

Projektrapport
Chattapplikation
för Objektorienterad programutveckling, trådar och
datakommunikation

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

11 mars 2015

Innehåll

1	Arbetsbeskrivning	3
1.1	Rasmus Andersson	3
1.2	Emil Sandgren	3
1.3	Erik Sandgren	3
1.4	Jimmy Maksymiw	3
1.5	Lorenz Puskas	3
1.6	Kalle Bornemark	3
2	Instruktioner för programstart	4
3	Systembeskrivning	4
4	Klassdiagram	5
4.1	Server	5
4.2	Klient	6
5	Kommunikationsdiagram	6
5.1	Kommunikationsdiagram 1	6
5.2	Kommunikationsdiagram 2	6
6	Sekvensdiagram	6
6.1	Sekvensdiagram 1	6
6.2	Sekvensdiagram 2	6
7	Källkod	6
7.1	Server	6
7.1.1	Server.java, Server.ConnectedClient.java	6
7.1.2	Startserver.java	15
7.2	Klient	17
7.2.1	ChatWindow.java	17
7.2.2	Client.java	20
7.2.3	ClientController.java	23
7.2.4	ClientUI.java	26
7.2.5	ImageScaleHandler.java	38
7.2.6	StartClient.java	40
7.3	Delade klasser	44
7.3.1	ChatLog	44
7.3.2	Message	45
7.3.3	User	46
7.3.4	Conversation	47

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.

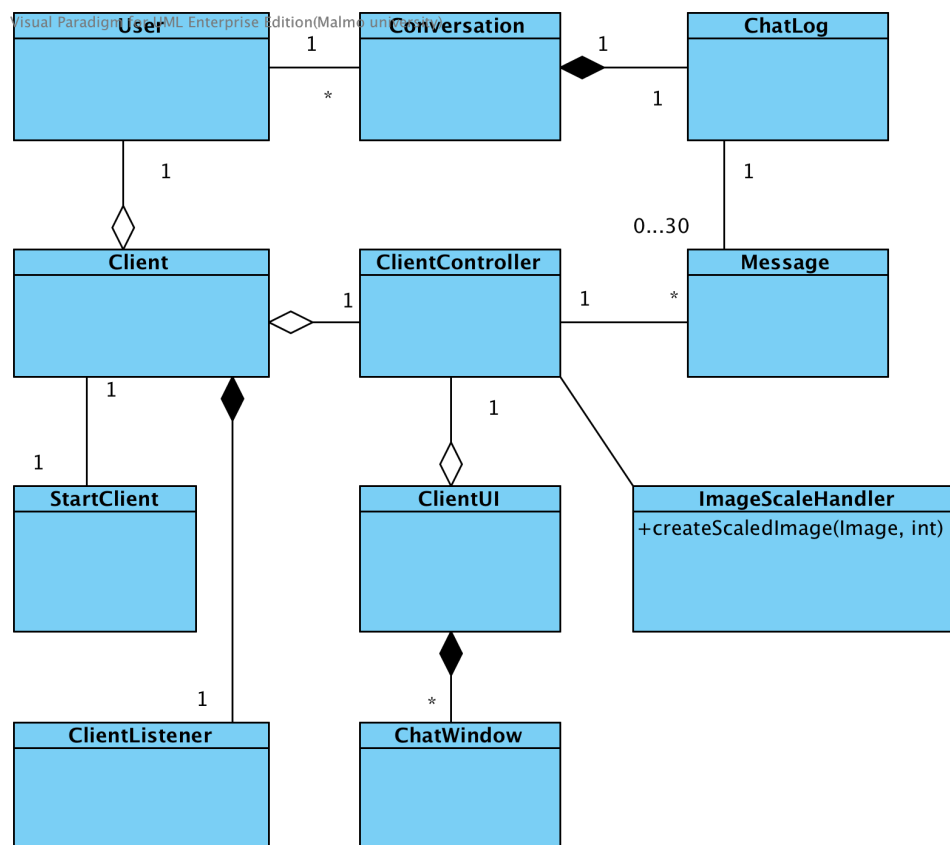
3 Systembeskrivning

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.

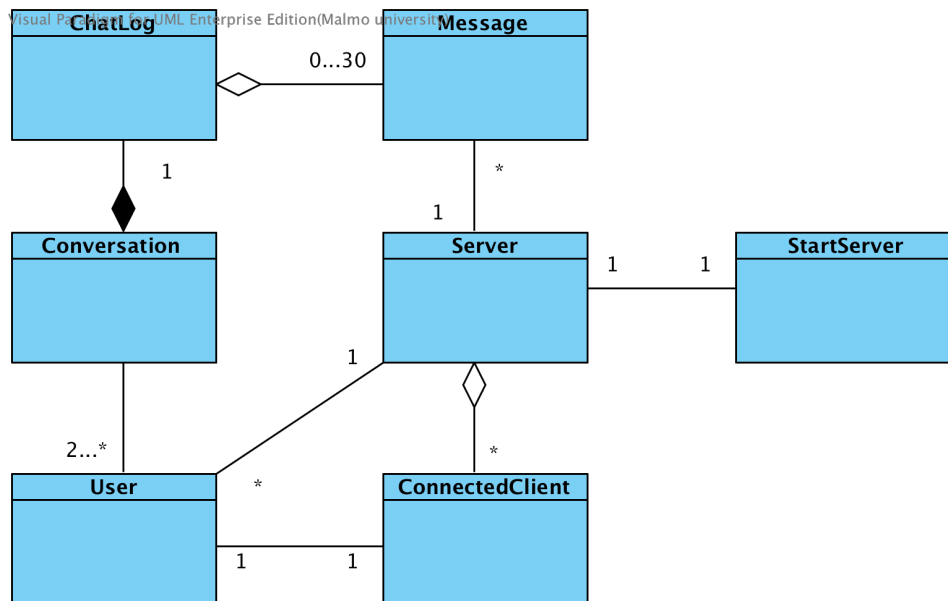
4 Klassdiagram

4.1 Server



Figur 1: GUI för Android-applikation

4.2 Klient



Figur 2: GUI för Android-applikation

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;
10 import java.util.logging.*;
11
12 /**
13  * Model class for the server.
14  *
15  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
16  * @author 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 }
```

```
61     }
62     }
63     return null;
64 }
65
66 /**
67  * Sends an object to all currently connected clients.
68  *
69  * @param object The object to be sent.
70  */
71 public synchronized void sendObjectToAll(Object object) {
72     for (ConnectedClient client : connectedClients) {
73         client.sendObject(object);
74     }
75 }
76
77 /**
78  * Checks who the message shall be sent to, then sends it.
79  *
80  * @param message The message to be sent.
81  */
82 public void sendMessage(Message message) {
83     Conversation conversation = null;
84     String to = "";
85
86     // Lobby message
87     if (message.getConversationID() == -1) {
88         sendObjectToAll(message);
89         to += "lobby";
90     } else {
91         User senderUser = null;
92
93         // Finds the sender user
94         for (ConnectedClient cClient : connectedClients) {
95             if (cClient.getUser().getId().equals(message.
96                 getFromUserID())) {
97                 senderUser = cClient.getUser();
98
99                 // Finds the conversation the message shall
100                 // be sent to
101                 for (Conversation con : senderUser.
102                     getConversations()) {
103                     if (con.getId() == message.
104                         getConversationID()) {
105                         conversation = con;
106                         to += conversation.getInvolvedUsers
107                             ().toString();
108
109                         // Finds the message's recipient
110                         // users, then sends the message
111                         for (String s : con.getInvolvedUsers
112                             ()) {
113                             for (ConnectedClient conClient :
114                                 connectedClients) {
```



```

107         if (conClient.getUser().
108             getId().equals(s)) {
109             conClient.sendObject(
110                 message);
111             }
112         }
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
154  * connection.
155  * Adds client to list of connected clients.
156  */
157 public void run() {
158     LOGGER.info("Server started.");

```

```
158         while (true) {
159             try {
160                 Socket socket = serverSocket.accept();
161                 ConnectedClient client = new ConnectedClient(
162                     socket, this);
163                 connectedClients.add(client);
164             } catch (IOException e) {
165                 e.printStackTrace();
166             }
167         }
168
169         /**
170          * Class to handle the communication between server and
171          * connected clients.
172          */
173         private class ConnectedClient implements Runnable {
174             private Thread client = new Thread(this);
175             private ObjectOutputStream oos;
176             private ObjectInputStream ois;
177             private Server server;
178             private User user;
179             private Socket socket;
180
181             public ConnectedClient(Socket socket, Server server) {
182                 LOG.info("Client connected: " + socket.
183                     getInetAddress());
184                 this.socket = socket;
185                 this.server = server;
186                 try {
187                     oos = new ObjectOutputStream(socket.
188                         getOutputStream());
189                     ois = new ObjectInputStream(socket.
190                         getInputStream());
191                 } catch (IOException e) {
192                     e.printStackTrace();
193                 }
194                 client.start();
195             }
196
197             /**
198              * Returns the connected clients current User.
199              *
200              * @return The connected clients current User
201              */
202             public User getUser() {
203                 return user;
204             }
205
206             /**
207              * Sends an object to the client.
208              *
209              * @param object The object to be sent.
210              */
```

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

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

```
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
311  * objects.
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
326  * conversation.
327  * @param message The message to be checked.
328  * @return Whether given message is part of a
329  *         conversation or not.
330  */
331 public Conversation isPartOfConversation(Message message) {
332     for (Conversation con : user.getConversations()) {
333         if (con.getId() == message.getConversationID()) {
334             return con;
335         }
336     }
337     return null;
338 }
339
340 /**
341  * Forces connecting users to pick a user that's not
342  * already logged in,
343  * and updates user database if needed.
344  * Announces connected to other connected users.
345  */
346 public void validateIncomingUser() {
347     Object object;
348     try {
349         object = ois.readObject();
350         user = (User) object;
```

```
349         LOGGER.info("Checking online status for user: "
350             + user.getId());
351         while (isUserOnline(user.getId())) {
352             LOGGER.info("User " + user.getId() + "
353                 already connected. Asking for new name.")
354             ;
355             sendObject("Client named " + user.getId() + "
356                 already connected, try again!");
357             // Wait for new user
358             object = ois.readObject();
359             user = (User) object;
360             LOGGER.info("Checking online status for user
361                 : " + user.getId());
362         }
363         if (!isUserInDatabase(user)) {
364             registeredUsers.add(user);
365         } else {
366             user = getUser(user.getId());
367         }
368         oos.writeObject(user);
369         server.sendObjectToAll("Client connected: " +
370             user.getId());
371         LOGGER.info("Client connected: " + user.getId())
372         ;
373         sendConnectedClients();
374     } catch (Exception e) {
375         e.printStackTrace();
376     }
377 }
378
379 /**
380  * Listens to incoming Messages, Conversations, HashSets
381  * of User IDs or server messages.
382  */
383 public void startCommunication() {
384     Object object;
385     Message message;
386     try {
387         while (!Thread.interrupted()) {
388             object = ois.readObject();
389             if (object instanceof Message) {
390                 message = (Message) object;
391                 server.sendMessage(message);
392             } else if (object instanceof Conversation) {
393                 Conversation con = (Conversation) object
394                 ;
395                 oos.writeObject(con);
396             } else if (object instanceof HashSet) {
397                 @SuppressWarnings("unchecked")
398                 HashSet<String> participants = (HashSet<
399                     String>) object;
400                 updateConversation(participants);
401             } else {
402                 server.sendObjectToAll(object);
403             }
404         }
405     } catch (Exception e) {
406         e.printStackTrace();
407     }
408 }
```

```
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
16         FlowLayout());
17     private JPanel pnlServerCenterGrid = new JPanel(new
18         GridLayout(2,2,5,5));
19
20     private JTextField txtServerPort = new JTextField("3450");
21     private JLabel lblServerPort = new JLabel("Port:");
22     private JLabel lblServerShowServerIp = new JLabel();
23
24     private JButton btnServerCreateServer = new JButton("Create
25         Server");
26
27     private Font fontWelcome = new Font("Sans-Serif",Font.BOLD
28         ,20);
29     private Font fontIpPort = new Font("Sans-Serif",Font.PLAIN
30         ,17);
31     private Font fontButtons = new Font("Sans-Serif",Font.BOLD,
32         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
107     ActionListener {
108     public void actionPerformed(ActionEvent e) {
109         if (btnServerCreateServer==e.getSource()) {
110             server= new Server(getPort());
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         ();
22     private SimpleAttributeSet nameFont = new SimpleAttributeSet
23         ();
24
25     /**
26      * Constructor that takes an ID from a Conversation, and
27      * creates a window to display it.
28      *
29      * @param ID The Conversation object's ID.
30      */
31     public ChatWindow(int ID) {
32         setLayout(new BorderLayout());
33         this.ID = ID;
34         textPane = new JTextPane();
35         scrollPane = new JScrollPane(textPane);
36
37         scrollPane.setVerticalScrollBarPolicy(JScrollPane.
38             VERTICAL_SCROLLBAR_AS_NEEDED);
39         scrollPane.setHorizontalScrollBarPolicy(JScrollPane.
40             HORIZONTAL_SCROLLBAR_NEVER);
41
42         StyleConstants.setForeground(chatFont, Color.BLACK);
43         StyleConstants.setFontSize(chatFont, 20);
44
45         StyleConstants.setForeground(nameFont, Color.BLACK);
46         StyleConstants.setFontSize(nameFont, 20);
47         StyleConstants.setBold(nameFont, true);
48
49         add(scrollPane, BorderLayout.CENTER);
50         textPane.setEditable(false);
51     }
52
53     /**
54      * Appends a new message into the panel window.
55      * The message can either contain a String or an ImageIcon.
56      *
57      * @param message The message object which content will be
58      * displayed.
59      */
60     public void append(final Message message) {
61         SwingUtilities.invokeLater(new Runnable() {
```

```

56         @Override
57         public void run() {
58             StyledDocument doc = textPane.getStyledDocument
59                 ();
60             try {
61                 doc.insertString(doc.getLength(), message.
62                     getTimestamp() + " - ", chatFont);
63                 doc.insertString(doc.getLength(), message.
64                     getFromUserID() + ": ", nameFont);
65                 if (message.getContent() instanceof String)
66                 {
67                     doc.insertString(doc.getLength(), (
68                         String)message.getContent(), chatFont
69                     );
70                 } else {
71                     ImageIcon icon = (ImageIcon)message.
72                         getContent();
73                     StyleContext context = new StyleContext
74                         ();
75                     Style labelStyle = context.getStyle(
76                         StyleContext.DEFAULT_STYLE);
77                     JLabel label = new JLabel(icon);
78                     StyleConstants.setComponent(labelStyle,
79                         label);
80                     doc.insertString(doc.getLength(), "
81                         Ignored", labelStyle);
82                 }
83                 doc.insertString(doc.getLength(), "\n",
84                     chatFont);
85                 textPane.setCaretPosition(textPane.
86                     getDocument().getLength());
87             } catch (BadLocationException e) {
88                 e.printStackTrace();
89             }
90         }
91     });
92 }
93
94 /**
95  * Appends a string into the panel window.
96  *
97  * @param stringMessage The string to be appended.
98  */
99 public void append(String stringMessage) {
100     StyledDocument doc = textPane.getStyledDocument();
101     try {
102         doc.insertString(doc.getLength(), "[Server: " +
103             stringMessage + "]\n", chatFont);
104     } catch (BadLocationException e) {
105         e.printStackTrace();
106     }
107 }

```

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

```
38         ois = new ObjectInputStream(socket.getInputStream())
39         ;
40         oos = new ObjectOutputStream(socket.getOutputStream())
41         ;
42         controller = new ClientController(this);
43         new ClientListener().start();
44     } catch (IOException e) {
45         System.err.println(e);
46         if (e.getCause() instanceof SocketTimeoutException)
47             {
48             }
49     }
50 }
51 /**
52  * Sends an object object to the server.
53  *
54  * @param object The object that should be sent to the
55  * server.
56  */
57 public void sendObject(Object object) {
58     try {
59         oos.writeObject(object);
60         oos.flush();
61     } catch (IOException e) {}
62 }
63 /**
64  * Sets the client user by creating a new User object with
65  * given name.
66  *
67  * @param name The name of the user to be created.
68  */
69 public void setName(String name) {
70     user = new User(name);
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  * 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<
141                     String>) object;
142                 controller.setConnectedUsers(userList);
143             } else if (object instanceof Conversation) {
144                 Conversation con = (Conversation) object;
145                 user.addConversation(con);
146                 controller.newConversation(con);
147             } else {
148                 controller.newMessage(object);
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
11  * GUI.
12  *
13  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
14  * 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.
31                      EXIT_ON_CLOSE);
32                  frame.add(ui);
33                  frame.pack();
34                  frame.setLocationRelativeTo(null);
35                  frame.setVisible(true);
36                  ui.focusTextField();
37              }
38          });
39      }
40
41      /**
42       * Receives an object that's either a Message object or a
43       * String
44       * and sends it to the UI.
45       *
46       * @param object A Message object or a String
47       */
48      public void newMessage(Object object) {
49          if (object instanceof Message) {
50              Message message = (Message) object;
51              ui.appendContent(message);
52          } else {
53              ui.appendServerMessage((String) object);
54          }
55      }
56
57      /**
58       * Returns the current user's ID.
59       *
60       * @return A string containing the current user's ID.
61       */
62      public String getUserID () {
63          return client.getUser().getId();
64      }
65
66      /**
```



```
65     * Creates a new message containing given ID and content ,
66     * then sends it to the client .
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().
72             getId(), content);
73         client.sendObject(message);
74     }
75
76     /**
77     * Takes a conversation ID and String with URL to image,
78     * scales the image and sends it to the client .
79     * @param conID Conversation-ID of the image .
80     * @param url A string containing the URL to the image to be
81     * sent .
82     */
83     public void sendImage(int conID, String url) {
84         ImageIcon icon = new ImageIcon(url);
85         Image img = icon.getImage();
86         BufferedImage scaledImage = ImageScaleHandler.
87             createScaledImage(img, 250);
88         icon = new ImageIcon(scaledImage);
89         sendMessage(conID, icon);
90     }
91
92     /**
93     * Creates a HashSet of given String array with participants
94     * , and sends it to the client .
95     * @param conversationParticipants A string array with
96     * conversaion participants .
97     */
98     public void sendParticipants(String []
99         conversationParticipants) {
100         HashSet<String> setParticipants = new HashSet<>();
101         for(String participant: conversationParticipants) {
102             setParticipants.add(participant);
103         }
104         client.sendObject(setParticipants);
105     }
106
107     /**
108     * Sends the ArrayList with connected users to the UI.
109     * @param userList The ArrayList with connected users .
110     */
111     public void setConnectedUsers(ArrayList<String> userList) {
112         ui.setConnectedUsers(userList);
113     }
```

```
111
112     /**
113      * Presents a Conversation in the UI.
114      *
115      * @param con The Conversation object to be presented in the
116      *           UI.
117      */
118     public void newConversation(Conversation con) {
119         HashSet<String> users = con.getInvolvedUsers();
120         String[] usersHashToStringArray = users.toArray(new
121             String[users.size()]);
122         int conID = con.getId();
123         ui.createConversation(usersHashToStringArray, conID);
124         for (Message message : con.getConversationLog()) {
125             ui.appendContent(message);
126         }
127     }
128 }
```

Listing 5: ClientController

7.2.4 ClientUI.java

```
1 package chat;
2
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import java.awt.Dimension;
6 import java.awt.FlowLayout;
7 import java.awt.Font;
8 import java.awt.GridLayout;
9 import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11 import java.awt.event.KeyEvent;
12 import java.awt.event.KeyListener;
13 import java.io.File;
14 import java.util.ArrayList;
15
16 import javax.swing.ImageIcon;
17 import javax.swing.JButton;
18 import javax.swing.JCheckBox;
19 import javax.swing.JFileChooser;
20 import javax.swing.JFrame;
21 import javax.swing.JLabel;
22 import javax.swing.JOptionPane;
23 import javax.swing.JPanel;
24 import javax.swing.JScrollPane;
25 import javax.swing.JTextField;
26 import javax.swing.JTextPane;
27 import javax.swing.UIManager;
28 import javax.swing.UnsupportedLookAndFeelException;
29 import javax.swing.text.BadLocationException;
```

```
30 import javax.swing.text.DefaultCaret;
31 import javax.swing.text.SimpleAttributeSet;
32 import javax.swing.text.StyleConstants;
33 import javax.swing.text.StyledDocument;
34
35 /**
36  * Viewer class to handle the GUI.
37  *
38  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
39  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
40  */
41
42 public class ClientUI extends JPanel {
43     private JPanel southPanel = new JPanel();
44     private JPanel eastPanel = new JPanel();
45     private JPanel eastPanelCenter = new JPanel(new BorderLayout
46         ());
47     private JPanel eastPanelCenterNorth = new JPanel(new
48         FlowLayout());
49     private JPanel pnlGroupSend = new JPanel(new GridLayout
50         (1,2,8,8));
51     private JPanel pnlFileSend = new JPanel(new BorderLayout
52         (5,5));
53
54     private String userString = "";
55     private int activeChatWindow = -1;
56     private boolean createdGroup = false;
57
58     private JLabel lblUser = new JLabel();
59     private JButton btnSend = new JButton("Send");
60     private JButton btnNewGroupChat = new JButton();
61     private JButton btnLobby = new JButton("Lobby");
62     private JButton btnCreateGroup = new JButton("");
63     private JButton btnFileChooser = new JButton();
64
65     private JTextPane tpConnectedUsers = new JTextPane();
66     private ChatWindow cwLobby = new ChatWindow(-1);
67     private ClientController clientController;
68     private GroupPanel groupPanel;
69
70     private JTextField tfMessageWindow = new JTextField();
71     private BorderLayout bL = new BorderLayout();
72
73     private JScrollPane scrollConnectedUsers = new JScrollPane(
74         tpConnectedUsers);
75     private JScrollPane scrollChatWindow = new JScrollPane(
76         cwLobby);
77     private JScrollPane scrollGroupRooms = new JScrollPane(
78         eastPanelCenterNorth);
79
80     private JButton[] groupChatList = new JButton[20];
81     private ArrayList<JCheckBox> arrayListCheckBox = new
82         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 buttons.
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.
135             VERTICAL_SCROLLBAR_AS_NEEDED);
136         scrollChatWindow.setHorizontalScrollBarPolicy(
137             JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
138         scrollConnectedUsers.setVerticalScrollBarPolicy(
139             JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
140         scrollConnectedUsers.setHorizontalScrollBarPolicy(
141             JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
142         DefaultCaret caretConnected = (DefaultCaret)
143             tpConnectedUsers.getCaret();
144         caretConnected.setUpdatePolicy(DefaultCaret.
145             ALWAYS_UPDATE);
146         tpConnectedUsers.setEditable(false);
147
148         tfMessageWindow.setFont(txtFont);
149         StyleConstants.setForeground(chatFont, Color.BLACK);
150         StyleConstants.setBold(chatFont, true);
151     }
152
153     /**
154      * Requests that tfMessageWindow gets focus.
155      */
156     public void focusTextField() {
157         tfMessageWindow.requestFocusInWindow();
158     }
159
160     /**
161      * Initialises listeners.
162      */
163     public void initListeners() {
164         tfMessageWindow.addKeyListener(new EnterListener());
165         GroupListener groupListener = new GroupListener();
166         SendListener sendListener = new SendListener();
167         LobbyListener disconnectListener = new LobbyListener();
168         btnNewGroupChat.addActionListener(groupListener);
169         btnCreateGroup.addActionListener(groupListener);
170         btnLobby.addActionListener(disconnectListener);
171         btnFileChooser.addActionListener(new FileChooserListener
172             ());
173         btnSend.addActionListener(sendListener);
174     }
```

```
168
169  /**
170   * The method takes a ArrayList of the connected users and
171   * sets the user-checkboxes and
172   * the connected user textpane based on the users in the
173   * ArrayList.
174   *
175   * @param connectedUsers The ArrayList of the connected
176   * users.
177   */
178 public void setConnectedUsers( ArrayList<String>
179   connectedUsers ) {
180   setUserText();
181   tpConnectedUsers.setText("");
182   updateCheckBoxes(connectedUsers);
183   for (String ID : connectedUsers) {
184     appendConnectedUsers(ID);
185   }
186 }
187
188 /**
189  * Sets the usertext in the labels to the connected user.
190  */
191 public void setUserText() {
192   lblUser.setText( clientController.getUserID() );
193   lblUser.setFont( txtFont );
194 }
195
196 /**
197  * The south panel in the ClientUI BorderLayout.SOUTH.
198  */
199 public void southPanel() {
200   southPanel.setLayout( new BorderLayout() );
201   southPanel.add( tfMessageWindow, BorderLayout.CENTER );
202   southPanel.setPreferredSize( new Dimension( 600, 50 ) );
203
204   btnSend.setPreferredSize( new Dimension( 134, 40 ) );
205   btnSend.setFont( fontButtons );
206   btnSend.setForeground( new Color( 1, 48, 69 ) );
207   southPanel.add( pnlFileSend, BorderLayout.EAST );
208
209   pnlFileSend.add( btnFileChooser, BorderLayout.WEST );
210   pnlFileSend.add( btnSend, BorderLayout.CENTER );
211
212   add( southPanel, BorderLayout.SOUTH );
213 }
214
215 /**
216  * The east panel in ClientUI BorderLayout.EAST.
217  */
218 public void eastPanel() {
219   eastPanel.setLayout( new BorderLayout() );
220   eastPanel.add( lblUser, BorderLayout.NORTH );
221   eastPanel.add( eastPanelCenter, BorderLayout.CENTER );
222 }
```

```

218         eastPanelCenterNorth.add(pnlGroupSend);
219         eastPanelCenter.add(scrollGroupRooms, BorderLayout.NORTH
220             );
221         eastPanelCenter.add(scrollConnectedUsers, BorderLayout.
222             CENTER);
223
224         pnlGroupSend.add(btnNewGroupChat);
225
226         eastPanel.add(btnLobby, BorderLayout.SOUTH);
227         add(eastPanel, BorderLayout.EAST);
228     }
229
230     /**
231     * Appends the message to the chatwindow object with the ID
232     * of the message object.
233     *
234     * @param message The message object with an ID and a
235     * message.
236     */
237     public void appendContent(Message message) {
238         getChatWindow(message.getConversationID()).append(
239             message);
240         if(activeChatWindow != message.getConversationID()) {
241             highlightGroup(message.getConversationID());
242         }
243     }
244
245     /**
246     * The method handles notice.
247     *
248     * @param ID The ID of the group.
249     */
250     public void highlightGroup(int ID) {
251         if(ID != -1)
252             groupChatList[ID].setBackground(Color.PINK);
253     }
254
255     /**
256     * Appends the string content in the chatwindow-lobby.
257     *
258     * @param content Is a server message
259     */
260     public void appendServerMessage(String content) {
261         cwLobby.append(content.toString());
262     }
263
264     /**
265     * The method updates the ArrayList of checkboxes and add
266     * the checkboxes to the panel.
267     * Also checks if the ID is your own ID and doesn't add a
268     * checkbox of yourself.
269     * Updates the UI.
270     *
271     * @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
282      * connected users.
283      *
284      * @param message Is a username.
285      */
286     public void appendConnectedUsers(String message){
287         StyledDocument doc = tpConnectedUsers.getStyledDocument
288             ();
289         try {
290             doc.insertString(doc.getLength(), message + "\n",
291                 chatFont);
292         } catch (BadLocationException e) {
293             e.printStackTrace();
294         }
295     }
296
297     /**
298      * Sets the text on the groupbuttons to the users you check
299      * in the checkbox.
300      * Adds the new group chat connected with a button and a
301      * ChatWindow.
302      * Enables you to change rooms.
303      * Updates UI.
304      *
305      * @param participants String-Array of the participants of
306      * the new groupchat.
307      * @param ID The ID of the participants of the new groupchat
308      * .
309      */
310     public void createConversation(String[] participants, int ID
311     ) {
312         GroupButtonListener gbListener = new GroupButtonListener
313             ();
314         for (int i = 0; i < participants.length; i++) {
315             if (!(participants[i].equals(clientController.
316                 getUserID())) {
317                 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
369      * object and
370      * returns the correct one based on the ID.
371      *
372      * @param ID The ID of the user.
373      * @return ChatWindow A ChatWindow object with the correct
374      * ID.
375      */
376     public ChatWindow getChatWindow(int ID) {
377         for (ChatWindow cw : arrayListChatWindows) {
378             if (cw.getID() == ID) {
379                 return cw;
380             }
381         }
382         return null;
383     }
384
385     /**
386      * The class extends Thread and handles the Create a group
387      * panel.
388      */
389     private class GroupPanel extends Thread {
390         private JFrame groupFrame;
391         private JPanel pnlOuterBorderLayout = new JPanel(new
392             BorderLayout());
393         private JPanel pnlNewGroup = new JPanel();
394         private JScrollPane scrollCheckConnectedUsers = new
395             JScrollPane(pnlNewGroup);
396
397         /**
398          * The metod returns the JFrame groupFrame.
399          *
400          * @return groupFrame
401          */
402         public JFrame getFrame() {
403             return groupFrame;
404         }
405
406         /**
407          * Runs the frames of the groupPanels.
408          */
409         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
416  * groupPanel.
417  */
418 public void panelBuilder() {
419     scrollCheckConnectedUsers.setVerticalScrollBarPolicy
420         (JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED);
421     scrollCheckConnectedUsers.
422         setHorizontalScrollBarPolicy(JScrollPane.
423             HORIZONTAL_SCROLLBAR_NEVER);
424     btnCreateGroup.setText("New Conversation");
425     pnlOuterBorderLayout.add(btnCreateGroup,
426         BorderLayout.SOUTH);
427     pnlOuterBorderLayout.add(scrollCheckConnectedUsers,
428         BorderLayout.CENTER);
429     scrollCheckConnectedUsers.setPreferredSize(new
430         Dimension(200,500));
431     pnlNewGroup.setLayout(new GridLayout(100,1,5,5));
432 }
433
434 /**
435  * KeyListener for the messagewindow.
436  * Enables you to send a message with enter.
437  */
438 private class EnterListener implements KeyListener {
439     public void keyPressed(KeyEvent e) {
440         if (e.getKeyCode() == KeyEvent.VK_ENTER && !(
441             tfMessageWindow.getText().isEmpty())) {
442             clientController.sendMessage(
443                 activeChatWindow, tfMessageWindow.getText
444                 ());
445             tfMessageWindow.setText("");
446         }
447     }
448     public void keyReleased(KeyEvent e) {}
449     public void keyTyped(KeyEvent e) {}
450 }
451
452 /**
453  * Listener that listens to New Group Chat-button and the
454  * Create Group Chat-button.
```

```

447     * If create group is pressed, a new button will be created
448     * with the right name,
449     * the right participants.
450     * The method use alot of ArrayLists of checkboxes,
451     * participants and strings.
452     * Also some error-handling with empty buttons.
453     */
454     private class GroupListener implements ActionListener {
455         private ArrayList<String> participants = new ArrayList<
456             String>();
457         private String [] temp;
458         public void actionPerformed(ActionEvent e) {
459             if (btnNewGroupChat == e.getSource() &&
460                 arrayListCheckBox.size() > 0) {
461                 groupPanel.getFrame().setVisible(true);
462             }
463             if (btnCreateGroup == e.getSource()) {
464                 participants.clear();
465                 temp = null;
466                 for (int i = 0; i < arrayListCheckBox.size(); i
467                     ++){
468                     if (arrayListCheckBox.get(i).isSelected()) {
469                         participants.add(arrayListCheckBox.get(i)
470                             .getText());
471                     }
472                 }
473                 temp = new String[participants.size() + 1];
474                 temp[0] = clientController.getUserID();
475                 for (int i = 1; i <= participants.size(); i++) {
476                     temp[i] = participants.get(i-1);
477                 }
478                 if (temp.length > 1) {
479                     clientController.sendParticipants(temp);
480                     groupPanel.getFrame().dispose();
481                     createdGroup = true;
482                 } else {
483                     JOptionPane.showMessageDialog(null, "You
484                         have to choose atleast one person!");
485                 }
486             }
487         }
488     }
489
490     /**
491     * Listener that connects the right GroupChatButton in an
492     * ArrayList to the right
493     * active chat window.
494     * Updates the UI.
495     */
496     private class GroupButtonListener implements ActionListener
497     {
498         public void actionPerformed(ActionEvent e) {
499             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].
498                     setBackground(null);
499             }
500             groupChatList[i].setBackground(new Color
501                 (201, 201, 201));
502             remove(bL.getLayoutComponent(BorderLayout.
503                 CENTER));
504             add(getChatWindow(i), BorderLayout.CENTER);
505             activeChatWindow = i;
506             validate();
507             repaint();
508         }
509     }
510
511     /**
512     * Listener that connects the user with the lobby chatWindow
513     * through the Lobby button.
514     * Updates UI.
515     */
516     private class LobbyListener implements ActionListener {
517         public void actionPerformed(ActionEvent e) {
518             if (btnLobby == e.getSource()) {
519                 btnLobby.setBackground(new Color(201, 201, 201));
520                 if (activeChatWindow != -1)
521                     groupChatList[activeChatWindow].
522                         setBackground(null);
523                 remove(bL.getLayoutComponent(BorderLayout.CENTER
524                     ));
525                 add(getChatWindow(-1), BorderLayout.CENTER);
526                 activeChatWindow = -1;
527                 invalidate();
528                 repaint();
529             }
530         }
531     }
532
533     /**
534     * Listener that creates a JFileChooser when the button
535     * btnFileChooser is pressed.
536     * The JFileChooser is for images in the chat and it calls
537     * the method sendImage in the controller.
538     */
539     private class FileChooserListener implements ActionListener
540     {
541         public void actionPerformed(ActionEvent e) {
542             if (btnFileChooser == e.getSource()) {
543                 JFileChooser fileChooser = new JFileChooser();

```

```
537         int returnValue = fileChooser.showOpenDialog(  
538             null);  
539         if (returnValue == JFileChooser.APPROVE_OPTION)  
540         {  
541             File selectedFile = fileChooser.  
542                 getSelectedFile();  
543             String fullPath = selectedFile.  
544                 getAbsolutePath();  
545             clientController.sendMessage(activeChatWindow,  
546                 fullPath);  
547         }  
548     }  
549 }  
550  
551 /**  
552  * Listener for the send message button.  
553  * Resets the message textfield text.  
554  */  
555 private class SendListener implements ActionListener {  
556     public void actionPerformed(ActionEvent e) {  
557         if (btnSend == e.getSource() && !(tfMessageWindow.  
558             getText().isEmpty())) {  
559             clientController.sendMessage(  
560                 activeChatWindow, tfMessageWindow.getText()  
561                 ());  
562             tfMessageWindow.setText("");  
563         }  
564     }  
565 }
```

Listing 6: ClientUI

7.2.5 ImageScaleHandler.java

```
1 package chat;  
2  
3 import java.awt.Graphics2D;  
4 import java.awt.Image;  
5 import java.awt.image.BufferedImage;  
6  
7 import javax.swing.ImageIcon;  
8 import javax.swing.JFrame;  
9 import javax.swing.JLabel;  
10 import javax.swing.JPanel;  
11  
12 import org.imgscalr.Scalr;  
13 import org.imgscalr.Scalr.Method;  
14  
15 /**  
16  * Scales down images to preferred size.
```

```
17  *
18  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
19  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
20  */
21  public class ImageScaleHandler {
22
23      private static BufferedImage toBufferedImage(Image img) {
24          if (img instanceof BufferedImage) {
25              return (BufferedImage) img;
26          }
27          BufferedImage bimage = new BufferedImage(img.getWidth(
28              null),
29              img.getHeight(null), BufferedImage.TYPE_INT_ARGB
30              );
31          Graphics2D bGr = bimage.createGraphics();
32          bGr.drawImage(img, 0, 0, null);
33          bGr.dispose();
34          return bimage;
35      }
36
37      public static BufferedImage createScaledImage(Image img, int
38          height) {
39          BufferedImage bimage = toBufferedImage(img);
40          bimage = Scalr.resize(bimage, Method.ULTRA_QUALITY,
41              Scalr.Mode.FIT_TO_HEIGHT, 0, height);
42          return bimage;
43      }
44
45      // Example
46      public static void main(String[] args) {
47          ImageIcon icon = new ImageIcon("src/filer/new1.jpg");
48          Image img = icon.getImage();
49
50          // Use this to scale images
51          BufferedImage scaledImage = ImageScaleHandler.
52              createScaledImage(img, 75);
53          icon = new ImageIcon(scaledImage);
54
55          JLabel lbl = new JLabel();
56          lbl.setIcon(icon);
57          JPanel panel = new JPanel();
58          panel.add(lbl);
59          JFrame frame = new JFrame();
60          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
61          frame.add(panel);
62          frame.pack();
63          frame.setVisible(true);
64      }
65  }
```

Listing 7: ImageScaleHandler

7.2.6 StartClient.java

```
1 package chat;
2
3 import java.awt.BorderLayout;
4 import java.awt.Color;
5 import java.awt.Dimension;
6 import java.awt.FlowLayout;
7 import java.awt.Font;
8 import java.awt.GridLayout;
9 import java.awt.event.ActionEvent;
10 import java.awt.event.ActionListener;
11
12 import javax.swing.*;
13
14 /**
15  * Log in UI and start-class for the chat.
16  *
17  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
18  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson.
19  */
20 public class StartClient extends JPanel {
21     private JLabel lblIp = new JLabel("IP:");
22     private JLabel lblPort = new JLabel("Port:");
23     private JLabel lblWelcomeText = new JLabel("Log in to BIRC")
24     ;
25     private JLabel lblUserName = new JLabel("Username:");
26
27     private JTextField txtIp = new JTextField("localhost");
28     private JTextField txtPort = new JTextField("3450");
29     private JTextField txtUserName = new JTextField();
30
31     private JButton btnLogIn = new JButton("Login");
32     private JButton btnCancel = new JButton("Cancel");
33
34     private Font fontWelcome = new Font("Sans-Serif",Font.BOLD
35     ,20);
36     private Font fontIpPort = new Font("Sans-Serif",Font.PLAIN
37     ,17);
38     private Font fontButtons = new Font("Sans-Serif", Font.BOLD,
39     15);
40     private Font fontInfo = new Font("Sans-Serif",Font.BOLD|Font
41     .ITALIC,17);
42
43     private BorderLayout borderLayout = new BorderLayout();
44     private JPanel pnlCenterGrid = new JPanel(new GridLayout
45     (3,2,5,5));
46     private JPanel pnlCenterFlow = new JPanel(new FlowLayout());
47     private JPanel pnlNorthGrid = new JPanel(new GridLayout
48     (2,1,5,5));
49     private JPanel pnlNorthGridGrid = new JPanel(new GridLayout
50     (1,2,5,5));
51     private JFrame frame;
```



```
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.
121                 getSystemLookAndFeelClassName());
122         } catch (ClassNotFoundException e) {
123             e.printStackTrace();
124         } catch (InstantiationException e) {
125             e.printStackTrace();
126         } catch (IllegalAccessException e) {
127             e.printStackTrace();
128         } catch (UnsupportedLookAndFeelException e) {
129             e.printStackTrace();
130         }
131     }
132
133     /**
134      * Run method for the login-frame.
135      */
136     public static void main(String[] args) {
137         SwingUtilities.invokeLater(new Runnable() {
138             @Override
139             public void run() {
140                 JFrame frame = new JFrame("bIRC Login");
141                 StartClient ui = new StartClient();
142                 frame.setDefaultCloseOperation(JFrame.
143                     DISPOSE_ON_CLOSE);
144                 frame.add(ui);
145                 frame.pack();
146                 frame.setVisible(true);
147                 frame.setLocationRelativeTo(null);
148                 frame.setResizable(false);
149             }
150         });
151     }
```

```
151
152
153
154  /**
155   * Listener for login-button, create server-button and for
156   * the cancel-button.
157   * Also limits the username to a 10 char max.
158   */
159  private class LogInMenuListener implements ActionListener {
160      public void actionPerformed(ActionEvent e) {
161          if (btnLogIn==e.getSource()) {
162              if (txtUserName.getText().length() <= 10) {
163                  new Client(txtIp.getText(), Integer.
164                      parseInt(txtPort.getText()),
165                      txtUserName.getText());
166              } else {
167                  JOptionPane.showMessageDialog(null, "Namnet
168                      får max vara 10 karaktärer!");
169                  txtUserName.setText("");
170              }
171          }
172          if (btnCancel==e.getSource()) {
173              System.exit(0);
174          }
175      }
176  }
177
178  /**
179   * Listener for the textField. Enables you to press enter
180   * 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
187                  (txtPort.getText()),txtUserName.getText()
188                  );
189          } else {
190              JOptionPane.showMessageDialog(null, "Namnet får
191                  max vara 10 karaktärer!");
192              txtUserName.setText("");
193          }
194      }
195  }
196
197  /**
198   * Listener for textfield in create a server. Enables you to
199   * press enter to create server.
200   * Disposes the serverpanel on create.
201   */
202  /**
```

```
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 {
15     private LinkedList<Message> list = new LinkedList<Message>()
16     ;
17     private static int MESSAGE_LIMIT = 30;
18     private static final long serialVersionUID =
19         13371337133732526L;
20
21     /**
22     * Adds a new message to the chat log.
23     *
24     * @param message The message to be added.
25     */
26     public void add(Message message) {
27         if (list.size() >= MESSAGE_LIMIT) {
28             list.removeLast();
29         }
30         list.add(message);
31     }
32
33     public Iterator<Message> iterator() {
34         return list.iterator();
35     }
36 }
```

34 }

Listing 9: ChatLog

7.3.2 Message

```
1 package chat;
2
3 import java.io.Serializable;
4 import java.text.SimpleDateFormat;
5 import java.util.Date;
6
7 /**
8  * Model class to handle messages
9  *
10 * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
11 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
12 */
13 public class Message implements Serializable {
14     private String fromUserID;
15     private Object content;
16     private String timestamp;
17     private int conversationID = -1;    /* -1 means it's a lobby
18         message */
19     private static final long serialVersionUID = 133713371337L;
20
21     /**
22      * Constructor that creates a new message with given
23      * conversation ID, String with information who sent it,
24      * and its content.
25      *
26      * @param conversationID The conversation ID.
27      * @param fromUserID A string with information who sent the
28      * message.
29      * @param content The message's content.
30      */
31     public Message(int conversationID, String fromUserID, Object
32         content) {
33         this.conversationID = conversationID;
34         this.fromUserID = fromUserID;
35         this.content = content;
36         newTime();
37     }
38
39     /**
40      * Creates a new timestamp for the message.
41      */
42     private void newTime() {
43         Date time = new Date();
44         SimpleDateFormat ft = new SimpleDateFormat("HH:mm:ss");
45         this.timestamp = ft.format(time);
46     }
47 }
```

```
42
43  /**
44   * Returns a string containing sender ID.
45   *
46   * @return A string with the sender ID.
47   */
48  public String getFromUserID() {
49      return fromUserID;
50  }
51
52  /**
53   * Returns an int with the conversation ID.
54   *
55   * @return An int with the conversation ID.
56   */
57  public int getConversationID() {
58      return conversationID;
59  }
60
61  /**
62   * Returns the message's timestamp.
63   *
64   * @return The message's timestamp.
65   */
66  public String getTimestamp() {
67      return this.timestamp;
68  }
69
70  /**
71   * Returns the message's content.
72   *
73   * @return The message's content.
74   */
75  public Object getContent() {
76      return content;
77  }
78 }
```

Listing 10: Message

7.3.3 User

```
1 package chat;
2
3 import java.io.Serializable;
4 import java.util.ArrayList;
5
6 /**
7  * Class to hold information of a user.
8  *
9  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
10  * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
```

```
11  */
12  public class User implements Serializable {
13      private static final long serialVersionUID = 1273274782824L;
14      private ArrayList<Conversation> conversations;
15      private String id;
16
17      /**
18       * Constructor to create a User with given ID.
19       *
20       * @param id A string with the user ID.
21       */
22      public User(String id) {
23          this.id = id;
24          conversations = new ArrayList<>();
25      }
26
27      /**
28       * Returns an ArrayList with the user's conversations
29       *
30       * @return The user's conversations.
31       */
32      public ArrayList<Conversation> getConversations() {
33          return conversations;
34      }
35
36      /**
37       * Adds a new conversation to the user.
38       *
39       * @param conversation The conversation to be added.
40       */
41      public void addConversation(Conversation conversation) {
42          conversations.add(conversation);
43      }
44
45      /**
46       * Returns the user's ID.
47       *
48       * @return The user's ID.
49       */
50      public String getId() {
51          return id;
52      }
53  }
```

Listing 11: User

7.3.4 Conversation

```
1  package chat;
2
3  import java.io.Serializable;
4  import java.util.HashSet;
```

```
5
6 /**
7  * Class to hold information of a conversation.
8  *
9  * @author Emil Sandgren, Kalle Bornemark, Erik Sandgren,
10 * Jimmy Maksymiw, Lorenz Puskas & Rasmus Andersson
11 */
12 public class Conversation implements Serializable {
13     private HashSet<String> involvedUsers;
14     private ChatLog conversationLog;
15     private int id;
16     private static int numberOfConversations = 0;
17
18     /**
19     * Constructor that takes a HashSet of involved users.
20     *
21     * @param involvedUsersID The user ID's to be added to the
22     * conversation.
23     */
24     public Conversation(HashSet<String> involvedUsersID) {
25         this.involvedUsers = involvedUsersID;
26         this.conversationLog = new ChatLog();
27         id = ++numberOfConversations;
28     }
29
30     /**
31     * Returns a HashSet of the conversation's involved users.
32     *
33     * @return A HashSet of the conversation's involved users.
34     */
35     public HashSet<String> getInvolvedUsers() {
36         return involvedUsers;
37     }
38
39     /**
40     * Returns the conversation's ChatLog.
41     *
42     * @return The conversation's ChatLog.
43     */
44     public ChatLog getConversationLog() {
45         return conversationLog;
46     }
47
48     /**
49     * Adds a message to the conversation.
50     *
51     * @param message The message to be added.
52     */
53     public void addMessage(Message message) {
54         conversationLog.add(message);
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