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

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.

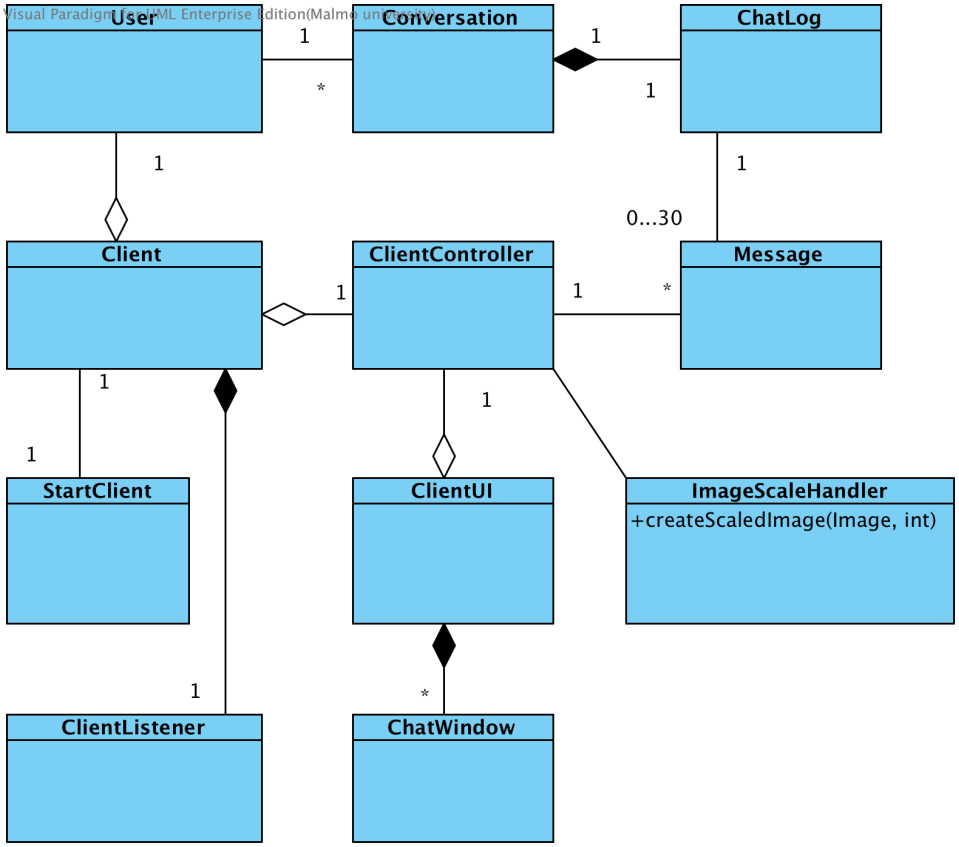
3 Systembeskrivning

Vårt system förser en Chatt-tjänst. I systemet finns det klienter och en server. Klienterna har ett grafiskt användargränssnitt för 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 personer och med rätt kontext, exempelvis som ett lobbymeddelande eller som ett meddelande i en viss gruppchatt.

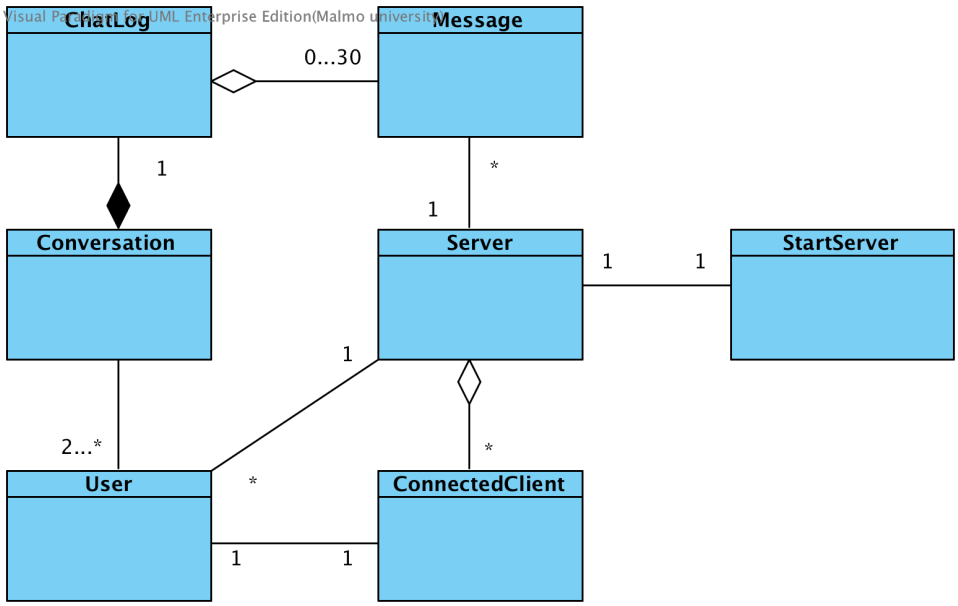
Servern lagrar alla textmeddelande som användarna skickar och loggar även namnet på de bilder som skickas. Det loggas även när användare ansluter eller stänger ner anslutningen mot servern.

4 Klassdiagram

4.1 Server



4.2 Klient



5 Kommunikationsdiagram

5.1 Kommunikationsdiagram 1

5.2 Kommunikationsdiagram 2

6 Sekvensdiagram

6.1 Sekvensdiagram 1

6.2 Sekvensdiagram 2

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;
```

```java
import java.util.logging.*;

/**
 * Model class for the server.
 *
 * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
 */
public class Server implements Runnable {
    private ServerSocket serverSocket;
    private ArrayList<ConnectedClient> connectedClients;
    private ArrayList<User> registeredUsers;
    private static final Logger LOGGER = Logger.getLogger(Server
        .class.getName());

    public Server(int port) {
        initLogger();
        registeredUsers = new ArrayList<>();
        connectedClients = new ArrayList<>();
        try {
            serverSocket = new ServerSocket(port);
            new Thread(this).start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    /**
     * Initiates the Logger
     */
    private void initLogger() {
        Handler fh;
        try {
            fh = new FileHandler("./src/log/Server.log");
            LOGGER.addHandler(fh);
            SimpleFormatter formatter = new SimpleFormatter();
            fh.setFormatter(formatter);
            LOGGER.setLevel(Level.FINE);
        } catch (IOException e) {}
    }

    /**
     * Returns the User which ID matches the given ID.
     * Returns null if it doesn't exist.
     *
     * @param id The ID of the User that is to be found.
     * @return The matching User object, or null.
     */
    public User getUser(String id) {
        for (User user : registeredUsers) {
            if (user.getId().equals(id)) {
                return user;
            }
        }
```

```
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 javax.swing.*;
4 
5  import java.awt.*;
6  import java.awt.event.ActionEvent;
7  import java.awt.event.ActionListener;
8  import java.net.InetAddress;
9  import java.net.UnknownHostException;
10 
11 /**
12  * Create an server-panel class.
13  */
14 public class StartServer extends JPanel{
15     private JPanel pnlServerCenterFlow = new JPanel(new
           FlowLayout());
16     private JPanel pnlServerCenterGrid = new JPanel(new
           GridLayout(2,2,5,5));
17 
18     private JTextField txtServerPort = new JTextField("3450");
19     private JLabel lblServerPort = new JLabel("Port:");
20     private JLabel lblServerShowServerIp = new JLabel();
21 
22     private JButton btnServerCreateServer = new JButton("Create
           Server");
23 
24     private Font fontWelcome = new Font("Sans-Serif",Font.BOLD
           ,20);
25     private Font fontIpPort = new Font("Sans-Serif",Font.PLAIN
           ,17);
26     private Font fontButtons = new Font("Sans-Serif", Font.BOLD,
            15);
27     private Font fontInfo = new Font("Sans-Serif",Font.BOLD|Font
           .ITALIC,17);
```

```
28
29      private Server server;
30
31      public StartServer() {
32          initPanels();
33          initLabels();
34          setlblServerShowServerIp();
35          initListeners();
36      }
37
38      /**
39       * Initiate Server-Panels.
40       */
41      public void initPanels() {
42          setPreferredSize(new Dimension(350,110));
43          add(pnlServerCenterFlow,BorderLayout.CENTER);
44          pnlServerCenterFlow.add(pnlServerCenterGrid);
45          add(lblServerShowServerIp,BorderLayout.SOUTH);
46
47          pnlServerCenterFlow.setOpaque(true);
48          pnlServerCenterFlow.setBackground(Color.WHITE);
49          pnlServerCenterGrid.setOpaque(true);
50          pnlServerCenterGrid.setBackground(Color.WHITE);
51
52          pnlServerCenterGrid.add(lblServerPort);
53          pnlServerCenterGrid.add(txtServerPort);
54          pnlServerCenterGrid.add(btnServerCreateServer);
55      }
56
57      /**
58       * Initiate Server-Labels.
59       */
60      public void initLabels() {
61          lblServerShowServerIp.setFont(fontInfo);
62          lblServerShowServerIp.setHorizontalAlignment(JLabel.
                CENTER);
63          lblServerPort.setFont(fontIpPort);
64          lblServerPort.setOpaque(true);
65          lblServerPort.setBackground(Color.WHITE);
66      }
67
68      public void initListeners() {
69          CreateStopServerListener create = new
                CreateStopServerListener();
70          btnServerCreateServer.addActionListener(create);
71      }
72
73      /**
74       * Sets the ip-label to the local ip of your own computer.
75       */
76      public void setlblServerShowServerIp() {
77          try {
78              String message = ""+ InetAddress.getLocalHost();
79              String realmessage[] = message.split("/");
```

```
80              lblServerShowServerIp.setText("Server ip is: " +
                    realmessage[1]);
81          } catch (UnknownHostException e) {
82              JOptionPane.showMessageDialog(null, "An error
                    occurred.");
83          }
84      }
85
86      public static void main(String[] args) {
87          StartServer server = new StartServer();
88          JFrame frame = new JFrame("bIRC Create Server");
89          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
90          frame.add(server);
91          frame.pack();
92          frame.setVisible(true);
93          frame.setLocationRelativeTo(null);
94          frame.setResizable(false);
95      }
96
97      /**
98       * Returns the port from the textfield.
99       *
100      * @return Port for creating a server.
101      */
102     public int getPort() {
103         return Integer.parseInt(this.txtServerPort.getText());
104     }
105
106     private class CreateStopServerListener implements
            ActionListener {
107         public void actionPerformed(ActionEvent e) {
108             if (btnServerCreateServer==e.getSource()) {
109                 server = new Server(getPort());
110             }
111         }
112     }
113 }
```

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      * Constructor that creates a new Client with given ip, port
            and user name.
29      *
30      * @param ip The IP address to connect to.
31      * @param port Port used in the connection.
32      * @param name The user name to connect with.
33      */
34     public Client(String ip, int port, String name) {
35         this.name = name;
36         try {
37             socket = new Socket(ip, port);
38             ois = new ObjectInputStream(socket.getInputStream())
                   ;
```

```
39              oos = new ObjectOutputStream(socket.getOutputStream
                    ());
40              controller = new ClientController(this);
41              new ClientListener().start();
42          } catch (IOException e) {
43              System.err.println(e);
44              if (e.getCause() instanceof SocketTimeoutException)
                    {
45
46              }
47          }
48      }
49
50      /**
51       * Sends an object object to the server.
52       *
53       * @param object The object that should be sent to the
             server.
54       */
55      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
             given name.
64       *
65       * @param name The name of the user to be created.
66       */
67      public void setName(String name) {
68          user = new User(name);
69      }
70
71      /**
72       * Returns the clients User object.
73       *
74       * @return The clients User object.
75       */
76      public User getUser() {
77          return user;
78      }
79
80      /**
81       * Closes the clients socket.
82       */
83      public void disconnectClient() {
84          try {
85              socket.close();
86          } catch (Exception e) {}
87      }
88
```

```
89      /**
90       * Sends the users conversations to the controller to be
             displayed in the UI.
91       */
92      public void initConversations() {
93          for (Conversation con : user.getConversations()) {
94              controller.newConversation(con);
95          }
96      }
97
98      /**
99       * Asks for a username, creates a User object with given
             name and sends it to the server.
100      * The server then either accepts or denies the User object.
101      * If successful, sets the received User object as current
             user and announces login in chat.
102      * If not, notifies in chat and requests a new name.
103      */
104     public synchronized void setUser() {
105         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 " +
                             user.getId());
114                     initConversations();
115                 } else {
116                     controller.newMessage(object);
117                     this.name = JOptionPane.showInputDialog("
                             Pick a name: ");
118                     setName(this.name);
119                 }
120             } catch (IOException e) {
121                 e.printStackTrace();
122             } catch (ClassNotFoundException e2) {
123                 e2.printStackTrace();
124             }
125
126         }
127     }
128
129     /**
130      * Listens to incoming Messages, user lists, Conversations
             or server messages, and deal with them accordingly.
131      */
132     public void startCommunication() {
133         Object object;
134         try {
135             while (!Thread.interrupted()) {
136                 object = ois.readObject();
```

```
137                 if (object instanceof Message) {
138                     controller.newMessage(object);
139                 } else if (object instanceof ArrayList) {
140                     ArrayList<String> userList = (ArrayList<
                            String>) object;
141                     controller.setConnectedUsers(userList);
142                 } else if (object instanceof Conversation) {
143                     Conversation con = (Conversation)object;
144                     user.addConversation(con);
145                     controller.newConversation(con);
146                 } else {
147                     controller.newMessage(object);
148                 }
149             }
150         } catch (IOException e) {
151             e.printStackTrace();
152         } catch (ClassNotFoundException e2) {
153             e2.printStackTrace();
154         }
155     }
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
        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         getChatWindow(message.getConversationID()).append(
                message);
235         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
                the checkboxes to the panel.
261      * Also checks if the ID is your own ID and doesn't add a
                checkbox of yourself.
262      * Updates the UI.
263      *
264      * @param checkBoxUserIDs ArrayList of UserID's.
265      */
```

```
266     public void updateCheckBoxes(ArrayList<String>
            checkBoxUserIDs) {
267         arrayListCheckBox.clear();
268         groupPanel.pnlNewGroup.removeAll();
269         for (String ID : checkBoxUserIDs) {
270             if (!ID.equals(clientController.getUserID())) {
271                 arrayListCheckBox.add(new JCheckBox(ID));
272             }
273         }
274         for (JCheckBox box: arrayListCheckBox) {
275             groupPanel.pnlNewGroup.add(box);
276         }
277         groupPanel.pnlOuterBorderLayout.revalidate();
278     }
279 
280     /**
281      * The method appends the text in the textpane of the
               connected users.
282      *
283      * @param message Is a username.
284      */
285     public void appendConnectedUsers(String message){
286         StyledDocument doc = tpConnectedUsers.getStyledDocument
                ();
287         try {
288             doc.insertString(doc.getLength(), message + "\n",
                    chatFont);
289         } catch (BadLocationException e) {
290             e.printStackTrace();
291         }
292     }
293 
294     /**
295      * Sets the text on the groupbuttons to the users you check
               in the checkbox.
296      * Adds the new group chat connected with a button and a
               ChatWindow.
297      * Enables you to change rooms.
298      * Updates UI.
299      *
300      * @param participants String-Array of the participants of
               the new groupchat.
301      * @param ID The ID of the participants of the new groupchat
               .
302      */
303     public void createConversation(String[] participants, int ID
            ) {
304         GroupButtonListener gbListener = new GroupButtonListener
                ();
305         for (int i = 0; i < participants.length; i++) {
306             if (!(participants[i].equals(clientController.
                    getUserID()))) {
307                 if (i == participants.length - 1) {
308                     userString += participants[i];
```

```
309                 }else {
310                     userString += participants[i] + " " ;
311                 }
312             }
313         }
314         if (ID < groupChatList.length && groupChatList[ID] ==
                null) {
315             groupChatList[ID] = (new JButton(userString));
316             groupChatList[ID].setPreferredSize(new Dimension
                    (120,30));
317             groupChatList[ID].setOpaque(true);
318             groupChatList[ID].setBorderPainted(false);
319             groupChatList[ID].setFont(fontGroupButton);
320             groupChatList[ID].setForeground(new Color(93,0,0));
321             groupChatList[ID].addActionListener(gbListener);
322
323             eastPanelCenterNorth.add(groupChatList[ID]);
324
325             if (getChatWindow(ID)==null) {
326                 arrayListChatWindows.add(new ChatWindow(ID));
327             }
328
329             eastPanelCenterNorth.revalidate();
330             if (createdGroup) {
331                 if (activeChatWindow == -1) {
332                     btnLobby.setBackground(null);
333                 }
334                 else {
335                     groupChatList[activeChatWindow].
                            setBackground(null);
336                 }
337
338                 groupChatList[ID].setBackground(new Color
                        (201,201,201));
339                 remove(bL.getLayoutComponent(BorderLayout.CENTER
                        ));
340                 add(getChatWindow(ID), BorderLayout.CENTER);
341                 activeChatWindow = ID;
342                 validate();
343                 repaint();
344                 createdGroup = false;
345             }
346         }
347         this.userString = "";
348     }
349
350     /**
351      * Sets the "Look and Feel" of the panels.
352      */
353     public void lookAndFeel() {
354          try {
355                 UIManager.setLookAndFeel(UIManager.
                        getSystemLookAndFeelClassName());
356             } catch (ClassNotFoundException e) {
```

```
357                 e.printStackTrace();
358             } catch (InstantiationException e) {
359                 e.printStackTrace();
360             } catch (IllegalAccessException e) {
361                 e.printStackTrace();
362             } catch (UnsupportedLookAndFeelException e) {
363                 e.printStackTrace();
364             }
365     }
366
367     /**
368      * The method goes through the ArrayList of chatwindow
             object and
369      * returns the correct one based on the ID.
370      *
371      * @param ID The ID of the user.
372      * @return ChatWindow A ChatWindow object with the correct
             ID.
373      */
374     public ChatWindow getChatWindow(int ID) {
375         for(ChatWindow cw : arrayListChatWindows) {
376             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
             panel.
385      */
386     private class GroupPanel extends Thread {
387         private JFrame groupFrame;
388         private JPanel pnlOuterBorderLayout = new JPanel(new
                 BorderLayout());
389         private JPanel pnlNewGroup = new JPanel();
390         private JScrollPane scrollCheckConnectedUsers = new
                 JScrollPane(pnlNewGroup);
391
392         /**
393          * The metod returns the JFrame groupFrame.
394          *
395          * @return groupFrame
396          */
397         public JFrame getFrame() {
398             return groupFrame;
399         }
400
401         /**
402          * Runs the frames of the groupPanes.
403          */
404         public void run() {
405             panelBuilder();
```

```
406             groupFrame = new JFrame();
407             groupFrame.setDefaultCloseOperation(JFrame.
                    DISPOSE_ON_CLOSE);
408             groupFrame.add(pnlOuterBorderLayout);
409             groupFrame.pack();
410             groupFrame.setVisible(false);
411             groupFrame.setLocationRelativeTo(null);
412         }
413
414         /**
415          * Initiates the scrollpanels and the panels of the
                groupPanel.
416          */
417         public void panelBuilder() {
418             scrollCheckConnectedUsers.setVerticalScrollBarPolicy
                    (JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
419             scrollCheckConnectedUsers.
                    setHorizontalScrollBarPolicy(JScrollPane.
                    HORIZONTAL_SCROLLBAR_NEVER);
420             btnCreateGroup.setText("New Conversation");
421             pnlOuterBorderLayout.add(btnCreateGroup,
                    BorderLayout.SOUTH);
422             pnlOuterBorderLayout.add(scrollCheckConnectedUsers,
                    BorderLayout.CENTER);
423             scrollCheckConnectedUsers.setPreferredSize(new
                    Dimension(200,500));
424             pnlNewGroup.setLayout(new GridLayout(100,1,5,5));
425         }
426     }
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 && !(
                    tfMessageWindow.getText().isEmpty())) {
435                 clientController.sendMessage(
                        activeChatWindow, tfMessageWindow.getText
                        ());
436                 tfMessageWindow.setText("");
437             }
438         }
439
440         public void keyReleased(KeyEvent e) {}
441
442         public void keyTyped(KeyEvent e) {}
443     }
444
445     /**
446      * Listener that listens to New Group Chat-button and the
            Create Group Chat-button.
```

```
447      * If create group is pressed, a new button will be created
             with the right name,
448      * the right participants.
449      * The method use alot of ArrayLists of checkboxes,
             participants and strings.
450      * Also some error-handling with empty buttons.
451      */
452     private class GroupListener implements ActionListener {
453         private ArrayList<String> participants = new ArrayList<
                String>();
454         private String[] temp;
455         public void actionPerformed(ActionEvent e) {
456             if (btnNewGroupChat == e.getSource() &&
                    arrayListCheckBox.size() > 0) {
457                 groupPanel.getFrame().setVisible(true);
458             }
459             if (btnCreateGroup == e.getSource()) {
460                 participants.clear();
461                 temp = null;
462                 for(int i = 0; i < arrayListCheckBox.size(); i
                        ++) {
463                     if(arrayListCheckBox.get(i).isSelected()) {
464                         participants.add(arrayListCheckBox.get(i
                                ).getText());
465                     }
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
                            have to choose atleast one person!");
479                 }
480             }
481         }
482     }
483 
484     /**
485      * Listener that connects the right GroupChatButton in an
             ArrayList to the right
486      * active chat window.
487      * Updates the UI.
488      */
489     private class GroupButtonListener implements ActionListener
            {
490         public void actionPerformed(ActionEvent e) {
491             for(int i = 0; i < groupChatList.length; i++) {
```

```
492                 if(groupChatList[i]==e.getSource()) {
493                     if(activeChatWindow == -1) {
494                         btnLobby.setBackground(null);
495                     }
496                     else {
497                         groupChatList[activeChatWindow].
                                setBackground(null);
498                     }
499                     groupChatList[i].setBackground(new Color
                            (201,201,201));
500                     remove(bL.getLayoutComponent(BorderLayout.
                            CENTER));
501                     add(getChatWindow(i), BorderLayout.CENTER);
502                     activeChatWindow = i;
503                     validate();
504                     repaint();
505                 }
506             }
507         }
508     }
509
510     /**
511      * Listener that connects the user with the lobby chatWindow
                through the Lobby button.
512      * Updates UI.
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].
                            setBackground(null);
520                 remove(bL.getLayoutComponent(BorderLayout.CENTER
                        ));
521                 add(getChatWindow(-1), BorderLayout.CENTER);
522                 activeChatWindow = -1;
523                 invalidate();
524                 repaint();
525             }
526         }
527     }
528
529     /**
530      * Listener that creates a JFileChooser when the button
            btnFileChooser is pressed.
531      * The JFileChooser is for images in the chat and it calls
            the method sendImage in the controller.
532      */
533     private class FileChooserListener implements ActionListener
            {
534         public void actionPerformed(ActionEvent e) {
535             if (btnFileChooser==e.getSource()) {
536                 JFileChooser fileChooser = new JFileChooser();
```

```
537                 int returnValue = fileChooser.showOpenDialog(
                        null);
538                 if (returnValue == JFileChooser.APPROVE_OPTION)
                        {
539                     File selectedFile = fileChooser.
                            getSelectedFile();
540                     String fullPath = selectedFile.
                            getAbsolutePath();
541                     clientController.sendImage(activeChatWindow,
                             fullPath);
542                 }
543             }
544         }
545     }
546
547     /**
548      * Listener for the send message button.
549      * Resets the message textfield text.
550      */
551     private class SendListener implements ActionListener {
552         public void actionPerformed(ActionEvent e) {
553             if (btnSend==e.getSource()  && !(tfMessageWindow.
                    getText().isEmpty())) {
554                     clientController.sendMessage(
                            activeChatWindow, tfMessageWindow.getText
                            ());
555                     tfMessageWindow.setText("");
556             }
557         }
558     }
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(
               null),
28                 img.getHeight(null), BufferedImage.TYPE_INT_ARGB
                   );
29         Graphics2D bGr = bimage.createGraphics();
30         bGr.drawImage(img, 0, 0, null);
31         bGr.dispose();
32         return bimage;
33     }
34
35     public static BufferedImage createScaledImage(Image img, int
            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.
               createScaledImage(img, 75);
49         icon = new ImageIcon(scaledImage);
50
51         JLabel lbl = new JLabel();
52         lbl.setIcon(icon);
53         JPanel panel = new JPanel();
54         panel.add(lbl);
55         JFrame frame = new JFrame();
56         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
57         frame.add(panel);
58         frame.pack();
59         frame.setVisible(true);
60     }
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     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
           ,20);
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 fontInfo = new Font("Sans-Serif",Font.BOLD|Font
           .ITALIC,17);
37
38     private BorderLayout borderLayout = new BorderLayout();
39     private JPanel pnlCenterGrid = new JPanel(new GridLayout
           (3,2,5,5));
40     private JPanel pnlCenterFlow = new JPanel(new FlowLayout());
41     private JPanel pnlNorthGrid = new JPanel(new GridLayout
           (2,1,5,5));
42     private JPanel pnlNorthGridGrid = new JPanel(new GridLayout
           (1,2,5,5));
43     private JFrame frame;
44
```

```
45      public StartClient() {
46          setLayout(new BorderLayout());
47          initPanels();
48          lookAndFeel();
49          initGraphics();
50          initButtons();
51          initListeners();
52      }
53
54      /**
55       * Initiates the listeners.
56       */
57      public void initListeners() {
58          LogInMenuListener log = new LogInMenuListener();
59          btnLogIn.addActionListener(log);
60          txtUserName.addActionListener(new EnterListener());
61          btnCancel.addActionListener(log);
62      }
63
64      /**
65       * Initiates the panels.
66       */
67      public void initPanels(){
68          setPreferredSize(new Dimension(430, 190));
69          pnlCenterGrid.setBounds(100, 200, 200, 50);
70          add(pnlCenterFlow,BorderLayout.CENTER);
71          pnlCenterFlow.add(pnlCenterGrid);
72
73          add(pnlNorthGrid,BorderLayout.NORTH);
74          pnlNorthGrid.add(lblWelcomeText);
75          pnlNorthGrid.add(pnlNorthGridGrid);
76          pnlNorthGridGrid.add(lblUserName);
77          pnlNorthGridGrid.add(txtUserName);
78
79          lblUserName.setHorizontalAlignment(JLabel.CENTER);
80          lblUserName.setFont(fontIpPort);
81          lblWelcomeText.setHorizontalAlignment(JLabel.CENTER);
82          lblWelcomeText.setFont(fontWelcome);
83          lblIp.setFont(fontIpPort);
84          lblPort.setFont(fontIpPort);
85      }
86
87      /**
88       * Initiates the buttons.
89       */
90      public void initButtons() {
91          btnCancel.setFont(fontButtons);
92          btnLogIn.setFont(fontButtons);
93
94          pnlCenterGrid.add(lblIp);
95          pnlCenterGrid.add(txtIp);
96          pnlCenterGrid.add(lblPort);
97          pnlCenterGrid.add(txtPort);
98          pnlCenterGrid.add(btnLogIn);
```

```
99          pnlCenterGrid.add(btnCancel);
100     }
101
102     /**
103      * Initiates the graphics and some design.
104      */
105     public void initGraphics() {
106         pnlCenterGrid.setOpaque(false);
107         pnlCenterFlow.setOpaque(false);
108         pnlNorthGridGrid.setOpaque(false);
109         pnlNorthGrid.setOpaque(false);
110         setBackground(Color.WHITE);
111         lblUserName.setBackground(Color.WHITE);
112         lblUserName.setOpaque(false);
113     }
114
115     /**
116      * Sets the "Look and Feel" of the JComponents.
117      */
118     public void lookAndFeel() {
119      try {
120             UIManager.setLookAndFeel(UIManager.
                    getSystemLookAndFeelClassName());
121         } catch (ClassNotFoundException e) {
122             e.printStackTrace();
123         } catch (InstantiationException e) {
124             e.printStackTrace();
125         } catch (IllegalAccessException e) {
126             e.printStackTrace();
127         } catch (UnsupportedLookAndFeelException e) {
128             e.printStackTrace();
129         }
130     }
131
132     /**
133      * Run method for the login-frame.
134      */
135     public static void main(String[] args) {
136         SwingUtilities.invokeLater(new Runnable() {
137             @Override
138             public void run() {
139                 JFrame frame = new JFrame("bIRC Login");
140                 StartClient ui = new StartClient();
141                 frame.setDefaultCloseOperation(JFrame.
                        DISPOSE_ON_CLOSE);
142                 frame.add(ui);
143                 frame.pack();
144                 frame.setVisible(true);
145                 frame.setLocationRelativeTo(null);
146                 frame.setResizable(false);
147             }
148         });
149
150     }
```

```
151
152
153
154     /**
155      * Listener for login-button, create server-button and for
             the cancel-button.
156      * Also limits the username to a 10 char max.
157      */
158     private class LogInMenuListener implements ActionListener {
159         public void actionPerformed(ActionEvent e) {
160             if (btnLogIn==e.getSource()) {
161                     if (txtUserName.getText().length() <= 10) {
162                         new Client(txtIp.getText(), Integer.
                                parseInt(txtPort.getText()),
                                txtUserName.getText());
163                     } else {
164                     JOptionPane.showMessageDialog(null, "Namnet
                            får max vara 10 karaktärer!");
165                     txtUserName.setText("");
166                 }
167             }
168             if (btnCancel==e.getSource()) {
169                 System.exit(0);
170             }
171         }
172     }
173
174     /**
175      * Listener for the textField. Enables you to press enter
             instead of login.
176      * Also limits the username to 10 chars.
177      */
178     private class EnterListener implements ActionListener {
179         public void actionPerformed(ActionEvent e) {
180                 if(txtUserName.getText().length() <= 10) {
181                     new Client(txtIp.getText(), Integer.parseInt
                            (txtPort.getText()),txtUserName.getText()
                            );
182                 } else {
183                 JOptionPane.showMessageDialog(null, "Namnet får
                        max vara 10 karaktärer!");
184                 txtUserName.setText("");
185                 }
186             }
187     }
188
189     /**
190      * Listener for textfield in create a server. Enables you to
               press enter to create server.
191      * Disposes the serverpanel on create.
192      */
193
194     /**
```

```
195         * Listener for the create server button and for the cancel
                button.
196         * Disposes the frames on click.
197         */
198        /**
199         * MAIN
200         *
201         * @param args
202         */
203 }
```

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 =
           1337133713373252 6L;
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);
    }
```

```
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
```

```
 */
public class User implements Serializable {
    private static final long serialVersionUID = 1273274782824L;
    private ArrayList<Conversation> conversations;
    private String id;

    /**
     * Constructor to create a User with given ID.
     *
     * @param id A string with the user ID.
     */
    public User(String id) {
        this.id = id;
        conversations = new ArrayList<>();
    }

    /**
     * Returns an ArrayList with the user's conversations
     *
     * @return The user's conversations.
     */
    public ArrayList<Conversation> getConversations() {
        return conversations;
    }

    /**
     * Adds a new conversation to the user.
     *
     * @param conversation The conversation to be added.
     */
    public void addConversation(Conversation conversation) {
        conversations.add(conversation);
    }

    /**
     * Returns the user's ID.
     *
     * @return The user's ID.
     */
    public String getId() {
        return id;
    }
}
```

Listing 11: User

7.3.4 Conversation

```
package chat;

import java.io.Serializable;
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