Projektrapport

Chattapplikation
Grupp 13
för Objektorienterad programutveckling, trådar och datakommunikation

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

18 mars 2015

Innehåll

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

Arbetade med UI klasserna ClientUI, StartClient och StartServer och ChatWindow. Huvudansvarig för UI. Jobbat med att koppla ihop UI:t med vad som kommer in från servern.

1.3 Erik Sandgren

Arbetade först med generell grundläggande kommunikation mellan server och klient. Jobbade sedan med UI och har även hoppat in där det behövdes på andra delar av systemet. Har ritat upp mycket av strukturen och fixat buggar.

1.4 Jimmy Maksymiw

Arbetade med planering av och struktur på hur chatten ska fungera. Vid programmeringen har han arbetat med logiken som används i både klient och server. Hur kommunikationen skall ske och vad som ska göras på de olika sidorna. Har också varit med och gjort diagrammen.

1.5 Lorenz Puskas

Arbetade främst med att designa ClientUI tillsammans med Emil.

1.6 Kalle Bornemark

Arbetade med server/klient-kommunikation, projektplanering och klasstrukturen. Skapade även diagrammen och har fungerat som projektledare till och från.

2 Instruktioner för programstart

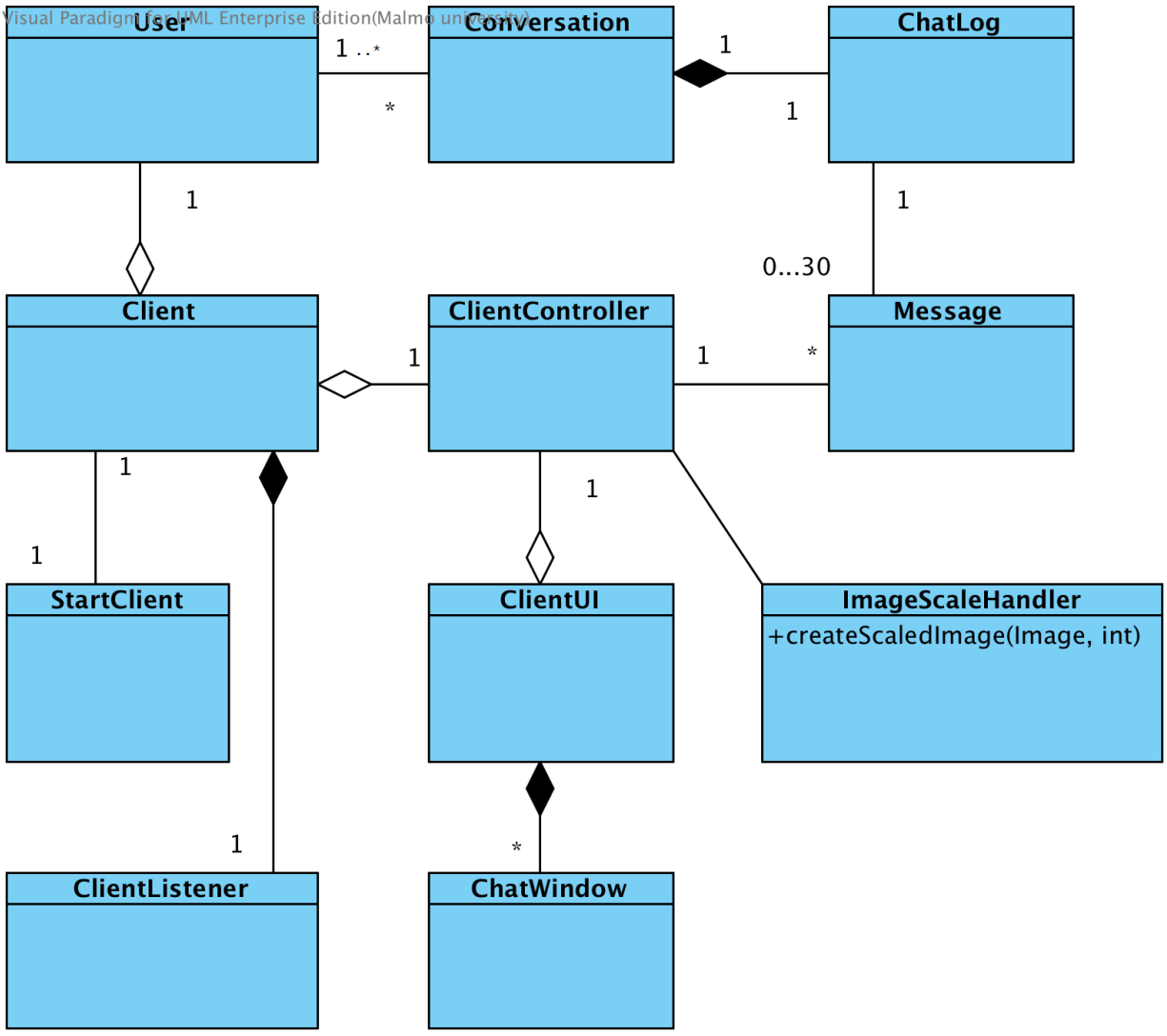
För att köra programmet krävs att man startar en server och minst en klient. Main-metoden för att starta servern finns i StartServer.java och main-metoden för att starta klienter finns i StartClient.java. Alla filvägar som används är relativa projektets workspace och behöver inte ändras.

3 Systembeskrivning

Systemet förser en Chatt-tjänst. I systemet finns det flera klienter och en server. Klienterna har ett grafiskt användargränssnitt 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 textmeddelande som användarna skickar och loggar även namn på de bilder som skickas via bildmeddelanden. Servern loggar även användarnamn för de klienter som ansluter och när dessa stänger ner anslutningen mot servern.

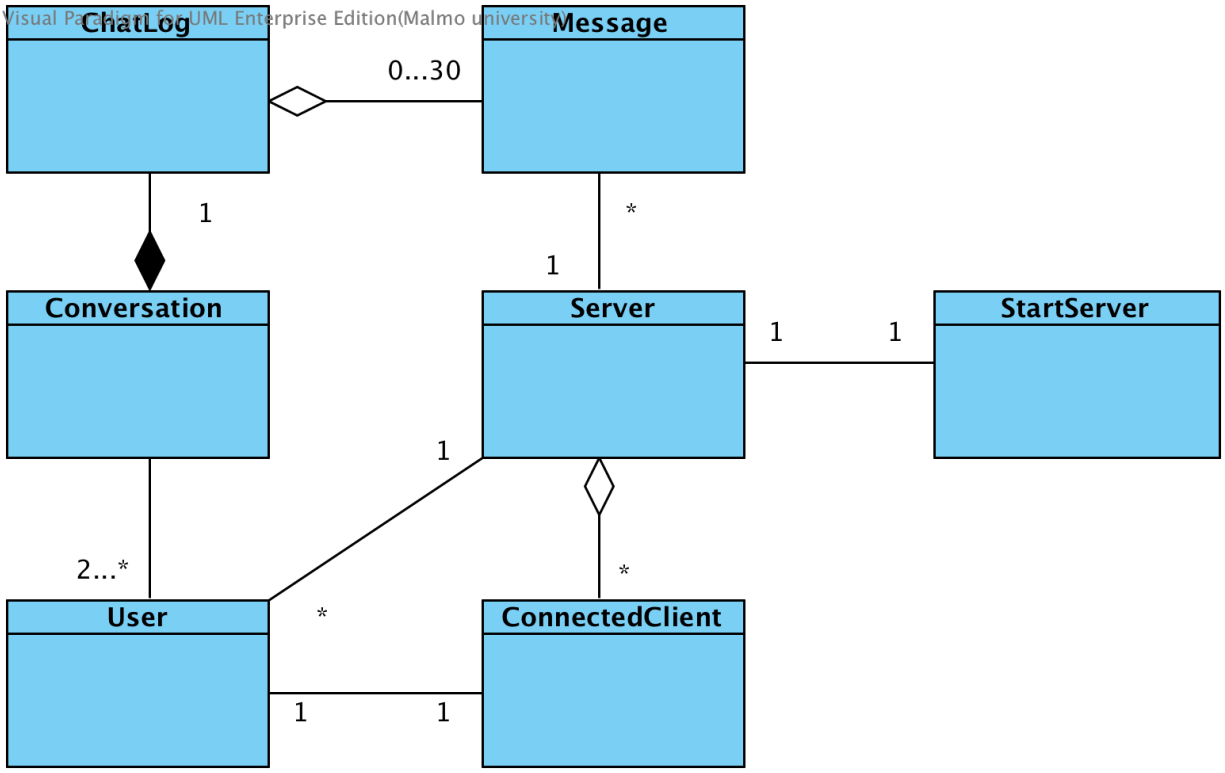
4 Klassdiagram

4.1 Klient



Figur 1: Klient

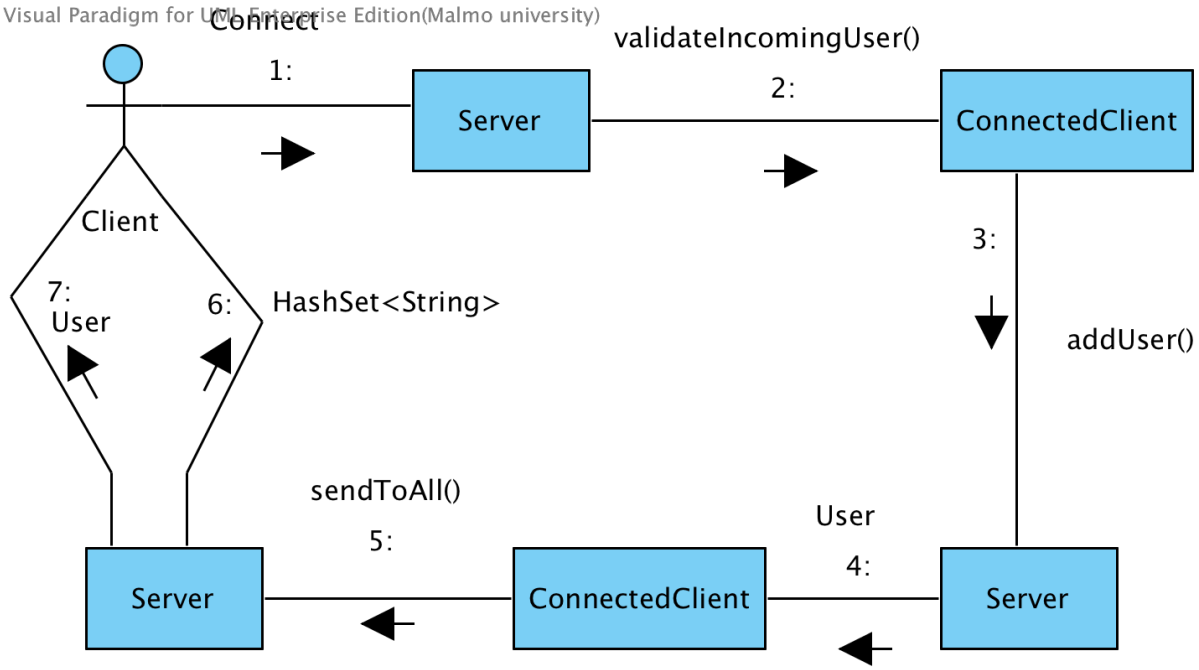
4.2 Server



Figur 2: Server

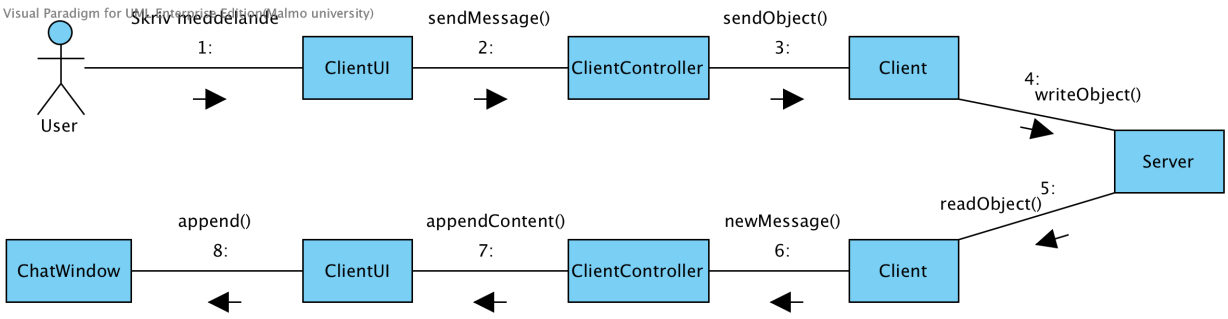
5 Kommunikationsdiagram

5.1 Connect and login



Figur 3: Client connecting and logging in

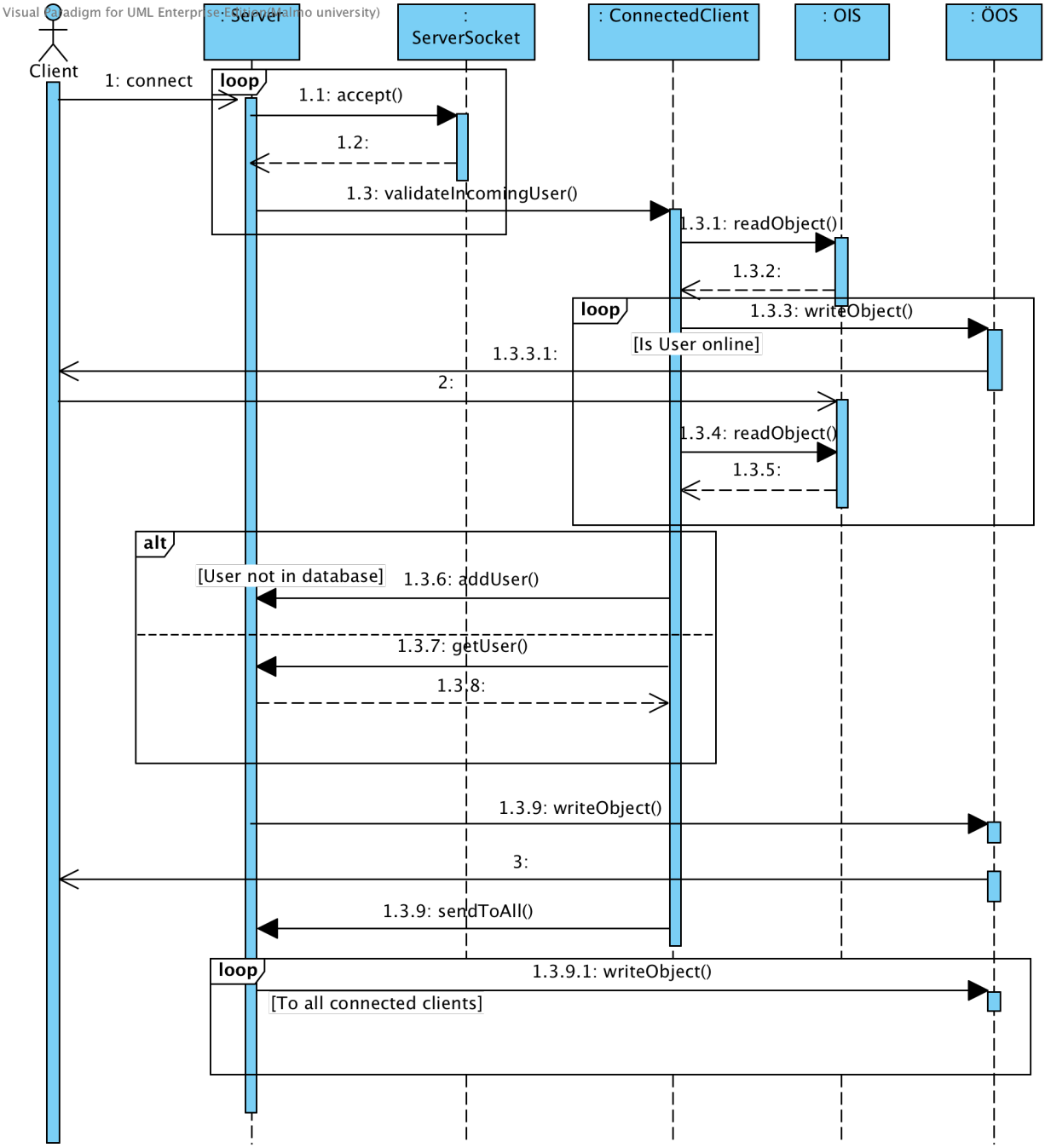
5.2 Client send Message



Figur 4: Client sending a message

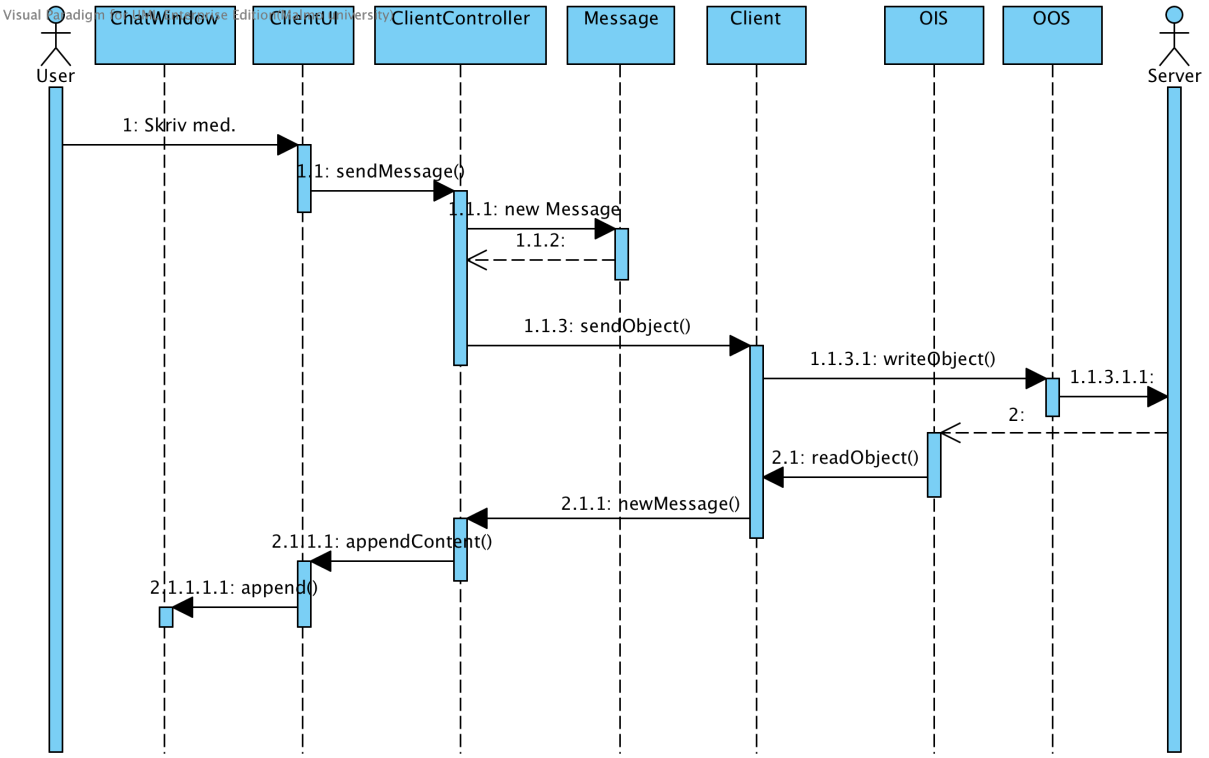
6 Sekvensdiagram

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.class.getName());
23
24      public Server(int port) {
25          initLogger();
26          registeredUsers = new ArrayList<>();
27          connectedClients = new ArrayList<>();
28          try {
29              serverSocket = new ServerSocket(port);
30              new Thread(this).start();
31          } catch (IOException e) {
32              e.printStackTrace();
33          }
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.getFromUserID())) {
96                         senderUser = cClient.getUser();
97 
98                         // Finds the conversation the message shall be sent to
99                         for (Conversation con : senderUser.getConversations()) {
100                            if (con.getId() == message.getConversationID()) {
101                                conversation = con;
102                                to += conversation.getInvolvedUsers().toString();
103 
104                                // Finds the message's recipient users, then sends the
                                    message
105                                for (String s : con.getInvolvedUsers()) {
106                                    for (ConnectedClient conClient : connectedClients)
                                        {
107                                        if (conClient.getUser().getId().equals(s)) {
108                                            conClient.sendObject(message);
109                                        }
110                                    }
111                                }
112                                conversation.addMessage(message);
113                            }
114                        }
115                    }
116                }
117            }
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 connection.
154      * Adds client to list of connected clients.
155      */
156     public void run() {
157         LOGGER.info("Server started.");
158         while (true) {
159             try {
160                 Socket socket = serverSocket.accept();
161                 ConnectedClient client = new ConnectedClient(socket, this);
162                 connectedClients.add(client);
163             } catch (IOException e) {
164                 e.printStackTrace();
165             }
166         }
167     }
168
169     /**
170      * Class to handle the communication between server and connected clients.
171      */
172     private class ConnectedClient implements Runnable {
173         private Thread client = new Thread(this);
174         private ObjectOutputStream oos;
175         private ObjectInputStream ois;
176         private Server server;
177         private User user;
178         private Socket socket;
179
180         public ConnectedClient(Socket socket, Server server) {
181             LOGGER.info("Client connected: " + socket.getInetAddress());
182             this.socket = socket;
183             this.server = server;
184             try {
185                 oos = new ObjectOutputStream(socket.getOutputStream());
186                 ois = new ObjectInputStream(socket.getInputStream());
187             } catch (IOException e) {
188                 e.printStackTrace();
189             }
190             client.start();
191         }
192
193         /**
194          * Returns the connected clients current User.
195          *
```

```
196             * @return The connected clients current User
197             */
198            public User getUser() {
199                return user;
200            }
201
202            /**
203             * Sends an object to the client.
204             *
205             * @param object The object to be sent.
206             */
207            public synchronized void sendObject(Object object) {
208                try {
209                    oos.writeObject(object);
210                } catch (IOException e) {
211                    e.printStackTrace();
212                }
213            }
214
215            /**
216             * Removes the user from the list of connected clients.
217             */
218            public void removeConnectedClient() {
219                for (int i = 0; i < connectedClients.size(); i++) {
220                    if (connectedClients.get(i).getUser().getId().equals(this.getUser()
                           .getId())) {
221                        connectedClients.remove(i);
222                        System.out.println("Client removed from connectedClients");
223                    }
224                }
225            }
226
227            /**
228             * Removes the connected client,
229             * sends an updated list of connected clients to other connected clients,
230             * sends a server message with information of who disconnected
231             * and closes the client's socket.
232             */
233            public void disconnectClient() {
234                removeConnectedClient();
235                sendConnectedClients();
236                sendObjectToAll("Client disconnected: " + user.getId());
237                LOGGER.info("Client disconnected: " + user.getId());
238                try {
239                    socket.close();
240                } catch (Exception e) {
241                    e.printStackTrace();
242                }
243            }
244
245            /**
246             * Checks if given user exists among already registered users.
247             *
248             * @return Whether given user already exists or not.
249             */
250            public boolean isUserInDatabase(User user) {
251                for (User u : registeredUsers) {
252                    if (u.getId().equals(user.getId())) {
253                        return true;
```

```
254                 }
255             }
256             return false;
257         }
258 
259         public User getUser(String ID) {
260             for (User user : registeredUsers) {
261                 if (user.getId().equals(ID)) {
262                     return user;
263                 }
264             }
265             return null;
266         }
267 
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             }
302             if (!exists) {
303                 conversation = new Conversation(participants);
304                 addConversation(conversation);
305             }
306             sendConversation(conversation);
307         }
308 
309         /**
310          * Adds given conversation to all its participants' User objects.
```

```
311             *
312             * @param con The conversation to be added.
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 conversation.
326             *
327             * @param message The message to be checked.
328             * @return Whether given message is part of a conversation or not.
329             */
330            public Conversation isPartOfConversation(Message message) {
331                for (Conversation con : user.getConversations()) {
332                    if (con.getId() == message.getConversationID()) {
333                        return con;
334                    }
335                }
336                return null;
337            }
338
339            /**
340             * Forces connecting users to pick a user that's not already logged in,
341             * and updates user database if needed.
342             * Announces connected to other connected users.
343             */
344            public void validateIncomingUser() {
345                Object object;
346                try {
347                    object = ois.readObject();
348                    user = (User) object;
349                    LOGGER.info("Checking online status for user: " + user.getId());
350                    while (isUserOnline(user.getId())) {
351                        LOGGER.info("User " + user.getId() + " already connected.
                                Asking for new name.");
352                        sendObject("Client named " + user.getId()+ " already connected,
                                try again!");
353                        // Wait for new user
354                        object = ois.readObject();
355                        user = (User) object;
356                        LOGGER.info("Checking online status for user: " + user.getId())
                                ;
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: " + user.getId());
365                    LOGGER.info("Client connected: " + user.getId());
366                    sendConnectedClients();
```

```
367             } catch (Exception e) {
368                 e.printStackTrace();
369             }
370         }
371 
372         /**
373          * Listens to incoming Messages, Conversations, HashSets of User IDs or
                server messages.
374          */
375         public void startCommunication() {
376             Object object;
377             Message message;
378             try {
379                 while (!Thread.interrupted()) {
380                     object = ois.readObject();
381                     if (object instanceof Message) {
382                         message = (Message) object;
383                         server.sendMessage(message);
384                     } else if (object instanceof Conversation) {
385                         Conversation con = (Conversation) object;
386                         oos.writeObject(con);
387                     } else if (object instanceof HashSet) {
388                         @SuppressWarnings("unchecked")
389                         HashSet<String> participants = (HashSet<String>) object;
390                         updateConversation(participants);
391                     } else {
392                         server.sendObjectToAll(object);
393                     }
394                 }
395             } catch (IOException e) {
396                 disconnectClient();
397                 e.printStackTrace();
398             } catch (ClassNotFoundException e2) {
399                 e2.printStackTrace();
400             }
401         }
402 
403         public void run() {
404             validateIncomingUser();
405             startCommunication();
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 FlowLayout());
30     private JPanel pnlServerCenterGrid = new JPanel(new GridLayout(1,2,5,5));
31     private JPanel pnlServerGrid = new JPanel(new GridLayout(2,1,5,5));
32     private JPanel pnlServerRunning = new JPanel(new BorderLayout());
33
34     private JTextField txtServerPort = new JTextField("3450");
35     private JLabel lblServerPort = new JLabel("Port:");
36     private JLabel lblServerShowServerIp = new JLabel();
37     private JLabel lblWelcome = new JLabel("Create a bIRC server");
38     private JLabel lblServerRunning = new JLabel("Server is 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.getSystemLookAndFeelClassName())
                       ;
165             } catch (ClassNotFoundException e) {
166                 e.printStackTrace();
167             } catch (InstantiationException e) {
168                 e.printStackTrace();
169             } catch (IllegalAccessException e) {
170                 e.printStackTrace();
171             } catch (UnsupportedLookAndFeelException e) {
172                 e.printStackTrace();
173             }
174       }
175 
176     /**
177      * Listener for create server. Starts a new server with the port of the
           textfield.
178      */
179     private class CreateStopServerListener implements ActionListener {
180         public void actionPerformed(ActionEvent e) {
181             if (btnServerCreateServer==e.getSource()) {
182                 server = new Server(getPort());
183                 setServerRunning();
184             }
185         }
```

```
186     }
187
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     private SimpleAttributeSet nameFont = new SimpleAttributeSet();
22
23     /**
24      * Constructor that takes an ID from a Conversation, and creates a window to
            display it.
25      *
26      * @param ID The Conversation object's ID.
27      */
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.
                VERTICAL_SCROLLBAR_AS_NEEDED);
35          scrollPane.setHorizontalScrollBarPolicy(JScrollPane.
                HORIZONTAL_SCROLLBAR_NEVER);
36
37          StyleConstants.setForeground(chatFont, Color.BLACK);
38          StyleConstants.setFontSize(chatFont, 20);
39
40          StyleConstants.setForeground(nameFont, Color.BLACK);
41          StyleConstants.setFontSize(nameFont, 20);
42          StyleConstants.setBold(nameFont, true);
43
44          add(scrollPane, BorderLayout.CENTER);
45          textPane.setEditable(false);
46      }
47
48      /**
49       * Appends a new message into the panel window.
50       * The message can either contain a String or an ImageIcon.
51       *
52       * @param message The message object which content will be displayed.
53       */
54      public void append(final Message message) {
55          SwingUtilities.invokeLater(new Runnable() {
56              @Override
57              public void run() {
58                  StyledDocument doc = textPane.getStyledDocument();
59                  try {
60                      doc.insertString(doc.getLength(), message.getTimestamp() + " -
                            ", chatFont);
61                      doc.insertString(doc.getLength(), message.getFromUserID() + ":
                            ", nameFont);
62                      if (message.getContent() instanceof String) {
63                          doc.insertString(doc.getLength(), (String)message.
                                getContent(), chatFont);
64                      } else {
65                          ImageIcon icon = (ImageIcon)message.getContent();
66                          StyleContext context = new StyleContext();
67                          Style labelStyle = context.getStyle(StyleContext.
                                DEFAULT_STYLE);
68                          JLabel label = new JLabel(icon);
69                          StyleConstants.setComponent(labelStyle, label);
70                          doc.insertString(doc.getLength(), "Ignored", labelStyle);
71                      }
72                      doc.insertString(doc.getLength(), "\n", chatFont);
73                      textPane.setCaretPosition(textPane.getDocument().getLength());
74
75                  } catch (BadLocationException e) {
76                      e.printStackTrace();
77                  }
78              }
79          });
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: " + stringMessage + "]\n",
                       chatFont);
91             } catch (BadLocationException e) {
92                 e.printStackTrace();
93             }
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   * 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      /**
29       * Constructor that creates a new Client with given ip, port 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 server.
55          */
56         public void sendObject(Object object) {
57             try {
58                 oos.writeObject(object);
59                 oos.flush();
60             } catch (IOException e) {}
61         }
62 
63         /**
64          * Sets the client user by creating a new User object with 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 displayed in the UI.
92          */
```

```
93      public void initConversations() {
94          for (Conversation con : user.getConversations()) {
95              controller.newConversation(con);
96          }
97      }
98
99      /**
100      * Asks for a username, creates a User object with given name and sends it to
             the server.
101      * The server then either accepts or denies the User object.
102      * If successful, sets the received User object as current user and announces
             login in chat.
103      * If not, notifies in chat and requests a new name.
104      */
105     public synchronized void setUser() {
106         Object object = null;
107         setName(this.name);
108         while (!(object instanceof User)) {
109             try {
110                 sendObject(user);
111                 object = ois.readObject();
112                 if (object instanceof User) {
113                     user = (User)object;
114                     controller.newMessage("You logged in as " + user.getId());
115                     initConversations();
116
117                 } else {
118                     controller.newMessage(object);
119                     this.name = JOptionPane.showInputDialog("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
131     /**
132      * Listens to incoming Messages, user lists, Conversations or server messages,
             and deal with them accordingly.
133      */
134     public void startCommunication() {
135         Object object;
136         try {
137             while (!Thread.interrupted()) {
138                 object = ois.readObject();
139                 if (object instanceof Message) {
140
141                     controller.newMessage(object);
142                 } else if (object instanceof ArrayList) {
143                     ArrayList<String> userList = (ArrayList<String>) object;
144                     controller.setConnectedUsers(userList);
145                 } else if (object instanceof Conversation) {
146                     Conversation con = (Conversation)object;
147                     user.addConversation(con);
148                     controller.newConversation(con);
```

```
149                 } else {
150                     controller.newMessage(object);
151                 }
152             }
153         } catch (IOException e) {
154             e.printStackTrace();
155         } catch (ClassNotFoundException e2) {
156             e2.printStackTrace();
157         }
158     }
159 
160     /**
161      * Class to handle communication between client and server.
162      */
163     private class ClientListener extends Thread {
164         public void run() {
165             setUser();
166             startCommunication();
167         }
168     }
169 }
```

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 GUI.
11  *
12  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
13  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
14  */
15 public class ClientController {
16     private ClientUI ui = new ClientUI(this);
17     private Client client;
18 
19     /**
20      * Creates a new Controller (with given Client).
21      * Also creates a new UI, and displays it in a JFrame.
22      *
23      * @param client
24      */
25     public ClientController(Client client) {
26         this.client = client;
27         SwingUtilities.invokeLater(new Runnable() {
28             public void run() {
29                 JFrame frame = new JFrame("bIRC");
30                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
31                 frame.add(ui);
32                 frame.pack();
```

```
33                  frame.setLocationRelativeTo(null);
34                  frame.setVisible(true);
35                  ui.focusTextField();
36              }
37          });
38      }
39
40      /**
41       * Receives an object that's either a Message object or a String
42       * and sends it to the UI.
43       *
44       * @param object A Message object or a String
45       */
46      public void newMessage(Object object) {
47          if (object instanceof Message) {
48              Message message = (Message)object;
49              ui.appendContent(message);
50          } else {
51              ui.appendServerMessage((String)object);
52          }
53      }
54
55      /**
56       * Returns the current user's ID.
57       *
58       * @return A string containing the current user's ID.
59       */
60      public String getUserID () {
61          return client.getUser().getId();
62      }
63
64      /**
65       * Creates a new message containing given ID and content, then sends it to the
             client.
66       *
67       * @param conID Conversation-ID of the message.
68       * @param content The message's content.
69       */
70      public void sendMessage(int conID, Object content) {
71          Message message = new Message(conID, client.getUser().getId(), content);
72          client.sendObject(message);
73      }
74
75      /**
76       * Takes a conversation ID and String with URL to image, scales the image and
             sends it to the client.
77       *
78       * @param conID Conversation-ID of the image.
79       * @param url A string containing the URl to the image to be sent.
80       */
81      public void sendImage(int conID, String url) {
82          ImageIcon icon = new ImageIcon(url);
83          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> setParticpants = new HashSet<>();
97          for(String participant: conversationParticipants) {
98              setParticpants.add(participant);
99          }
100         client.sendObject(setParticpants);
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     private JPanel eastPanelCenterNorth = new JPanel(new FlowLayout());
47     private JPanel pnlGroupSend = new JPanel(new GridLayout(1,2,8,8));
48     private JPanel pnlFileSend = new JPanel(new BorderLayout(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(tpConnectedUsers);
70     private JScrollPane scrollChatWindow = new JScrollPane(cwLobby);
71     private JScrollPane scrollGroupRooms = new JScrollPane(eastPanelCenterNorth);
72
73     private JButton[] groupChatList = new JButton[20];
```

```
74      private ArrayList<JCheckBox> arrayListCheckBox = new ArrayList<JCheckBox>();
75      private ArrayList<ChatWindow> arrayListChatWindows = new ArrayList<ChatWindow
            >();
76
77      private Font txtFont = new Font("Sans-Serif", Font.BOLD , 20);
78      private Font fontGroupButton = new Font("Sans-Serif",Font.PLAIN, 12);
79      private Font fontButtons = new Font("Sans-Serif", Font.BOLD,15);
80      private SimpleAttributeSet chatFont = new SimpleAttributeSet();
81
82      public ClientUI(ClientController clientController) {
83          this.clientController = clientController;
84          arrayListChatWindows.add(cwLobby);
85          groupPanel = new GroupPanel();
86          groupPanel.start();
87          lookAndFeel();
88          initGraphics();
89          initListeners();
90      }
91
92      /**
93       * 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(130,260));
100         initScroll();
101         initButtons();
102         add(scrollChatWindow, BorderLayout.CENTER);
103         southPanel();
104         eastPanel();
105     }
106
107     /**
108      * Initiates the butons.
109      * Also sets the icons and the design of the buttons.
110      */
111     public void initButtons() {
112         btnNewGroupChat.setIcon(new ImageIcon("src/resources/newGroup.png"));
113         btnNewGroupChat.setBorder(null);
114         btnNewGroupChat.setPreferredSize(new Dimension(64,64));
115
116         btnFileChooser.setIcon(new ImageIcon("src/resources/newImage.png"));
117         btnFileChooser.setBorder(null);
118         btnFileChooser.setPreferredSize(new Dimension(64, 64));
119
120         btnLobby.setFont(fontButtons);
121         btnLobby.setForeground(new Color(1,48,69));
122         btnLobby.setBackground(new Color(201,201,201));
123         btnLobby.setOpaque(true);
124         btnLobby.setBorderPainted(false);
125
126         btnCreateGroup.setFont(fontButtons);
127         btnCreateGroup.setForeground(new Color(1,48,69));
128     }
129
130     /**
131      * Initiates the scrollpanes and styleconstants.
```

```
132         */
133         public void initScroll() {
134             scrollChatWindow.setVerticalScrollBarPolicy(JScrollPane.
                    VERTICAL_SCROLLBAR_AS_NEEDED);
135             scrollChatWindow.setHorizontalScrollBarPolicy(JScrollPane.
                    HORIZONTAL_SCROLLBAR_NEVER);
136             scrollConnectedUsers.setVerticalScrollBarPolicy(JScrollPane.
                    VERTICAL_SCROLLBAR_AS_NEEDED);
137             scrollConnectedUsers.setHorizontalScrollBarPolicy(JScrollPane.
                    HORIZONTAL_SCROLLBAR_NEVER);
138             DefaultCaret caretConnected = (DefaultCaret)tpConnectedUsers.getCaret();
139             caretConnected.setUpdatePolicy(DefaultCaret.ALWAYS_UPDATE);
140             tpConnectedUsers.setEditable(false);
141 
142             tfMessageWindow.setFont(txtFont);
143             StyleConstants.setForeground(chatFont, Color.BLACK);
144             StyleConstants.setBold(chatFont, true);
145         }
146 
147         /**
148          * Requests that tfMessageWindow gets focus.
149          */
150         public void focusTextField() {
151             tfMessageWindow.requestFocusInWindow();
152         }
153 
154         /**
155          * Initialises listeners.
156          */
157         public void initListeners() {
158             tfMessageWindow.addKeyListener(new EnterListener());
159             GroupListener groupListener = new GroupListener();
160             SendListener sendListener = new SendListener();
161             LobbyListener disconnectListener = new LobbyListener();
162             btnNewGroupChat.addActionListener(groupListener);
163             btnCreateGroup.addActionListener(groupListener);
164             btnLobby.addActionListener(disconnectListener);
165             btnFileChooser.addActionListener(new FileChooserListener());
166             btnSend.addActionListener(sendListener);
167         }
168 
169         /**
170          * The method takes a ArrayList of the connected users and sets the user-
                checkboxes and
171          * the connected user textpane based on the users in the ArrayList.
172          *
173          * @param connectedUsers The ArrayList of the connected users.
174          */
175         public void setConnectedUsers(ArrayList<String> connectedUsers) {
176             setUserText();
177             tpConnectedUsers.setText("");
178             updateCheckBoxes(connectedUsers);
179             for (String ID : connectedUsers) {
180                 appendConnectedUsers(ID);
181             }
182         }
183 
184         /**
185          * Sets the usertext in the labels to the connected user.
```

```
186         */
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 of the message
                object.
230          *
231          * @param message The message object with an ID and a message.
232          */
233         public void appendContent(Message message) {
234 
235 
236 
237             getChatWindow(message.getConversationID()).append(message);
238             if(activeChatWindow != message.getConversationID()) {
239                 highlightGroup(message.getConversationID());
240             }
241         }
242 
243         /**
```

```
244         * The method handles notice.
245         *
246         * @param ID The ID of the group.
247         */
248        public void highlightGroup(int ID) {
249            if(ID != -1)
250                groupChatList[ID].setBackground(Color.PINK);
251        }
252
253        /**
254         * Appends the string content in the chatwindow-lobby.
255         *
256         * @param content Is a server message
257         */
258        public void appendServerMessage(String content) {
259            cwLobby.append(content.toString());
260        }
261
262        /**
263         * The method updates the ArrayList of checkboxes and add the checkboxes to the
                panel.
264         * Also checks if the ID is your own ID and doesn't add a checkbox of yourself.
265         * Updates the UI.
266         *
267         * @param checkBoxUserIDs ArrayList of UserID's.
268         */
269        public void updateCheckBoxes(ArrayList<String> 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 connected users.
285         *
286         * @param message Is a username.
287         */
288        public void appendConnectedUsers(String message){
289            StyledDocument doc = tpConnectedUsers.getStyledDocument();
290            try {
291                doc.insertString(doc.getLength(), message + "\n", chatFont);
292            } catch (BadLocationException e) {
293                e.printStackTrace();
294            }
295        }
296
297        /**
298         * Sets the text on the groupbuttons to the users you check in the checkbox.
299         * Adds the new group chat connected with a button and a ChatWindow.
300         * Enables you to change rooms.
301         * Updates UI.
```

```
302         *
303         * @param participants String-Array of the participants of the new groupchat.
304         * @param ID The ID of the participants of the new groupchat.
305         */
306        public void createConversation(String[] participants, int ID) {
307            GroupButtonListener gbListener = new GroupButtonListener();
308            for (int i = 0; i < participants.length; i++) {
309                if (!(participants[i].equals(clientController.getUserID()))) {
310                    if (i == participants.length - 1) {
311                        userString += participants[i];
312                    }else {
313                        userString += participants[i] + " " ;
314                    }
315                }
316            }
317            if (ID < groupChatList.length && groupChatList[ID] == null) {
318                groupChatList[ID] = (new JButton(userString));
319                groupChatList[ID].setPreferredSize(new Dimension(120,30));
320                groupChatList[ID].setOpaque(true);
321                groupChatList[ID].setBorderPainted(false);
322                groupChatList[ID].setFont(fontGroupButton);
323                groupChatList[ID].setForeground(new Color(93,0,0));
324                groupChatList[ID].addActionListener(gbListener);
325
326                eastPanelCenterNorth.add(groupChatList[ID]);
327
328                if (getChatWindow(ID)==null) {
329                    arrayListChatWindows.add(new ChatWindow(ID));
330                }
331
332                eastPanelCenterNorth.revalidate();
333                if (createdGroup) {
334                    if (activeChatWindow == -1) {
335                        btnLobby.setBackground(null);
336                    }
337                    else {
338                        groupChatList[activeChatWindow].setBackground(null);
339                    }
340
341                    groupChatList[ID].setBackground(new Color(201,201,201));
342                    remove(bL.getLayoutComponent(BorderLayout.CENTER));
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.getSystemLookAndFeelClassName())
                           ;
359                } catch (ClassNotFoundException e) {
```

```
360                 e.printStackTrace();
361             } catch (InstantiationException e) {
362                 e.printStackTrace();
363             } catch (IllegalAccessException e) {
364                 e.printStackTrace();
365             } catch (UnsupportedLookAndFeelException e) {
366                 e.printStackTrace();
367             }
368     }
369
370     /**
371      * The method goes through the ArrayList of chatwindow object and
372      * returns the correct one based on the ID.
373      *
374      * @param ID The ID of the user.
375      * @return ChatWindow A ChatWindow object with the correct ID.
376      */
377     public ChatWindow getChatWindow(int ID) {
378         for(ChatWindow cw : arrayListChatWindows) {
379             if(cw.getID() == ID) {
380                 return cw;
381             }
382         }
383         return null;
384     }
385
386     /**
387      * The class extends Thread and handles the Create a group panel.
388      */
389     private class GroupPanel extends Thread {
390         private JFrame groupFrame;
391         private JPanel pnlOuterBorderLayout = new JPanel(new BorderLayout());
392         private JPanel pnlNewGroup = new JPanel();
393         private JScrollPane scrollCheckConnectedUsers = new 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.DISPOSE_ON_CLOSE);
411             groupFrame.add(pnlOuterBorderLayout);
412             groupFrame.pack();
413             groupFrame.setVisible(false);
414             groupFrame.setLocationRelativeTo(null);
415         }
416
417         /**
```

```
418             * Initiates the scrollpanels and the panels of the groupPanel.
419             */
420            public void panelBuilder() {
421                scrollCheckConnectedUsers.setVerticalScrollBarPolicy(JScrollPane.
                       VERTICAL_SCROLLBAR_AS_NEEDED);
422                scrollCheckConnectedUsers.setHorizontalScrollBarPolicy(JScrollPane.
                       HORIZONTAL_SCROLLBAR_NEVER);
423                btnCreateGroup.setText("New Conversation");
424                pnlOuterBorderLayout.add(btnCreateGroup, BorderLayout.SOUTH);
425                pnlOuterBorderLayout.add(scrollCheckConnectedUsers,BorderLayout.CENTER)
                       ;
426                scrollCheckConnectedUsers.setPreferredSize(new Dimension(200,500));
427                pnlNewGroup.setLayout(new GridLayout(100,1,5,5));
428            }
429        }
430
431        /**
432         * KeyListener for the messagewindow.
433         * Enables you to send a message with enter.
434         */
435        private class EnterListener implements KeyListener {
436            public void keyPressed(KeyEvent e) {
437                if (e.getKeyCode() == KeyEvent.VK_ENTER && !(tfMessageWindow.getText().
                       isEmpty())) {
438                    clientController.sendMessage(activeChatWindow, tfMessageWindow.
                           getText());
439                    tfMessageWindow.setText("");
440                }
441            }
442
443            public void keyReleased(KeyEvent e) {}
444
445            public void keyTyped(KeyEvent e) {}
446        }
447
448        /**
449         * Listener that listens to New Group Chat-button and the Create Group Chat-
                button.
450         * If create group is pressed, a new button will be created with the right name
                ,
451         * the right participants.
452         * The method use alot of ArrayLists of checkboxes, 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<String>();
457            private String[] temp;
458            public void actionPerformed(ActionEvent e) {
459                if (btnNewGroupChat == e.getSource() && arrayListCheckBox.size() > 0) {
460                    groupPanel.getFrame().setVisible(true);
461                }
462                if (btnCreateGroup == e.getSource()) {
463                    participants.clear();
464                    temp = null;
465                    for(int i = 0; i < arrayListCheckBox.size(); i++) {
466                        if(arrayListCheckBox.get(i).isSelected()) {
467                            participants.add(arrayListCheckBox.get(i).getText());
468                        }
469                    }
```

```
470
471                 temp = new String[participants.size() + 1];
472                 temp[0] = clientController.getUserID();
473                 for (int i = 1; i <= participants.size(); i++) {
474                     temp[i] = participants.get(i-1);
475                 }
476                 if (temp.length > 1) {
477                     clientController.sendParticipants(temp);
478                     groupPanel.getFrame().dispose();
479                     createdGroup = true;
480                 } else {
481                     JOptionPane.showMessageDialog(null, "You have to choose atleast
                         one person!");
482                 }
483             }
484         }
485     }
486
487     /**
488      * Listener that connects the right GroupChatButton in an ArrayList to the
            right
489      * active chat window.
490      * Updates the UI.
491      */
492     private class GroupButtonListener implements ActionListener {
493         public void actionPerformed(ActionEvent e) {
494             for(int i = 0; i < groupChatList.length; i++) {
495                 if(groupChatList[i]==e.getSource()) {
496                     if(activeChatWindow == -1) {
497                         btnLobby.setBackground(null);
498                     }
499                     else {
500                         groupChatList[activeChatWindow].setBackground(null);
501                     }
502                     groupChatList[i].setBackground(new Color(201,201,201));
503                     remove(bL.getLayoutComponent(BorderLayout.CENTER));
504                     add(getChatWindow(i), BorderLayout.CENTER);
505                     activeChatWindow = i;
506                     validate();
507                     repaint();
508                 }
509             }
510         }
511     }
512
513     /**
514      * Listener that connects the user with the lobby chatWindow through the Lobby
            button.
515      * Updates UI.
516      */
517     private class LobbyListener implements ActionListener {
518         public void actionPerformed(ActionEvent e) {
519             if (btnLobby==e.getSource()) {
520                 btnLobby.setBackground(new Color(201,201,201));
521                 if(activeChatWindow != -1)
522                     groupChatList[activeChatWindow].setBackground(null);
523                 remove(bL.getLayoutComponent(BorderLayout.CENTER));
524                 add(getChatWindow(-1), BorderLayout.CENTER);
525                 activeChatWindow = -1;
```

```
526                 invalidate();
527                 repaint();
528             }
529         }
530     }
531 
532     /**
533      * Listener that creates a JFileChooser when the button btnFileChooser is
           pressed.
534      * The JFileChooser is for images in the chat and it calls the method sendImage
           in the controller.
535      */
536     private class FileChooserListener implements ActionListener {
537         public void actionPerformed(ActionEvent e) {
538             if (btnFileChooser==e.getSource()) {
539                 JFileChooser fileChooser = new JFileChooser();
540                 int returnValue = fileChooser.showOpenDialog(null);
541                 if (returnValue == JFileChooser.APPROVE_OPTION) {
542                     File selectedFile = fileChooser.getSelectedFile();
543                     String fullPath = selectedFile.getAbsolutePath();
544                     clientController.sendImage(activeChatWindow, fullPath);
545                 }
546             }
547         }
548     }
549 
550     /**
551      * Listener for the send message button.
552      * Resets the message textfield text.
553      */
554     private class SendListener implements ActionListener {
555         public void actionPerformed(ActionEvent e) {
556             if (btnSend==e.getSource() && !(tfMessageWindow.getText().isEmpty()))
                   {
557 
558                     clientController.sendMessage(activeChatWindow, tfMessageWindow.
                        getText());
559                     tfMessageWindow.setText("");
560             }
561         }
562     }
563 }
```

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 
```

```java
import org.imgscalr.Scalr;
import org.imgscalr.Scalr.Method;

/**
 * Scales down images to preferred size.
 *
 * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
 */
public class ImageScaleHandler {

    private static BufferedImage toBufferedImage(Image img) {
        if (img instanceof BufferedImage) {
            return (BufferedImage) img;
        }
        BufferedImage bimage = new BufferedImage(img.getWidth(null),
                img.getHeight(null), BufferedImage.TYPE_INT_ARGB);
        Graphics2D bGr = bimage.createGraphics();
        bGr.drawImage(img, 0, 0, null);
        bGr.dispose();
        return bimage;
    }

    public static BufferedImage createScaledImage(Image img, int height) {
        BufferedImage bimage = toBufferedImage(img);
        bimage = Scalr.resize(bimage, Method.ULTRA_QUALITY,
                Scalr.Mode.FIT_TO_HEIGHT, 0, height);
        return bimage;
    }

    // Example
    public static void main(String[] args) {
        ImageIcon icon = new ImageIcon("src/filer/new1.jpg");
        Image img = icon.getImage();

        // Use this to scale images
        BufferedImage scaledImage = ImageScaleHandler.createScaledImage(img, 75);
        icon = new ImageIcon(scaledImage);

        JLabel lbl = new JLabel();
        lbl.setIcon(icon);
        JPanel panel = new JPanel();
        panel.add(lbl);
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

Listing 7: ImageScaleHandler

7.2.6 StartClient.java

```java
package chat;

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

```
63          */
64         public void initListeners() {
65             LogInMenuListener log = new LogInMenuListener();
66             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.getSystemLookAndFeelClassName());
135         } catch (ClassNotFoundException e) {
136             e.printStackTrace();
137         } catch (InstantiationException e) {
138             e.printStackTrace();
139         } catch (IllegalAccessException e) {
140             e.printStackTrace();
141         } catch (UnsupportedLookAndFeelException e) {
142             e.printStackTrace();
143         }
144     }
145 
146     /**
147      * Main method for the login-frame.
148      */
149     public static void main(String[] args) {
150         SwingUtilities.invokeLater(new Runnable() {
151             @Override
152             public void run() {
153                 StartClient ui = new StartClient();
154             }
155         });
156 
157     }
158 
159     /**
160      * Listener for login-button, create server-button and for the cancel-button.
161      * Also limits the username to a 10 char max.
162      */
163     private class LogInMenuListener implements ActionListener {
164         public void actionPerformed(ActionEvent e) {
165             if (btnLogIn==e.getSource()) {
166                     if (txtUserName.getText().length() <= 10) {
167                         new Client(txtIp.getText(), Integer.parseInt(txtPort.
                                getText()),txtUserName.getText());
168                     } else {
169                     JOptionPane.showMessageDialog(null, "Namnet får max vara 10
                            karaktärer!");
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 instead of login.
181      * Also limits the username to 10 chars.
182      */
183     private class EnterListener implements ActionListener {
184         public void actionPerformed(ActionEvent e) {
185             if(txtUserName.getText().length() <= 10) {
186                 new Client(txtIp.getText(), Integer.parseInt(txtPort.getText()),
                        txtUserName.getText());
187                 frame.dispose();
188             } else {
189                 JOptionPane.showMessageDialog(null, "Namnet får max vara 10 karaktä
                        rer!");
190                 txtUserName.setText("");
191             }
192         }
193     }
194 }
```

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     private LinkedList<Message> list = new LinkedList<Message>();
15     private static int MESSAGE_LIMIT = 30;
16     private static final long serialVersionUID = 13371337133732526L;
17
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     }
```

```
}
```

Listing 9: ChatLog

7.3.2 Message

```
package chat;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Model class to handle messages
 *
 * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
 */
public class Message implements Serializable {
    private String fromUserID;
    private Object content;
    private String timestamp;
    private int conversationID = -1;    /* -1 means it's a lobby message */
    private static final long serialVersionUID = 133713371337L;

    /**
     * Constructor that creates a new message with given conversation ID, String
         with information who sent it, and its content.
     *
     * @param conversationID The conversation ID.
     * @param fromUserID A string with information who sent the message.
     * @param content The message's content.
     */
    public Message(int conversationID, String fromUserID, Object content) {
        this.conversationID = conversationID;
        this.fromUserID = fromUserID;
        this.content = content;
        newTime();
    }

    /**
     * Creates a new timestamp for the message.
     */
    private void newTime() {
        Date time = new Date();
        SimpleDateFormat ft = new SimpleDateFormat("HH:mm:ss");
        this.timestamp = ft.format(time);
    }

    /**
     * Returns a string containing sender ID.
     *
     * @return A string with the sender ID.
     */
    public String getFromUserID() {
        return fromUserID;
    }

```

```
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 conversation.
22       */
23      public Conversation(HashSet<String> involvedUsersID) {
24          this.involvedUsers = involvedUsersID;
25          this.conversationLog = new ChatLog();
26          id = ++numberOfConversations;
```

```
27      }
28
29      /**
30       * Returns a HashSet of the conversation's involved users.
31       *
32       * @return A hashSet of the conversation's involved users.
33       */
34      public HashSet<String> getInvolvedUsers() {
35          return involvedUsers;
36      }
37
38      /**
39       * 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