10 import javax.swing.JOptionPane;
11
12 /**
13  * Model class for the client.
14  *
15  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
16  * @author Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
17  */
18
19 public class Client {
20     private Socket socket;
21     private ClientController controller;
22     private ObjectInputStream ois;
23     private ObjectOutputStream oos;
24     private User user;
25     private String name;
26
27     /**
28      * Constructor that creates a new Client with given ip, port
29      * and user name.
30      *
31      * @param ip The IP address to connect to.
32      * @param port Port used in the connection.
33      * @param name The user name to connect with.
34      */
35     public Client(String ip, int port, String name) {
36         this.name = name;
37         try {
38             socket = new Socket(ip, port);
39             ois = new ObjectInputStream(socket.getInputStream());
40             oos = new ObjectOutputStream(socket.getOutputStream());
41             controller = new ClientController(this);
42             new ClientListener().start();
43         } catch (IOException e) {
44             System.err.println(e);
45             if (e.getCause() instanceof SocketTimeoutException) {
46                 //
47             }
48         }
49     }
50
51     /**
52      * Sends an object object to the server.
53      *
54      * @param object The object that should be sent to the
55      * server.
56      */
57     public void sendObject(Object object) {
```



```
56         try {
57             oos.writeObject(object);
58             oos.flush();
59         } catch (IOException e) {}
60     }
61
62     /**
63      * Sets the client user by creating a new User object with
64      * given name.
65      *
66      * @param name The name of the user to be created.
67      */
68     public void setName(String name) {
69         user = new User(name);
70     }
71
72     /**
73      * Returns the clients User object.
74      *
75      * @return The clients User object.
76      */
77     public User getUser() {
78         return user;
79     }
80
81     /**
82      * Closes the clients socket.
83      */
84     public void disconnectClient() {
85         try {
86             socket.close();
87         } catch (Exception e) {}
88     }
89
90     /**
91      * Sends the users conversations to the controller to be
92      * displayed in the UI.
93      */
94     public void initConversations() {
95         for (Conversation con : user.getConversations()) {
96             controller.newConversation(con);
97         }
98     }
99
100     /**
101      * Asks for a username, creates a User object with given
102      * name and sends it to the server.
103      *
104      * The server then either accepts or denies the User object.
105      * If successful, sets the received User object as current
106      * user and announces login in chat.
107      * If not, notifies in chat and requests a new name.
108      */
109     public synchronized void setUser() {
110         Object object = null;
```

```
106     setName(this.name);
107     while (!(object instanceof User)) {
108         try {
109             sendObject(user);
110             object = ois.readObject();
111             if (object instanceof User) {
112                 user = (User) object;
113                 controller.newMessage("You logged in as " +
114                                     user.getId());
115                 initConversations();
116             } else {
117                 controller.newMessage(object);
118                 this.name = JOptionPane.showInputDialog("
119                 Pick a name: ");
120                 setName(this.name);
121             }
122         } catch (IOException e) {
123             e.printStackTrace();
124         } catch (ClassNotFoundException e2) {
125             e2.printStackTrace();
126         }
127     }
128
129     /**
130     * Listens to incoming Messages, user lists, Conversations
131     * or server messages, and deal with them accordingly.
132     */
133     public void startCommunication() {
134         Object object;
135         try {
136             while (!Thread.interrupted()) {
137                 object = ois.readObject();
138                 if (object instanceof Message) {
139                     controller.newMessage(object);
140                 } else if (object instanceof ArrayList) {
141                     ArrayList<String> userList = (ArrayList<
142                     String>) object;
143                     controller.setConnectedUsers(userList);
144                 } else if (object instanceof Conversation) {
145                     Conversation con = (Conversation) object;
146                     user.addConversation(con);
147                     controller.newConversation(con);
148                 } else {
149                     controller.newMessage(object);
150                 }
151             }
152         } catch (IOException e) {
153             e.printStackTrace();
154         } catch (ClassNotFoundException e2) {
155             e2.printStackTrace();
156         }
157     }
158 }
```

```
156
157     /**
158     * Class to handle communication between client and server.
159     */
160     private class ClientListener extends Thread {
161         public void run() {
162             setUser();
163             startCommunication();
164         }
165     }
166 }
```

Listing 4: Client

7.2.3 ClientController.java

```
1 package chat;
2
3 import javax.swing.*;
4 import java.awt.*;
5 import java.awt.image.BufferedImage;
6 import java.util.ArrayList;
7 import java.util.HashSet;
8
9 /**
10  * Controller class to handle system logic between client and
11  * GUI.
12  *
13  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
14  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
15  */
16 public class ClientController {
17     private ClientUI ui = new ClientUI(this);
18     private Client client;
19
20     /**
21     * Creates a new Controller (with given Client).
22     * Also creates a new UI, and displays it in a JFrame.
23     *
24     * @param client
25     */
26     public ClientController(Client client) {
27         this.client = client;
28         SwingUtilities.invokeLater(new Runnable() {
29             public void run() {
30                 JFrame frame = new JFrame("bIRC");
31                 frame.setDefaultCloseOperation(JFrame.
32                     EXIT_ON_CLOSE);
33                 frame.add(ui);
34                 frame.pack();
35                 frame.setLocationRelativeTo(null);
36                 frame.setVisible(true);
37             }
38         });
39     }
40 }
```

```
35         ui.focusTextField();
36     }
37     });
38 }
39
40 /**
41  * Receives an object that's either a Message object or a
42  * String
43  * and sends it to the UI.
44  *
45  * @param object A Message object or a String
46  */
47 public void newMessage(Object object) {
48     if (object instanceof Message) {
49         Message message = (Message) object;
50         ui.appendContent(message);
51     } else {
52         ui.appendServerMessage((String) object);
53     }
54 }
55
56 /**
57  * Returns the current user's ID.
58  *
59  * @return A string containing the current user's ID.
60  */
61 public String getUserID () {
62     return client.getUser().getId();
63 }
64
65 /**
66  * Creates a new message containing given ID and content,
67  * then sends it to the client.
68  *
69  * @param conID Conversation-ID of the message.
70  * @param content The message's content.
71  */
72 public void sendMessage(int conID, Object content) {
73     Message message = new Message(conID, client.getUser().
74         getId(), content);
75     client.sendObject(message);
76 }
77
78 /**
79  * Takes a conversation ID and String with URL to image,
80  * scales the image and sends it to the client.
81  *
82  * @param conID Conversation-ID of the image.
83  * @param url A string containing the URL to the image to be
84  * sent.
85  */
86 public void sendImage(int conID, String url) {
87     ImageIcon icon = new ImageIcon(url);
88     Image img = icon.getImage();
```

```
84         BufferedImage scaledImage = ImageScaleHandler.  
            createScaledImage(img, 250);  
85         icon = new ImageIcon(scaledImage);  
86         sendMessage(conID, icon);  
87     }  
88  
89  
90     /**  
91      * Creates a HashSet of given String array with participants  
            , and sends it to the client.  
92      *  
93      * @param conversationParticipants A string array with  
            conversaion participants.  
94      */  
95     public void sendParticipants(String []  
        conversationParticipants) {  
96         HashSet<String> setParticipants = new HashSet<>();  
97         for (String participant: conversationParticipants) {  
98             setParticipants.add(participant);  
99         }  
100        client.sendObject(setParticipants);  
101    }  
102  
103     /**  
104      * Sends the ArrayList with connected users to the UI.  
105      *  
106      * @param userList The ArrayList with connected users.  
107      */  
108     public void setConnectedUsers(ArrayList<String> userList) {  
109         ui.setConnectedUsers(userList);  
110     }  
111  
112     /**  
113      * Presents a Conversation in the UI.  
114      *  
115      * @param con The Conversation object to be presented in the  
            UI.  
116      */  
117     public void newConversation(Conversation con) {  
118         HashSet<String> users = con.getInvolvedUsers();  
119         String [] usersHashToStringArray = users.toArray(new  
            String [ users.size() ] );  
120         int conID = con.getId();  
121         ui.createConversation(usersHashToStringArray, conID);  
122         for (Message message : con.getConversationLog()) {  
123             ui.appendContent(message);  
124         }  
125     }  
126 }
```

Listing 5: ClientController

7.2.4 ClientUI.java

```
1 package chat ;
2
3 import java . awt . BorderLayout ;
4 import java . awt . Color ;
5 import java . awt . Dimension ;
6 import java . awt . FlowLayout ;
7 import java . awt . Font ;
8 import java . awt . GridLayout ;
9 import java . awt . event . ActionEvent ;
10 import java . awt . event . ActionListener ;
11 import java . awt . event . KeyEvent ;
12 import java . awt . event . KeyListener ;
13 import java . io . File ;
14 import java . util . ArrayList ;
15
16 import javax . swing . ImageIcon ;
17 import javax . swing . JButton ;
18 import javax . swing . JCheckBox ;
19 import javax . swing . JFileChooser ;
20 import javax . swing . JFrame ;
21 import javax . swing . JLabel ;
22 import javax . swing . JOptionPane ;
23 import javax . swing . JPanel ;
24 import javax . swing . JScrollPane ;
25 import javax . swing . JTextField ;
26 import javax . swing . JTextPane ;
27 import javax . swing . UIManager ;
28 import javax . swing . UnsupportedLookAndFeelException ;
29 import javax . swing . text . BadLocationException ;
30 import javax . swing . text . DefaultCaret ;
31 import javax . swing . text . SimpleAttributeSet ;
32 import javax . swing . text . StyleConstants ;
33 import javax . swing . text . StyledDocument ;
34
35 /**
36  * Viewer class to handle the GUI.
37  *
38  * @author Emil Sandgren , Kalle Bornemark , Erik Sandgren ,
39  * Jimmy Maksymiw , Lorenz Puskas & Rasmus Andersson
40  */
41
42 public class ClientUI extends JPanel {
43     private JPanel southPanel = new JPanel() ;
44     private JPanel eastPanel = new JPanel() ;
45     private JPanel eastPanelCenter = new JPanel( new BorderLayout
46         () ) ;
47     private JPanel eastPanelCenterNorth = new JPanel( new
48         FlowLayout() ) ;
49     private JPanel pnlGroupSend = new JPanel( new GridLayout
50         ( 1 , 2 , 8 , 8 ) ) ;
51     private JPanel pnlFileSend = new JPanel( new BorderLayout
52         ( 5 , 5 ) ) ;
```

```
49
50     private String userString = "";
51     private int activeChatWindow = -1;
52     private boolean createdGroup = false;
53
54     private JLabel lblUser = new JLabel();
55     private JButton btnSend = new JButton("Send");
56     private JButton btnNewGroupChat = new JButton();
57     private JButton btnLobby = new JButton("Lobby");
58     private JButton btnCreateGroup = new JButton("");
59     private JButton btnFileChooser = new JButton();
60
61     private JTextPane tpConnectedUsers = new JTextPane();
62     private ChatWindow cwLobby = new ChatWindow(-1);
63     private ClientController clientController;
64     private GroupPanel groupPanel;
65
66     private JTextField tfMessageWindow = new JTextField();
67     private BorderLayout bL = new BorderLayout();
68
69     private JScrollPane scrollConnectedUsers = new JScrollPane(
70         tpConnectedUsers);
71     private JScrollPane scrollChatWindow = new JScrollPane(
72         cwLobby);
73     private JScrollPane scrollGroupRooms = new JScrollPane(
74         eastPanelCenterNorth);
75
76     private JButton[] groupChatList = new JButton[20];
77     private ArrayList<JCheckBox> arrayListCheckBox = new
78         ArrayList<JCheckBox>();
79     private ArrayList<ChatWindow> arrayListChatWindows = new
80         ArrayList<ChatWindow>();
81
82     private Font txtFont = new Font("Sans-Serif", Font.BOLD ,
83         20);
84     private Font fontGroupButton = new Font("Sans-Serif",Font.
85         PLAIN, 12);
86     private Font fontButtons = new Font("Sans-Serif", Font.BOLD
87         ,15);
88     private SimpleAttributeSet chatFont = new SimpleAttributeSet
89         ();
90
91     public ClientUI(ClientController clientController) {
92         this.clientController = clientController;
93         arrayListChatWindows.add(cwLobby);
94         groupPanel = new GroupPanel();
95         groupPanel.start();
96         lookAndFeel();
97         initGraphics();
98         initListeners();
99     }
100
101     /**
102      * Initiates graphics and design.
```

```
94      * Also initiates the panels and buttons.
95      */
96      public void initGraphics() {
97          setLayout(bL);
98          setPreferredSize(new Dimension(900,600));
99          eastPanelCenterNorth.setPreferredSize(new Dimension
100              (130,260));
101          initScroll();
102          initButtons();
103          add(scrollChatWindow, BorderLayout.CENTER);
104          southPanel();
105          eastPanel();
106      }
107
108      /**
109       * Initiates the buttons.
110       * Also sets the icons and the design of the buttons.
111       */
112      public void initButtons() {
113          btnNewGroupChat.setIcon(new ImageIcon("src/resources/
114              newGroup.png"));
115          btnNewGroupChat.setBorder(null);
116          btnNewGroupChat.setPreferredSize(new Dimension(64,64));
117
118          btnFileChooser.setIcon(new ImageIcon("src/resources/
119              newImage.png"));
120          btnFileChooser.setBorder(null);
121          btnFileChooser.setPreferredSize(new Dimension(64, 64));
122
123          btnLobby.setFont(fontButtons);
124          btnLobby.setForeground(new Color(1,48,69));
125          btnLobby.setBackground(new Color(201,201,201));
126          btnLobby.setOpaque(true);
127          btnLobby.setBorderPainted(false);
128
129          btnCreateGroup.setFont(fontButtons);
130          btnCreateGroup.setForeground(new Color(1,48,69));
131      }
132
133      /**
134       * Initiates the scrollpanes and styleconstants.
135       */
136      public void initScroll() {
137          scrollChatWindow.setVerticalScrollBarPolicy(JScrollPane.
138              VERTICAL_SCROLLBAR_AS_NEEDED);
139          scrollChatWindow.setHorizontalScrollBarPolicy(
140              JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
141          scrollConnectedUsers.setVerticalScrollBarPolicy(
142              JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
143          scrollConnectedUsers.setHorizontalScrollBarPolicy(
144              JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
145          DefaultCaret caretConnected = (DefaultCaret)
146              tpConnectedUsers.getCaret();
```



```
139         caretConnected.setUpdatePolicy(DefaultCaret.  
140             ALWAYS_UPDATE);  
141         tpConnectedUsers.setEditable(false);  
142  
143         tfMessageWindow.setFont(txtFont);  
144         StyleConstants.setForeground(chatFont, Color.BLACK);  
145         StyleConstants.setBold(chatFont, true);  
146     }  
147  
148     /**  
149     * Requests that tfMessageWindow gets focus.  
150     */  
151     public void focusTextField() {  
152         tfMessageWindow.requestFocusInWindow();  
153     }  
154  
155     /**  
156     * Initialises listeners.  
157     */  
158     public void initListeners() {  
159         tfMessageWindow.addKeyListener(new EnterListener());  
160         GroupListener groupListener = new GroupListener();  
161         SendListener sendListener = new SendListener();  
162         LobbyListener disconnectListener = new LobbyListener();  
163         btnNewGroupChat.addActionListener(groupListener);  
164         btnCreateGroup.addActionListener(groupListener);  
165         btnLobby.addActionListener(disconnectListener);  
166         btnFileChooser.addActionListener(new FileChooserListener  
167             ());  
168         btnSend.addActionListener(sendListener);  
169     }  
170  
171     /**  
172     * The method takes a ArrayList of the connected users and  
173     * sets the user-checkboxes and  
174     * the connected user textpane based on the users in the  
175     * ArrayList.  
176     *  
177     * @param connectedUsers The ArrayList of the connected  
178     * users.  
179     */  
180     public void setConnectedUsers(ArrayList<String>  
181         connectedUsers) {  
182         setUserText();  
183         tpConnectedUsers.setText("");  
184         updateCheckBoxes(connectedUsers);  
185         for (String ID : connectedUsers) {  
186             appendConnectedUsers(ID);  
187         }  
188     }  
189  
190     /**  
191     * Sets the usertext in the labels to the connected user.  
192     */
```

```
187     public void setUserText() {
188         lblUser.setText(clientController.getUserID());
189         lblUser.setFont(txtFont);
190     }
191
192     /**
193      * The south panel in the ClientUI BorderLayout.SOUTH.
194      */
195     public void southPanel() {
196         southPanel.setLayout(new BorderLayout());
197         southPanel.add(tfMessageWindow, BorderLayout.CENTER);
198         southPanel.setPreferredSize(new Dimension(600, 50));
199
200         btnSend.setPreferredSize(new Dimension(134, 40));
201         btnSend.setFont(fontButtons);
202         btnSend.setForeground(new Color(1, 48, 69));
203         southPanel.add(pnlFileSend, BorderLayout.EAST);
204
205         pnlFileSend.add(btnFileChooser, BorderLayout.WEST);
206         pnlFileSend.add(btnSend, BorderLayout.CENTER);
207
208         add(southPanel, BorderLayout.SOUTH);
209     }
210
211     /**
212      * The east panel in ClientUI BorderLayout.EAST.
213      */
214     public void eastPanel() {
215         eastPanel.setLayout(new BorderLayout());
216         eastPanel.add(lblUser, BorderLayout.NORTH);
217         eastPanel.add(eastPanelCenter, BorderLayout.CENTER);
218         eastPanelCenterNorth.add(pnlGroupSend);
219         eastPanelCenter.add(scrollGroupRooms, BorderLayout.NORTH);
220         eastPanelCenter.add(scrollConnectedUsers, BorderLayout.CENTER);
221
222         pnlGroupSend.add(btnNewGroupChat);
223
224         eastPanel.add(btnLobby, BorderLayout.SOUTH);
225         add(eastPanel, BorderLayout.EAST);
226     }
227
228     /**
229      * Appends the message to the chatwindow object with the ID
230      * of the message object.
231      *
232      * @param message The message object with an ID and a
233      * message.
234      */
235     public void appendContent(Message message) {
236         getChatWindow(message.getConversationID()).append(
237             message);
238         if (activeChatWindow != message.getConversationID()) {
```

```
236         highlightGroup(message.getConversationID());
237     }
238 }
239
240 /**
241  * The method handles notice.
242  *
243  * @param ID The ID of the group.
244  */
245 public void highlightGroup(int ID) {
246     if (ID != -1)
247         groupChatList[ID].setBackground(Color.PINK);
248 }
249
250 /**
251  * Appends the string content in the chatwindow-lobby.
252  *
253  * @param content Is a server message
254  */
255 public void appendServerMessage(String content) {
256     cwLobby.append(content.toString());
257 }
258
259 /**
260  * The method updates the ArrayList of checkboxes and add
261  * the checkboxes to the panel.
262  * Also checks if the ID is your own ID and doesn't add a
263  * checkbox of yourself.
264  * Updates the UI.
265  *
266  * @param checkBoxUserIDs ArrayList of UserID's.
267  */
268 public void updateCheckBoxes(ArrayList<String>
269     checkBoxUserIDs) {
270     arrayListCheckBox.clear();
271     groupPanel.pnlNewGroup.removeAll();
272     for (String ID : checkBoxUserIDs) {
273         if (!ID.equals(clientController.getUserID())) {
274             arrayListCheckBox.add(new JCheckBox(ID));
275         }
276     }
277     for (JCheckBox box : arrayListCheckBox) {
278         groupPanel.pnlNewGroup.add(box);
279     }
280     groupPanel.pnlOuterBorderLayout.revalidate();
281 }
282
283 /**
284  * The method appends the text in the textpane of the
285  * connected users.
286  *
287  * @param message Is a username.
288  */
289 public void appendConnectedUsers(String message){
```

```
286         StyledDocument doc = tpConnectedUsers.getStyledDocument
287         ();
288         try {
289             doc.insertString(doc.getLength(), message + "\n",
290                             chatFont);
291         } catch (BadLocationException e) {
292             e.printStackTrace();
293         }
294     }
295
296     /**
297     * Sets the text on the groupbuttons to the users you check
298     * in the checkbox.
299     * Adds the new group chat connected with a button and a
300     * ChatWindow.
301     * Enables you to change rooms.
302     * Updates UI.
303     *
304     * @param participants String-Array of the participants of
305     * the new groupchat.
306     * @param ID The ID of the participants of the new groupchat
307     */
308     public void createConversation(String[] participants, int ID
309     ) {
310         GroupButtonListener gbListener = new GroupButtonListener
311         ();
312         for (int i = 0; i < participants.length; i++) {
313             if (!(participants[i].equals(clientController.
314                 getUserID()))) {
315                 if (i == participants.length - 1) {
316                     userString += participants[i];
317                 } else {
318                     userString += participants[i] + " ";
319                 }
320             }
321         }
322         if (ID < groupChatList.length && groupChatList[ID] ==
323             null) {
324             groupChatList[ID] = (new JButton(userString));
325             groupChatList[ID].setPreferredSize(new Dimension
326                 (120, 30));
327             groupChatList[ID].setOpaque(true);
328             groupChatList[ID].setBorderPainted(false);
329             groupChatList[ID].setFont(fontGroupButton);
330             groupChatList[ID].setForeground(new Color(93, 0, 0));
331             groupChatList[ID].addActionListener(gbListener);
332
333             eastPanelCenterNorth.add(groupChatList[ID]);
334
335             if (getChatWindow(ID) == null) {
336                 arrayListChatWindows.add(new ChatWindow(ID));
337             }
338         }
339     }
340 }
```

```

329         eastPanelCenterNorth.revalidate();
330         if (createdGroup) {
331             if (activeChatWindow == -1) {
332                 btnLobby.setBackground(null);
333             }
334             else {
335                 groupChatList[activeChatWindow].
336                     setBackground(null);
337             }
338
339             groupChatList[ID].setBackground(new Color
340                 (201,201,201));
341             remove(bL.getLayoutComponent(BorderLayout.CENTER
342                 ));
343             add(getChatWindow(ID), BorderLayout.CENTER);
344             activeChatWindow = ID;
345             validate();
346             repaint();
347             createdGroup = false;
348         }
349     }
350     this.userString = "";
351 }
352
353 /**
354  * Sets the "Look and Feel" of the panels.
355  */
356 public void lookAndFeel() {
357     try {
358         UIManager.setLookAndFeel(UIManager.
359             getSystemLookAndFeelClassName());
360     } catch (ClassNotFoundException e) {
361         e.printStackTrace();
362     } catch (InstantiationException e) {
363         e.printStackTrace();
364     } catch (IllegalAccessException e) {
365         e.printStackTrace();
366     } catch (UnsupportedLookAndFeelException e) {
367         e.printStackTrace();
368     }
369 }
370
371 /**
372  * The method goes through the ArrayList of chatwindow
373  * object and
374  * returns the correct one based on the ID.
375  *
376  * @param ID The ID of the user.
377  * @return ChatWindow A ChatWindow object with the correct
378  * ID.
379  */
380 public ChatWindow getChatWindow(int ID) {
381     for (ChatWindow cw : arrayListChatWindows) {
382         if (cw.getID() == ID) {

```

```
377         return cw;
378     }
379 }
380 return null;
381 }
382
383 /**
384  * The class extends Thread and handles the Create a group
385  * panel.
386  */
387 private class GroupPanel extends Thread {
388     private JFrame groupFrame;
389     private JPanel pnlOuterBorderLayout = new JPanel(new
390         BorderLayout());
391     private JPanel pnlNewGroup = new JPanel();
392     private JScrollPane scrollCheckConnectedUsers = new
393         JScrollPane(pnlNewGroup);
394
395     /**
396      * The metod returns the JFrame groupFrame.
397      *
398      * @return groupFrame
399      */
400     public JFrame getFrame() {
401         return groupFrame;
402     }
403
404     /**
405      * Runs the frames of the groupPanes.
406      */
407     public void run() {
408         panelBuilder();
409         groupFrame = new JFrame();
410         groupFrame.setDefaultCloseOperation(JFrame.
411             DISPOSE_ON_CLOSE);
412         groupFrame.add(pnlOuterBorderLayout);
413         groupFrame.pack();
414         groupFrame.setVisible(false);
415         groupFrame.setLocationRelativeTo(null);
416     }
417
418     /**
419      * Initiates the scrollpanels and the panels of the
420      * groupPanel.
421      */
422     public void panelBuilder() {
423         scrollCheckConnectedUsers.setVerticalScrollBarPolicy
424             (JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
425         scrollCheckConnectedUsers.
426             setHorizontalScrollBarPolicy(JScrollPane.
427                 HORIZONTAL_SCROLLBAR_NEVER);
428         btnCreateGroup.setText("New Conversation");
429         pnlOuterBorderLayout.add(btnCreateGroup,
430             BorderLayout.SOUTH);
431     }
432 }
```

```

422         pnlOuterBorderLayout.add(scrollCheckConnectedUsers,
423                                 BorderLayout.CENTER);
424         scrollCheckConnectedUsers.setPreferredSize(new
425             Dimension(200,500));
426         pnlNewGroup.setLayout(new GridLayout(100,1,5,5));
427     }
428     /**
429     * KeyListener for the messagewindow.
430     * Enables you to send a message with enter.
431     */
432     private class EnterListener implements KeyListener {
433         public void keyPressed(KeyEvent e) {
434             if (e.getKeyCode() == KeyEvent.VK_ENTER && !(
435                 tfMessageWindow.getText().isEmpty())) {
436                 clientController.sendMessage(
437                     activeChatWindow, tfMessageWindow.getText
438                     ());
439                 tfMessageWindow.setText("");
440             }
441         }
442         public void keyReleased(KeyEvent e) {}
443         public void keyTyped(KeyEvent e) {}
444     }
445     /**
446     * Listener that listens to New Group Chat-button and the
447     * Create Group Chat-button.
448     * If create group is pressed, a new button will be created
449     * with the right name,
450     * the right participants.
451     * The method use alot of ArrayLists of checkboxes,
452     * participants and strings.
453     * Also some error-handling with empty buttons.
454     */
455     private class GroupListener implements ActionListener {
456         private ArrayList<String> participants = new ArrayList<
457             String>();
458         private String[] temp;
459         public void actionPerformed(ActionEvent e) {
460             if (btnNewGroupChat == e.getSource() &&
461                 arrayListCheckBox.size() > 0) {
462                 groupPanel.getFrame().setVisible(true);
463             }
464             if (btnCreateGroup == e.getSource()) {
465                 participants.clear();
466                 temp = null;
467                 for(int i = 0; i < arrayListCheckBox.size(); i
468                     ++){
469                     if(arrayListCheckBox.get(i).isSelected()) {

```

```

464         participants.add(arrayListCheckBox.get(i)
465             ).getText());
466     }
467 }
468 temp = new String[participants.size() + 1];
469 temp[0] = clientController.getUserID();
470 for (int i = 1; i <= participants.size(); i++) {
471     temp[i] = participants.get(i-1);
472 }
473 if (temp.length > 1) {
474     clientController.sendParticipants(temp);
475     groupPanel.getFrame().dispose();
476     createdGroup = true;
477 } else {
478     JOptionPane.showMessageDialog(null, "You
479         have to choose atleast one person!");
480 }
481 }
482 }
483 }
484 /**
485  * Listener that connects the right GroupChatButton in an
486  * ArrayList to the right
487  * active chat window.
488  * Updates the UI.
489  */
490 private class GroupButtonListener implements ActionListener
491 {
492     public void actionPerformed(ActionEvent e) {
493         for(int i = 0; i < groupChatList.length; i++) {
494             if(groupChatList[i]==e.getSource()) {
495                 if(activeChatWindow == -1) {
496                     btnLobby.setBackground(null);
497                 }
498                 else {
499                     groupChatList[activeChatWindow].
500                         setBackground(null);
501                 }
502                 groupChatList[i].setBackground(new Color
503                     (201,201,201));
504                 remove(bL.getLayoutComponent(BorderLayout.
505                     CENTER));
506                 add(getChatWindow(i), BorderLayout.CENTER);
507                 activeChatWindow = i;
508                 validate();
509                 repaint();
510             }
511         }
512     }
513 }
514 }
515 /**

```



```
511     * Listener that connects the user with the lobby chatWindow
512     * through the Lobby button.
513     */
514     private class LobbyListener implements ActionListener {
515         public void actionPerformed(ActionEvent e) {
516             if (btnLobby==e.getSource()) {
517                 btnLobby.setBackground(new Color(201,201,201));
518                 if(activeChatWindow != -1)
519                     groupChatList[activeChatWindow].
520                         setBackground(null);
521                 remove(bL.getLayoutComponent(BorderLayout.CENTER));
522                 add(getChatWindow(-1), BorderLayout.CENTER);
523                 activeChatWindow = -1;
524                 invalidate();
525                 repaint();
526             }
527         }
528     }
529     /**
530     * Listener that creates a JFileChooser when the button
531     * btnFileChooser is pressed.
532     * The JFileChooser is for images in the chat and it calls
533     * the method sendImage in the controller.
534     */
535     private class FileChooserListener implements ActionListener
536     {
537         public void actionPerformed(ActionEvent e) {
538             if (btnFileChooser==e.getSource()) {
539                 JFileChooser fileChooser = new JFileChooser();
540                 int returnValue = fileChooser.showOpenDialog(
541                     null);
542                 if (returnValue == JFileChooser.APPROVE_OPTION)
543                 {
544                     File selectedFile = fileChooser.
545                         getSelectedFile();
546                     String fullPath = selectedFile.
547                         getAbsolutePath();
548                     clientController.sendImage(activeChatWindow,
549                         fullPath);
550                 }
551             }
552         }
553     }
554     /**
555     * Listener for the send message button.
556     * Resets the message textfield text.
557     */
558     private class SendListener implements ActionListener {
559         public void actionPerformed(ActionEvent e) {
```

```
553         if (btnSend==e.getSource() && !(tfMessageWindow.  
554             getText().isEmpty())) {  
555                 clientController.sendMessage(  
556                     activeChatWindow, tfMessageWindow.getText  
557                     ());  
558                 tfMessageWindow.setText("");  
559             }  
        }  
    }
```

Listing 6: ClientUI

7.2.5 ImageScaleHandler.java

```
1 package chat;  
2  
3 import java.awt.Graphics2D;  
4 import java.awt.Image;  
5 import java.awt.image.BufferedImage;  
6  
7 import javax.swing.ImageIcon;  
8 import javax.swing.JFrame;  
9 import javax.swing.JLabel;  
10 import javax.swing.JPanel;  
11  
12 import org.imgscalr.Scalr;  
13 import org.imgscalr.Scalr.Method;  
14  
15 /**  
16  * Scales down images to preferred size.  
17  *  
18  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,  
19  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson  
20  */  
21 public class ImageScaleHandler {  
22  
23     private static BufferedImage toBufferedImage(Image img) {  
24         if (img instanceof BufferedImage) {  
25             return (BufferedImage) img;  
26         }  
27         BufferedImage bimage = new BufferedImage(img.getWidth(  
28             null),  
29             img.getHeight(null), BufferedImage.TYPE_INT_ARGB  
30             );  
31         Graphics2D bGr = bimage.createGraphics();  
32         bGr.drawImage(img, 0, 0, null);  
33         bGr.dispose();  
34         return bimage;  
35     }  
36  
37     public static BufferedImage createScaledImage(Image img, int  
38         height) {
```

```
36         BufferedImage bimage = toBufferedImage(img);
37         bimage = Scalr.resize(bimage, Method.ULTRA_QUALITY,
38                               Scalr.Mode.FIT_TO_HEIGHT, 0, height);
39         return bimage;
40     }
41
42     // Example
43     public static void main(String[] args) {
44         ImageIcon icon = new ImageIcon("src/filer/new1.jpg");
45         Image img = icon.getImage();
46
47         // Use this to scale images
48         BufferedImage scaledImage = ImageScaleHandler.
49             createScaledImage(img, 75);
50         icon = new ImageIcon(scaledImage);
51
52         JLabel lbl = new JLabel();
53         lbl.setIcon(icon);
54         JPanel panel = new JPanel();
55         panel.add(lbl);
56         JFrame frame = new JFrame();
57         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
58         frame.add(panel);
59         frame.pack();
60         frame.setVisible(true);
61     }
```

Listing 7: ImageScaleHandler

7.2.6 StartClient.java

```
1 package chat;
2
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import java.awt.Dimension;
6 import java.awt.FlowLayout;
7 import java.awt.Font;
8 import java.awt.GridLayout;
9 import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11
12 import javax.swing.*;
13
14 /**
15  * Log in UI and start-class for the chat.
16  *
17  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
18  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson.
19  */
20 public class StartClient extends JPanel {
```

```
21     private JLabel lblIp = new JLabel("IP:");
22     private JLabel lblPort = new JLabel("Port:");
23     private JLabel lblWelcomeText = new JLabel("Log in to bIRC")
24     ;
25     private JLabel lblUserName = new JLabel("Username:");
26
27     private JTextField txtIp = new JTextField("localhost");
28     private JTextField txtPort = new JTextField("3450");
29     private JTextField txtUserName = new JTextField();
30
31     private JButton btnLogIn = new JButton("Login");
32     private JButton btnCancel = new JButton("Cancel");
33
34     private Font fontWelcome = new Font("Sans-Serif",Font.BOLD
35     ,25);
36     private Font fontIpPort = new Font("Sans-Serif",Font.PLAIN
37     ,17);
38     private Font fontButtons = new Font("Sans-Serif",Font.BOLD
39     ,15);
40     private Font fontUserName = new Font("Sans-Serif",Font.BOLD
41     ,17);
42
43     private JPanel pnlCenterGrid = new JPanel(new GridLayout
44     (3,2,5,5));
45     private JPanel pnlCenterFlow = new JPanel(new FlowLayout());
46     private JPanel pnlNorthGrid = new JPanel(new GridLayout
47     (2,1,5,5));
48     private JPanel pnlNorthGridGrid = new JPanel(new GridLayout
49     (1,2,5,5));
50
51     private JFrame frame;
52
53     public StartClient() {
54         setLayout(new BorderLayout());
55         initPanels();
56         lookAndFeel();
57         initGraphics();
58         initButtons();
59         initListeners();
60         frame = new JFrame("bIRC Login");
61         frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
62         frame.add(this);
63         frame.pack();
64         frame.setVisible(true);
65         frame.setLocationRelativeTo(null);
66         frame.setResizable(false);
67     }
68
69     /**
70      * Initiates the listeners.
71      */
72     public void initListeners() {
73         LogInMenuListener log = new LogInMenuListener();
74         btnLogIn.addActionListener(log);
```

```
67         txtUserName.addActionListener(new EnterListener());
68         btnCancel.addActionListener(log);
69     }
70
71     /**
72      * Initiates the panels.
73      */
74     public void initPanels() {
75         setPreferredSize(new Dimension(400, 180));
76         pnlCenterGrid.setBounds(100, 200, 200, 50);
77         add(pnlCenterFlow, BorderLayout.CENTER);
78         pnlCenterFlow.add(pnlCenterGrid);
79
80         add(pnlNorthGrid, BorderLayout.NORTH);
81         pnlNorthGrid.add(lblWelcomeText);
82         pnlNorthGrid.add(pnlNorthGridGrid);
83         pnlNorthGridGrid.add(lblUserName);
84         pnlNorthGridGrid.add(txtUserName);
85
86         lblUserName.setHorizontalAlignment(JLabel.CENTER);
87         lblUserName.setFont(fontIpPort);
88         lblWelcomeText.setHorizontalAlignment(JLabel.CENTER);
89         lblWelcomeText.setFont(fontWelcome);
90         lblIp.setFont(fontIpPort);
91         lblPort.setFont(fontIpPort);
92     }
93
94     /**
95      * Initiates the buttons.
96      */
97     public void initButtons() {
98         btnCancel.setFont(fontButtons);
99         btnLogIn.setFont(fontButtons);
100
101         pnlCenterGrid.add(lblIp);
102         pnlCenterGrid.add(txtIp);
103         pnlCenterGrid.add(lblPort);
104         pnlCenterGrid.add(txtPort);
105         pnlCenterGrid.add(btnLogIn);
106         pnlCenterGrid.add(btnCancel);
107     }
108
109     /**
110      * Initiates the graphics and some design.
111      */
112     public void initGraphics() {
113         pnlCenterGrid.setOpaque(false);
114         pnlCenterFlow.setOpaque(false);
115         pnlNorthGridGrid.setOpaque(false);
116         pnlNorthGrid.setOpaque(false);
117         setBackground(Color.WHITE);
118         lblUserName.setBackground(Color.WHITE);
119         lblUserName.setOpaque(false);
120     }
```

```
121         btnLogIn.setForeground(new Color(41,1,129));
122         btnCancel.setForeground(new Color(41,1,129));
123
124         txtIp.setFont(fontIpPort);
125         txtPort.setFont(fontIpPort);
126         txtUserName.setFont(fontUserName);
127     }
128
129     /**
130      * Sets the "Look and Feel" of the JComponents.
131      */
132     public void lookAndFeel() {
133         try {
134             UIManager.setLookAndFeel(UIManager.
135                                     getSystemLookAndFeelClassName());
136         } catch (ClassNotFoundException e) {
137             e.printStackTrace();
138         } catch (InstantiationException e) {
139             e.printStackTrace();
140         } catch (IllegalAccessException e) {
141             e.printStackTrace();
142         } catch (UnsupportedLookAndFeelException e) {
143             e.printStackTrace();
144         }
145     }
146
147     /**
148      * Main method for the login-frame.
149      */
150     public static void main(String[] args) {
151         SwingUtilities.invokeLater(new Runnable() {
152             @Override
153             public void run() {
154                 StartClient ui = new StartClient();
155             }
156         });
157     }
158
159     /**
160      * Listener for login-button, create server-button and for
161      * the cancel-button.
162      * Also limits the username to a 10 char max.
163      */
164     private class LogInMenuListener implements ActionListener {
165         public void actionPerformed(ActionEvent e) {
166             if (btnLogIn==e.getSource()) {
167                 if (txtUserName.getText().length() <= 10) {
168                     new Client(txtIp.getText(), Integer.
169                             parseInt(txtPort.getText()),
170                             txtUserName.getText());
171                 } else {
172                     JOptionPane.showMessageDialog(null, "Namnet
173                                                     får max vara 10 karaktärer!");
174                 }
175             }
176         }
177     }
```

```
170         txtUserName.setText("");
171     }
172 }
173 if (btnCancel==e.getSource()) {
174     System.exit(0);
175 }
176 }
177 }
178
179 /**
180  * Listener for the textField. Enables you to press enter
181  * instead of login.
182  * Also limits the username to 10 chars.
183  */
184 private class EnterListener implements ActionListener {
185     public void actionPerformed(ActionEvent e) {
186         if(txtUserName.getText().length() <= 10) {
187             new Client(txtIp.getText(), Integer.parseInt(
188                 txtPort.getText()),txtUserName.getText());
189             frame.dispose();
190         } else {
191             JOptionPane.showMessageDialog(null, "Namnet får
192                 max vara 10 karaktärer!");
193             txtUserName.setText("");
194         }
195     }
196 }
```

Listing 8: LoginUI

7.3 Delade klasser

7.3.1 ChatLog

```
1 package chat;
2 import java.io.Serializable;
3 import java.util.Iterator;
4 import java.util.LinkedList;
5
6 /**
7  * Class to hold logged messages.
8  *
9  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
10  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
11  */
12
13 public class ChatLog implements Iterable<Message>, Serializable
14 {
15     private LinkedList<Message> list = new LinkedList<Message>()
16     ;
17     private static int MESSAGE_LIMIT = 30;
```

```
16     private static final long serialVersionUID =
17         13371337133732526L;
18
19     /**
20      * Adds a new message to the chat log.
21      *
22      * @param message The message to be added.
23      */
24     public void add(Message message) {
25         if (list.size() >= MESSAGE_LIMIT) {
26             list.removeLast();
27         }
28         list.add(message);
29     }
30
31     public Iterator<Message> iterator() {
32         return list.iterator();
33     }
34 }
```

Listing 9: ChatLog

7.3.2 Message

```
1 package chat;
2
3 import java.io.Serializable;
4 import java.text.SimpleDateFormat;
5 import java.util.Date;
6
7 /**
8  * Model class to handle messages
9  *
10  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
11  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
12  */
13 public class Message implements Serializable {
14     private String fromUserID;
15     private Object content;
16     private String timestamp;
17     private int conversationID = -1;    /* -1 means it's a lobby
18         message */
19     private static final long serialVersionUID = 133713371337L;
20
21     /**
22      * Constructor that creates a new message with given
23      * conversation ID, String with information who sent it,
24      * and its content.
25      *
26      * @param conversationID The conversation ID.
27      * @param fromUserID A string with information who sent the
28      * message.
29      */
30 }
```



```
25     * @param content The message's content.
26     */
27     public Message(int conversationID, String fromUserID, Object
        content) {
28         this.conversationID = conversationID;
29         this.fromUserID = fromUserID;
30         this.content = content;
31         newTime();
32     }
33
34     /**
35     * Creates a new timestamp for the message.
36     */
37     private void newTime() {
38         Date time = new Date();
39         SimpleDateFormat ft = new SimpleDateFormat("HH:mm:ss");
40         this.timestamp = ft.format(time);
41     }
42
43     /**
44     * Returns a string containing sender ID.
45     *
46     * @return A string with the sender ID.
47     */
48     public String getFromUserID() {
49         return fromUserID;
50     }
51
52     /**
53     * Returns an int with the conversation ID.
54     *
55     * @return An int with the conversation ID.
56     */
57     public int getConversationID() {
58         return conversationID;
59     }
60
61     /**
62     * Returns the message's timestamp.
63     *
64     * @return The message's timestamp.
65     */
66     public String getTimestamp() {
67         return this.timestamp;
68     }
69
70     /**
71     * Returns the message's content.
72     *
73     * @return The message's content.
74     */
75     public Object getContent() {
76         return content;
77     }
```

78 }

Listing 10: Message

7.3.3 User

```
1 package chat;
2
3 import java.io.Serializable;
4 import java.util.ArrayList;
5
6 /**
7  * Class to hold information of a user.
8  *
9  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
10 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
11 */
12 public class User implements Serializable {
13     private static final long serialVersionUID = 1273274782824L;
14     private ArrayList<Conversation> conversations;
15     private String id;
16
17     /**
18      * Constructor to create a User with given ID.
19      *
20      * @param id A string with the user ID.
21      */
22     public User(String id) {
23         this.id = id;
24         conversations = new ArrayList<>();
25     }
26
27     /**
28      * Returns an ArrayList with the user's conversations
29      *
30      * @return The user's conversations.
31      */
32     public ArrayList<Conversation> getConversations() {
33         return conversations;
34     }
35
36     /**
37      * Adds a new conversation to the user.
38      *
39      * @param conversation The conversation to be added.
40      */
41     public void addConversation(Conversation conversation) {
42         conversations.add(conversation);
43     }
44
45     /**
46      * Returns the user's ID.
```

```
47     *  
48     * @return The user's ID.  
49     */  
50     public String getId() {  
51         return id;  
52     }  
53 }
```

Listing 11: User

7.3.4 Conversation

```
1 package chat;  
2  
3 import java.io.Serializable;  
4 import java.util.HashSet;  
5  
6 /**  
7  * Class to hold information of a conversation.  
8  *  
9  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,  
10 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson  
11 */  
12 public class Conversation implements Serializable {  
13     private HashSet<String> involvedUsers;  
14     private ChatLog conversationLog;  
15     private int id;  
16     private static int numberOfConversations = 0;  
17  
18     /**  
19     * Constructor that takes a HashSet of involved users.  
20     *  
21     * @param involvedUsersID The user ID's to be added to the  
22     * conversation.  
23     */  
24     public Conversation(HashSet<String> involvedUsersID) {  
25         this.involvedUsers = involvedUsersID;  
26         this.conversationLog = new ChatLog();  
27         id = ++numberOfConversations;  
28     }  
29  
30     /**  
31     * Returns a HashSet of the conversation's involved users.  
32     *  
33     * @return A hashSet of the conversation's involved users.  
34     */  
35     public HashSet<String> getInvolvedUsers() {  
36         return involvedUsers;  
37     }  
38  
39     /**  
40     * Returns the conversion's ChatLog.
```

```
40      *
41      * @return The conversation's ChatLog.
42      */
43      public ChatLog getConversationLog() {
44          return conversationLog;
45      }
46
47      /**
48       * Adds a message to the conversation.
49       *
50       * @param message The message to be added.
51       */
52      public void addMessage(Message message) {
53          conversationLog.add(message);
54      }
55
56
57      /**
58       * Return the conversation's ID.
59       *
60       * @return The conversation's ID.
61       */
62      public int getId() {
63          return id;
64      }
65
66  }
```

Listing 12: Conversation