Cinema Booking System

Project Report

COMP20081 – Systems Software

Table of Contents

1 Objectives 3
2 Analysis of the System 4
2.1 Documentation and revision control 7
3 Commented listings of the client and server 9
3.1 Commented Administrator Interface Code 9
3.2 Commented Client Code 13
3.3 Commented Default class 32
3.4 Commented Server code 32
3.5 Commented Shared code 53
4 Explanation of the System Design 62
4.1 Nature of the Network Connection 62
4.2 Server Data Structure 62
4.3 Client Server asynchronous communication 63
4.4 Additional Features and Enhancements 63
4.4.1 Graceful Quit 63
4.4.2 Server Log File 63
4.4.3 Graphical User Interface 64
4.4.4 Administration facilities 64
4.4.5 Amend Bookings 64
6 References 66
7 Appendices 67

1 Objectives

The Aim of this assignment was to implement an on-line service booking system. The topic of the booking system was left to the discretion of the students creating it. After many considerations the decision was to make a cinema reservation system. It was also specified that the system must implement the following guidelines:

1. The system must use client server communications
2. It must manage the bookings made by the users of the system, without the use of a database.
3. It should allow users to request the available positions for a given date.
4. It should allow users to make a reservation for a given date.
5. It should allow users to cancel a booking they have made.
6. It must allow users to download currently active messages, such as special offers.
7. The system must either use Datagram Socket or Stream Socket technology.
8. The system must implement a server that supports multiple clients.

The system has two main components the server and the client; these elements also had a required level of functionality:

1. The server should keep a record of all registered users and their IP addresses.
2. The server should accept new user registrations and add them to a user list.
3. The server should accept requests from the client to log off.
4. The server must read and write reservations to and from a text file.
5. The server should receive messages.
6. The server should report free bookings for a given data, reserve a ticket, cancel a booking and send special offer messages.
7. The server should handle multiple clients concurrently
8. The server should maintain a list of available tickets for all dates available in the system.
9. The client must connect to the server and register its IP address as a valid user along with the users name
10. The client must be able to logoff from the server
11. The client must have the ability to send a message to the server specifying information on the number of bookings available for a given event, make a reservation or cancel a reservation and request information on special offers.
12. The client must asynchronously receive and display urgent messages from the server.

2 Analysis of the System

To analyse the system, preliminary flow charts were created to show a conceptual view of the system. Key concepts of the system are visualised in the flowcharts below. These diagrams were created to ensure that all aspects of the system had been considered in a logical manner before designing the system.

Client Flowchart

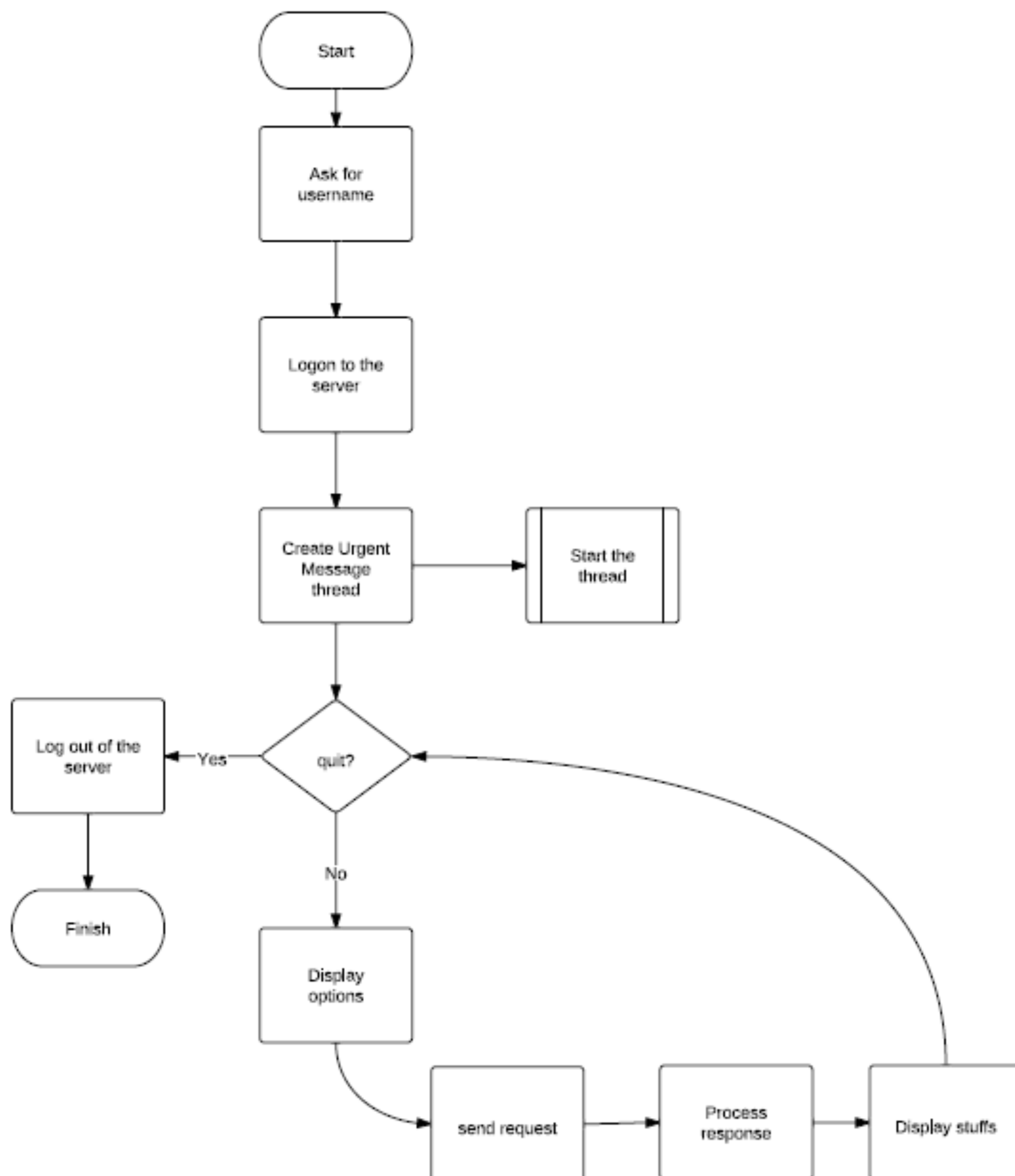


Figure 1. Client Flowchart

Command Line Thread Flowchart

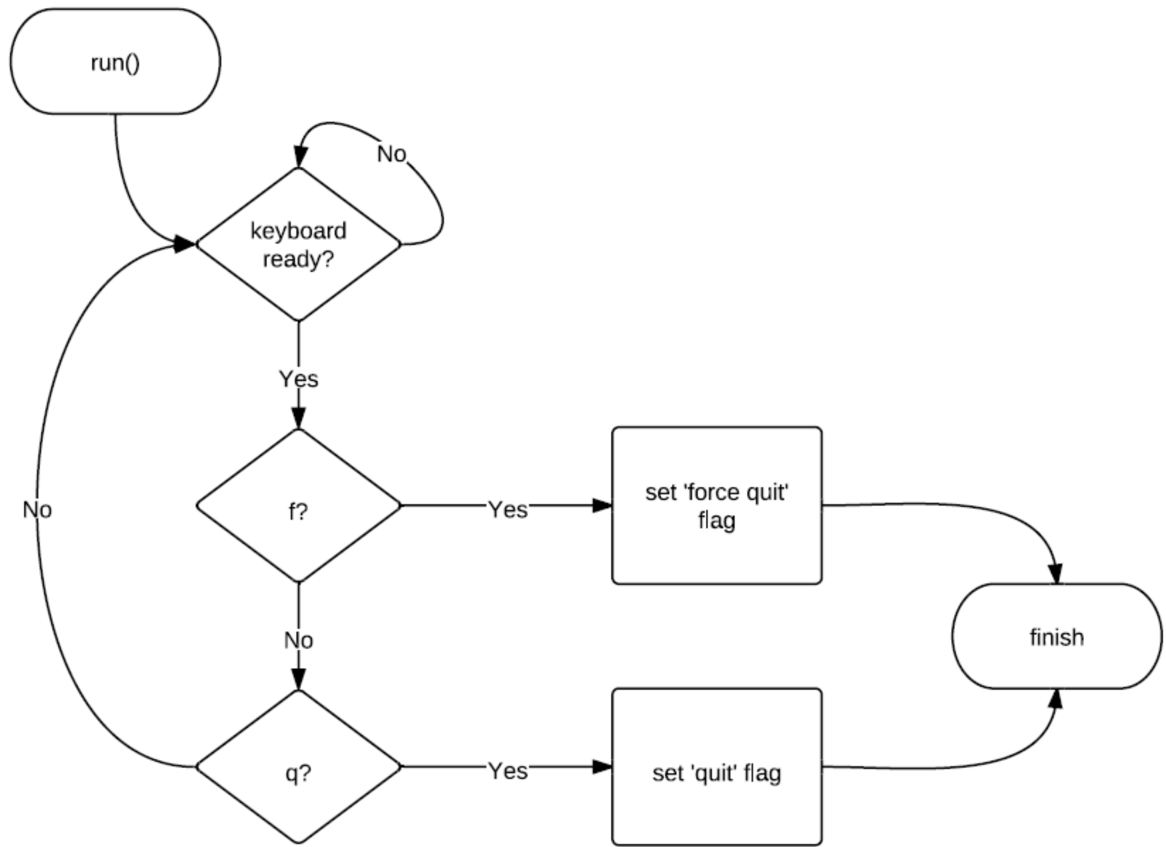


Figure 2. Command Line Thread Flowchart

Server Client Thread Flowchart

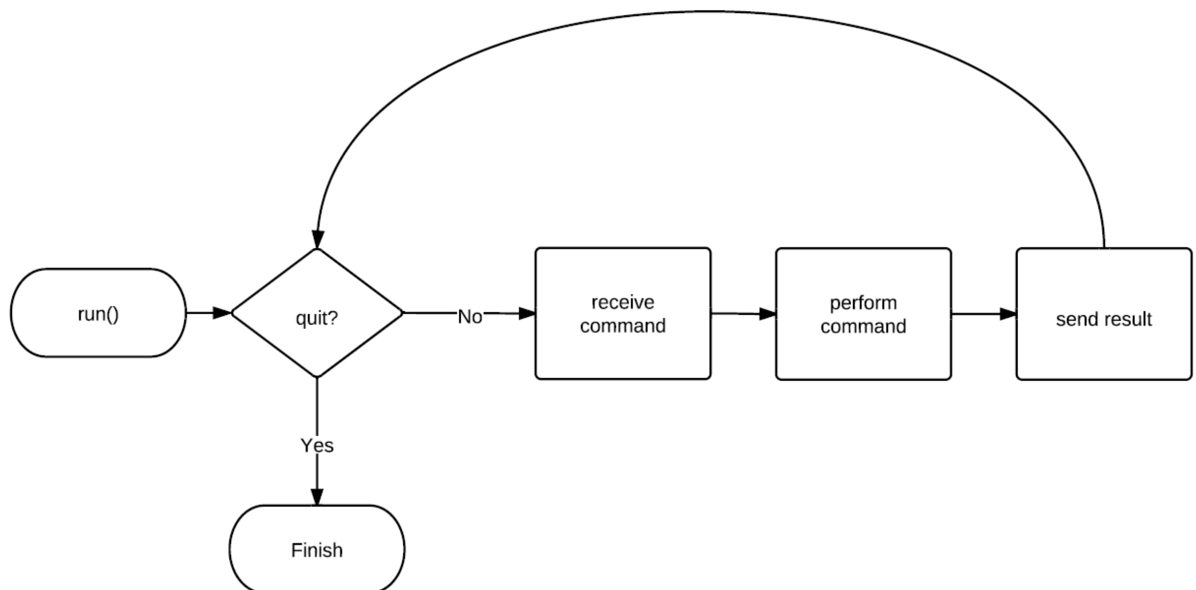


Figure 3. Server Client Thread Flowchart

Server Flowchart

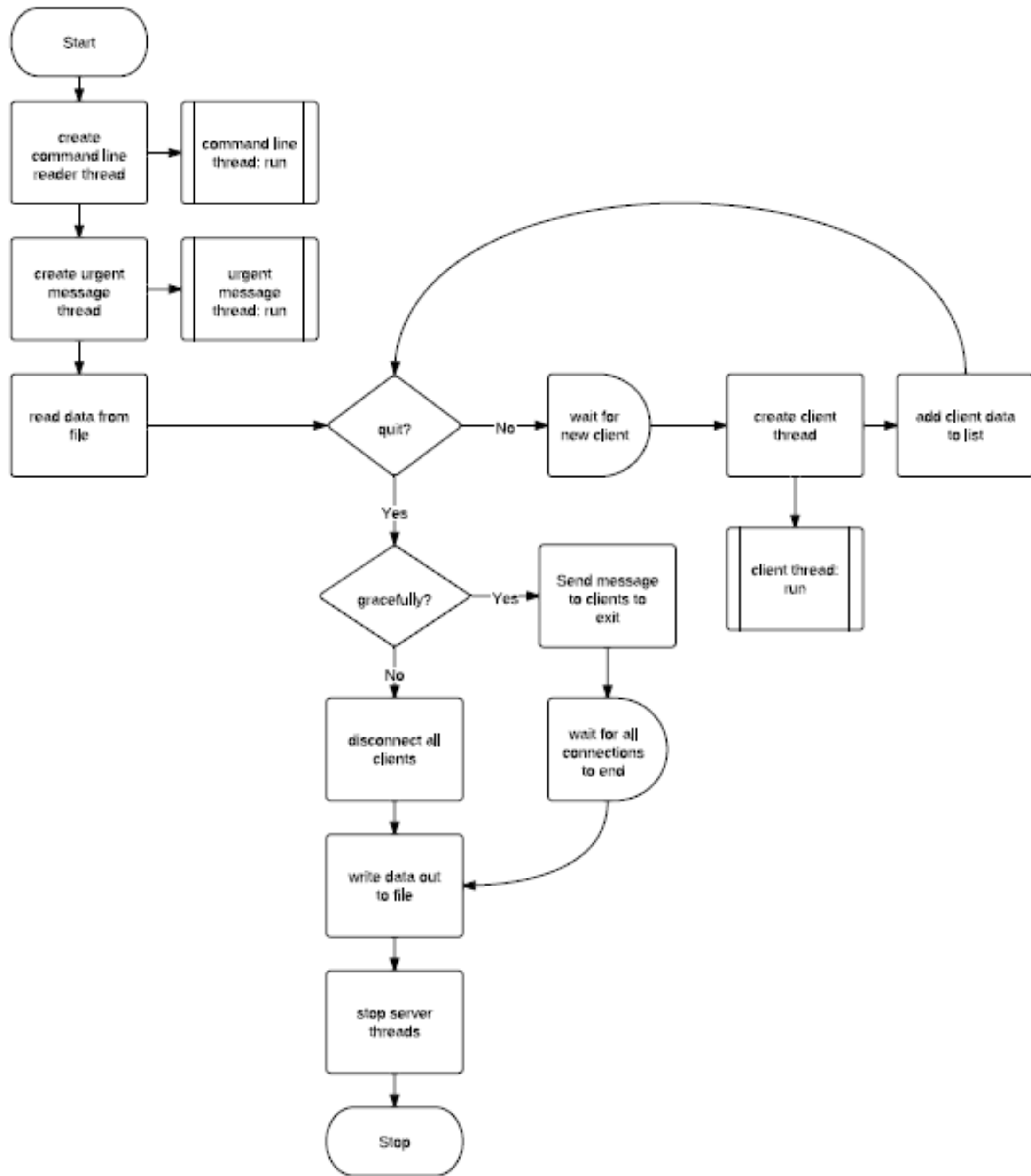


Figure 4. Server Flowchart

Urgent Message Thread Flowchart

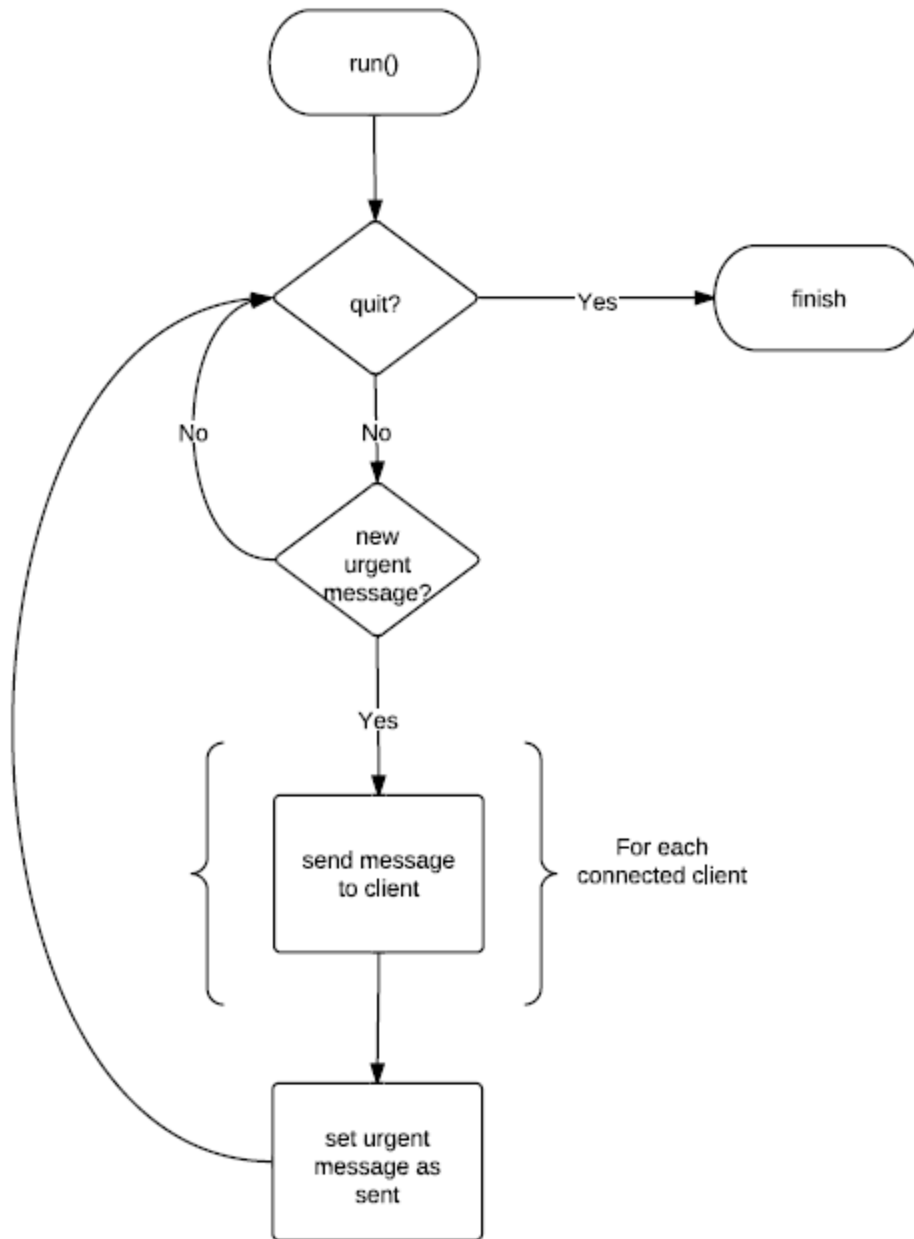


Figure 5. Urgent Message Thread Flowchart

2.1 Documentation and Revision Control

JavaDocs were created from comments in the code to document the methods and classes (Read, N and Shippey, R. 2012). This outlines each class and method's purpose and a description of how it is implemented along with JavaDocs standard parameters and return types. Using this form of documentation provides a high level explanation of the actual implementation for ease of use. Collaboration on the code was aided by having access to JavaDocs for previously written for other methods.

After analysing the conceptual view of the system, it became apparent that revision control would be needed to keep edits and versions of the code. The version control system, GitHub, has been used to keep all changes made to the project on a centralised server so that the code can be accessed from anywhere, will be backed up, and branch-able (GitHub. 2012). The projects repository (Read, N and Shippey, R. 2012) was private during development but is now publically available for the convenience of the reader. A guide was produced to assist in using the Git tools available, shown in Appendix A.

3 Commented Code Listings

3.1 Commented Administrator Interface Code

```
1 /*
2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5 package Admin;
6
7 import Server.Server;
8 import java.io.BufferedReader;
9 import java.io.File;
10 import java.io.FileInputStream;
11 import java.io.FileOutputStream;
12 import java.io.IOException;
13 import java.io.InputStreamReader;
14 import java.io.ObjectInputStream;
15 import java.io.ObjectOutputStream;
16 import java.net.Socket;
17 import java.util.Arrays;
18 import java.util.Iterator;
19 import java.util.LinkedList;
20 import shared.Request;
21
22 /**
23  * The interface class.
24  * @author Robert
25  */
26 public class Admin extends Thread {
27
28     private File filmFile;
29     private LinkedList<String> films;
30     private Socket s;
31     private ObjectOutputStream oos;
32     private ObjectInputStream ois;
33     private BufferedReader cmd;
34
35     /**
36      * Sets up the file and command line BufferedReader
37      */
38     public Admin() {
39         filmFile = new File("data" + File.separator + "films.txt");
40         cmd = new BufferedReader(new InputStreamReader(System.in));
41     }
42
43     /**
44      * Waits for the server to shut down, then presents the user with
           the menu and accepts input to carry out each task.
45      */
46     @Override
47     public void run() {
48         System.out.println("Waiting for the server to shutdown...");
49         waitForServer();
50         System.out.println("Server is shut down. Safe to edit text
                                    files.");
51         readData();
52
```

```
53          while (true) {
54
55              System.out.println("");
56              System.out.println("0) Add a film");
57              System.out.println("1) Remove a film");
58              System.out.println("2) View all films");
59              System.out.println("Type quit to exit");
60
61              String command = "quit";
62              try {
63                  command = cmd.readLine();
64              } catch (IOException ioe) {
65              }
66
67
68              if (command.equals("quit")) {
69                  break;
70              }
71
72              int option = -1;
73              try {
74                  option = Integer.parseInt(command);
75              } catch (Exception e) {
76                  System.err.println("Command not recognised. Try
                                        again.");
77                  continue;
78              }
79
80              int i;
81
82              switch (option) {
83                  case 0:
84                      //<editor-fold desc="Add a film">
85                      String film = null;
86                      try {
87                          System.out.println("Enter the film name:");
88                          String filmName = cmd.readLine();
89                          System.out.println("Enter the film date: (DD-
                                               MM-YYYY");
90                          String filmDate = cmd.readLine();
91                          System.out.println("Enter the film time:
                                               (HH:MM)");
92                          String filmTime = cmd.readLine();
93                          System.out.println("Enter the films
                                               capacity:");
94                          String filmCapacity = cmd.readLine();
95
96                          film = filmName + "," + filmDate + "," +
                                   filmTime + "," + filmCapacity + "," +
                                   "0";
97                      } catch (Exception e) {
98                          System.err.println("Something very bad
                                 happened. Your data wasn't saved.");
99                          System.err.println("Please try again.");
100                         continue;
101                     }
102                     if (film != null) {
103                         films.add(film);
104                     }
105                     System.out.println("Item sucessfully added!");
106
```

```
107                         break;
108                         //</editor-fold>
109                     case 1:
110                         //<editor-fold desc="Remove a film">
111                         System.out.println("Please select a film to
                                delete");
112                         Iterator<String> it = films.iterator();
113                         i = 0;
114                         while (it.hasNext()) {
115                             System.out.println((i++) + ") " + it.next());
116                         }
117                         try {
118                             int item = Integer.parseInt(cmd.readLine());
119                             films.remove(item);
120                         } catch (Exception e) {
121                             System.err.println("Something very bad
                                    happened. Your data wasn't saved.");
122                             System.err.println("Please try again.");
123                         }
124                         System.out.println("Item successfully removed!");
125 
126                         break;
127                         //</editor-fold>
128                     case 2:
129                         System.out.println("All films: ");
130                         Iterator<String> itt = films.iterator();
131                         i = 0;
132                         while (itt.hasNext()) {
133                             System.out.println((i++) + ") " + itt.next());
134                         }
135                         break;
136                     default:
137                         System.err.println("Command not recognised. Try
                                again.");
138                         continue;
139                 }
140             }
141             writeData();
142         }
143 
144         /**
145          * Hooks up a socket to the server, if it fails then the server is
              down and it will break, otherwise it will carry on looping.
146          */
147         private void waitForServer() {
148             while (true) {
149                 try {
150                     s = new Socket(Editor.server, 2000);
151                     oos = new ObjectOutputStream(s.getOutputStream());
152                     ois = new ObjectInputStream(s.getInputStream());
153 
154                     oos.writeObject("ADMINISTRATOR");
155                     ois.readObject();
156 
157                     oos.writeObject(new Request(Request.LOG_OFF));
158 
159                     s.close();
160                 } catch (Exception e) {
161                     break;
162                 }
163             }
```

```
164     }
165
166
167     /**
168      * Reads the data from the file into the linked list
169      */
170     private void readData() {
171         films = new LinkedList<String>();
172         try {
173             if (filmFile.exists()) {
174                 FileInputStream ff = new FileInputStream(filmFile);
175                 byte[] fileBytes = new byte[ff.available()];
176                 ff.read(fileBytes);
177                 String filmsString = new String(fileBytes);
178                 String[] film = filmsString.split(Server.endLine);
179                 if (!film[0].equals("")) {
180                     films.addAll(Arrays.asList(film));
181                 } else {
182                     films = null;
183                 }
184             }
185         } catch (IOException ioe) {
186             films = null;
187             System.err.println("Could not read data from
                                   data/films.txt");
188         }
189     }
190
191     /**
192      * Iterates over the linked list and writes the data out to the
           file.
193      */
194     private void writeData() {
195         try {
196             filmFile.getParentFile().mkdirs();
197             filmFile.createNewFile();
198         } catch (Exception e) {
199         }
200         try {
201             FileOutputStream pw = new FileOutputStream(filmFile);
202             Iterator<String> it = films.iterator();
203             while (it.hasNext()) {
204                 pw.write((it.next() + "\r\n").getBytes());
205             }
206             pw.flush();
207         } catch (Exception e) {
208             System.err.println("Could not save data.");
209         }
210     }
211 }
212
```

```
1 /*
2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5 package Admin;
6
7 /**
8  * Class to hold the entry point for the Administrator interface.
9  * @author Robert
```

```
10  */
11 public class Editor {
12 
13     /**
14      * Host name of the server.
15      */
16     public static String server;
17 
18     /**
19      * Entry point of the Administrator interface. Checks that one
        argument has been passed in.
20      * @param args the host of the server
21      */
22     public static void main(String[] args) {
23         if (args.length == 1) {
24             server = args[0];
25             Admin a = new Admin();
26             a.start();
27         } else {
28             System.err.println("Error. Usage: java -cp Task2.jar
                                    Admin.Editor <host>");
29             System.exit(0);
30         }
31     }
32 }
33 
```

3.2 Commented Client Code

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package Client;
 6 
 7 /**
 8  * Class to hold the main function.
 9  * @author Robert and Nathan
10  */
11 public class Client {
12     private static String Server = "localhost";
13     /**
14      * Get the IP address or hostname of the server, as passed it via
        args.
15      * @return a String representation of the server's location
16      */
17     public static String getServer(){ return Server;}
18     /**
19      * Entry point into the client.
20      * Pass in the IP address or host name of the server as the first
        and only arg
21      * @param arg location of the server
22      */
23     public static void main(String arg[]) {
24         if(arg.length == 1){
25             Client.Server = arg[0];
26             Login frame = new Login();
27         } else {
28             System.err.println("Error. Argument should be IP
                                address/host name of the server");
```

```
29          }
30
31      }
32 }
33
```

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package Client;
 6
 7 import java.io.IOException;
 8 import java.io.ObjectInputStream;
 9 import java.io.ObjectOutputStream;
10 import java.net.Socket;
11 import javax.swing.JOptionPane;
12 import shared.Booking;
13 import shared.Request;
14 import shared.Response;
15
16 /**
17  * The clients communication with the server
18  * @author Robert
19  */
20 public class Comms {
21
22     private Socket server;
23     private ObjectInputStream in;
24     private ObjectOutputStream out;
25     private String user;
26     private Booking[] reservations;
27
28     /**
29      * Create a new session with the server. Uses the argument passed
          in as the address to connect to.
30      * Sets the ObjectStream too.
31      * @throws IOException can't connect to the server
32      */
33     public Comms() throws IOException {
34         server = new Socket(Client.getServer(), 2000);
35         server.setSoTimeout(0);
36         in = new ObjectInputStream(server.getInputStream());
37         out = new ObjectOutputStream(server.getOutputStream());
38     }
39
40     /**
41      * Log on to the server and get back the users reservations.
42      * @param name the users name
43      */
44     public void logon(String name) {
45         user = name;
46         try {
47             out.writeObject(user);
48             Object[] r = (Object[]) in.readObject();
49             if (r != null) {
50                 Booking[] a = new Booking[r.length];
51                 int i = 0;
52                 for (Object o : r) {
53                     a[i++] = (Booking) o;
54                 }
```

```
55
56                  reservations = a;
57              }
58          } catch (ClassNotFoundException cnf) {
59          } catch (IOException ioe) {
60          }
61      }
62
63      /**
64       * Sends Log off to the server to finish the session, then close
           the socket.
65       */
66      public void logoff() {
67          try {
68              out.writeObject(new Request(Request.LOG_OFF));
69              server.close();
70          } catch (IOException e) {
71              System.err.println(e.getMessage());
72          }
73      }
74
75      /**
76       * Get how long the reservations array is
77       * @return reservations.length
78       */
79      public int getReservationsLength() {
80          if (reservations == null) {
81              return 0;
82          }
83          return reservations.length;
84      }
85
86      /**
87       * Get a specific reservation
88       * @param i the index of the reservation
89       * @return the reservation indexed by i
90       * @throws ArrayIndexOutOfBoundsException
91       */
92      public Booking getReservation(final int i) throws
              ArrayIndexOutOfBoundsException {
93          if (i >= reservations.length || i < 0) {
94              throw new ArrayIndexOutOfBoundsException("Out of bounds.
                                             Use getReservationLength.");
95          } else {
96              return reservations[i];
97          }
98      }
99
100     /**
101      * Get local or remote reservations that are owned by this user and
           format them in a comma separated format.
102      * @param refresh use true to get new data from the server, false
           to use local cache
103      * @return all reservations as Strings.
104      */
105     public String[] getAllReservationsAsStrings(boolean refresh) {
106         if (refresh) {
107             Request r = new Request(Request.MY_RESERVATIONS);
108             Response response = sendRequest(r);
109             Object[] objs = response.getResponseObjects();
110             if (objs == null) {
```

```
111                 String[] n = {""};
112                 return n;
113             }
114             Booking[] bookings = new Booking[objs.length];
115             for (int x = 0; x < objs.length; x++) {
116                 bookings[x] = (Booking) objs[x];
117             }
118             reservations = bookings;
119         }
120         return getAllReservationsAsStrings();
121     }
122 
123     /**
124      * Get local reservations formatted as comma separated Strings
125      * @return all reservations as Strings
126      */
127     public String[] getAllReservationsAsStrings() {
128         if (reservations == null) {
129             String[] r = {""};
130             return r;
131         }
132         String[] r = new String[reservations.length];
133         for (int x = 0; x < r.length; x++) {
134             r[x] = reservations[x].getFilm().getName() + ", "
135                     + reservations[x].getFilm().getDate() + ", "
136                     + reservations[x].getFilm().getTime() + ", "
137                     + reservations[x].getSeats();
138         }
139         return r;
140     }
141 
142     /**
143      * Sends a Request object to the server. Ensure that it was
         constructed with a static string from Request.
144      * This will return null if any exception is caught.
145      * @param r the request
146      * @return the Response object created by the server or null.
147      */
148     public Response sendRequest(Request r) {
149         try {
150             r.setName(user);
151             out.writeObject(r);
152             return (Response) in.readObject();
153         } catch (Exception e) {
154             JOptionPane.showMessageDialog(null, e.getMessage());
155         }
156         return null;
157 
158     }
159 
160     /**
161      * Requests the list of films from the server, removes duplicated,
         then returns them as Strings.
162      * @return Film names as Strings
163      */
164     public String[] getFilmNames() {
165         Request r = new Request(Request.FILMS);
166         Response response = sendRequest(r);
167         Object[] filmObjs =
                     removeDuplicates(response.getResponseObjects());
168         if(filmObjs == null){
```

```
169             String[] n = {""};
170             return n;
171         }
172         String[] films = new String[filmObjs.length];
173         for (int x = 0; x < filmObjs.length; x++) {
174             films[x] = (String) filmObjs[x];
175         }
176         return films;
177     }
178 
179     /**
180      * Requests the list of dates of the films with the name passed in
           from the server, removes duplicated, then returns them as
           Strings.
181      * @param film the film name
182      * @return Strings of dates
183      */
184     String[] getFilmDates(String film) {
185         Request req = new Request(Request.FILM_DATES);
186         req.setFilm(film);
187 
188         Response response = sendRequest(req);
189         Object[] obj = removeDuplicates(response.getResponseObjects());
190         if(obj == null){
191             String[] n = {""};
192             return n;
193         }
194         String[] r = new String[obj.length];
195         for (int x = 0; x < obj.length; x++) {
196             r[x] = (String) obj[x];
197         }
198         return r;
199     }
200 
201     /**
202      * Requests the list of dates of the films with the name passed in
           from the server, removes duplicated, then returns them as
           Strings.
203      * @param film the film name
204      * @param date the film date
205      * @return String of times
206      */
207     String[] getFilmDateTimes(String film, String date) {
208         Request req = new Request(Request.FILM_DATE_TIMES);
209         req.setFilm(film);
210         req.setDate(date);
211 
212         Response response = sendRequest(req);
213         Object[] obj = removeDuplicates(response.getResponseObjects());
214         if (obj != null) {
215             String[] r = new String[obj.length];
216             for (int x = 0; x < obj.length; x++) {
217                 r[x] = (String) obj[x];
218             }
219             return r;
220         } else {
221             String[] n = {""};
222             return n;
223         }
224 
225     }
```

```
226
227     /**
228      * Gets, from the server, an array of available seats. If the film
           is fully booked it will return 0 to capacity.
229      * @param film the film name
230      * @param date the film date
231      * @param time the film time
232      * @return the seats as Strings
233      */
234     String[] getFilmDateTimeSeats(String film, String date, String
                time) {
235         Request req = new Request(Request.FILM_DATE_TIME_SEATS);
236         req.setFilm(film);
237         req.setDate(date);
238         req.setTime(time);
239 
240         Response response = sendRequest(req);
241         Object[] obj = removeDuplicates(response.getResponseObjects());
242         if (obj == null) {
243             String[] n = {""};
244             return n;
245         }
246         String[] r = new String[obj.length];
247         for (int x = 0; x < obj.length; x++) {
248             r[x] = (String) obj[x];
249         }
250         return r;
251     }
252 
253     /**
254      * Removes duplicated of any Object array.
255      * Used for removing duplications in the dropdown boxes.
256      * @param objs any object array
257      * @return a new Object array with no duplications
258      */
259     private Object[] removeDuplicates(final Object[] objs) {
260         if (objs == null || objs.length == 0) {
261             return null;
262         }
263         int count = 1;
264         boolean[] unique = new boolean[objs.length];
265         unique[0] = true;
266         for (int x = 1; x < objs.length; x++) {
267             unique[x] = true;
268             for (int y = 0; y < x; y++) {
269                 if ((objs[x].toString().equals(objs[y].toString()))) {
270                     unique[x] = false;
271                     break;
272                 }
273             }
274             if (unique[x]) {
275                 count++;
276             }
277         }
278 
279         Object[] r = new Object[count];
280         int i = 0;
281         for (int x = 0; x < objs.length; x++) {
282             if (unique[x]) {
283                 r[i++] = objs[x];
284             }
```

```
285             }
286             return r;
287         }
288 }
289
```

```
  1 /*
  2  * To change this template, choose Tools | Templates
  3  * and open the template in the editor.
  4  */
  5 package Client;
  6
  7 import java.awt.BorderLayout;
  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.IOException;
 14 import javax.swing.JButton;
 15 import javax.swing.JFrame;
 16 import javax.swing.JLabel;
 17 import javax.swing.JOptionPane;
 18 import javax.swing.JPanel;
 19 import javax.swing.JTextField;
 20 import javax.swing.WindowConstants;
 21
 22 /**
 23  * A Login GUI that connects to the server.
 24  * @author Robert and Nathan
 25  */
 26 public class Login extends JFrame implements ActionListener,
          KeyListener {
 27
 28     private static final long serialVersionUID = 1L;
 29     private JButton submit;
 30     private JPanel panel;
 31     private JLabel usernameLabel;
 32     private JTextField usernameText;
 33     private Comms server;
 34
 35     /**
 36      * Creates elements and adds them to this instance of Login, sets
          up ActionListeners to this instance.
 37      */
 38     public Login() {
 39
 40         usernameLabel = new JLabel();
 41         usernameLabel.setText("Username:");
 42         usernameText = new JTextField(15);
 43         usernameText.addKeyListener(this);
 44
 45         submit = new JButton("SUBMIT");
 46
 47         panel = new JPanel(new GridLayout(3, 1));
 48         panel.add(usernameLabel);
 49         panel.add(usernameText);
 50         panel.add(submit);
 51
 52         add(panel, BorderLayout.CENTER);
 53         submit.addActionListener(this);
```

```
54          setTitle("LOGIN FORM");
55
56          setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
57          setLocationRelativeTo(null);
58          setSize(300, 100);
59          setVisible(true);
60      }
61
62      /**
63       * Makes a new connection to the server (exits if it can't connect)
           and logs the name with the server's session.
64       * Constructs a new Menu page and shows it, hides and disposes of
           this page.
65       * @param ae
66       */
67      @Override
68      public void actionPerformed(ActionEvent ae) {
69          try {
70              server = new Comms();
71          } catch (IOException e) {
72              JOptionPane.showMessageDialog(null, e.getMessage());
73              System.exit(0);
74          }
75          String name = usernameText.getText();
76          server.logon(name);
77
78          Menu page = new Menu(server);
79          this.setVisible(false);
80          page.setVisible(true);
81          this.dispose();
82
83      }
84
85      /**
86       * Listens for an enter key press, runs actionPerformed when it's
            found.
87       * @param ke
88       */
89      @Override
90      public void keyReleased(KeyEvent ke) {
91          if (ke.getKeyChar() == '\n') {
92              actionPerformed(null);
93          }
94      }
95
96      /**
97       * Not implemented.
98       * @param ke
99       */
100     @Override
101     public void keyTyped(KeyEvent ke) {
102     }
103
104     /**
105      * Not implemented.
106      * @param ke
107      */
108     @Override
109     public void keyPressed(KeyEvent ke) {
110     }
111 }
```

112

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package Client;
 6
 7 import java.awt.event.ActionEvent;
 8 import java.awt.event.ActionListener;
 9 import java.awt.event.ItemEvent;
10 import java.awt.event.ItemListener;
11 import java.awt.event.WindowEvent;
12 import java.awt.event.WindowListener;
13 import javax.swing.*;
14 import javax.swing.event.ChangeEvent;
15 import javax.swing.event.ChangeListener;
16 import shared.Request;
17 import shared.Response;
18
19 /**
20  * The main menu GUI
21  * @author Robert and Nathan
22  */
23 public class Menu extends JFrame implements WindowListener,
         ActionListener, ChangeListener, ItemListener {
24
25     private static final long serialVersionUID = 1L;
26     private final Comms server;
27     private Urgent messages;
28     private JComboBox CBFilmDropdown;
29     private JComboBox CBDateDropdown;
30     private JComboBox CBTimeDropdown;
31     private JComboBox CBSeatsDropdown;
32     private JTextArea DealsText;
33     private JComboBox ABBookingDropdown;
34     private JSpinner ABSeatsSpinner;
35     private JComboBox DBBookingDropdown;
36     private final JTabbedPane tabbedGUI;
37
38     /**
39      * Constructs a menu. Adds elements to the tabbed view, adds
        listeners.
40      * @param s the instance that deals with this session on the server
41      */
42     public Menu(Comms s) {
43
44         super("Client GUI");
45         this.server = s;
46
47         messages = new Urgent();
48         messages.start();
49
50
51         tabbedGUI = new JTabbedPane();
52         tabbedGUI.addChangeListener(this);
53
54
55         // Pane 1
56
57         JPanel panel1 = new JPanel();
```

```
58
59          // label for film
60
61          JLabel labelFilm = new JLabel("Film", SwingConstants.LEFT);
62          panel1.add(labelFilm);
63
64          // dropdown box for films
65
66
67          String CBfilm[] = server.getFilmNames();
68          CBFilmDropdown = new JComboBox(CBfilm);
69          CBFilmDropdown.addItemListener(this);
70          panel1.add(CBFilmDropdown);
71
72          // label for date
73
74          JLabel labelDates = new JLabel("Dates", SwingConstants.LEFT);
75          panel1.add(labelDates);
76
77          // Dropdown box for date
78
79
80          String CBdate[] = server.getFilmDates((String)
                   CBFilmDropdown.getModel().getSelectedItem());
81          CBDateDropdown = new JComboBox(CBdate);
82          CBDateDropdown.addItemListener(this);
83          panel1.add(CBDateDropdown);
84
85          // label for time
86
87          JLabel labelTime = new JLabel("Time", SwingConstants.LEFT);
88          panel1.add(labelTime);
89
90          // Dropdown box for time
91
92          String time[] = server.getFilmDateTimes((String)
                   CBFilmDropdown.getModel().getSelectedItem(),
93                   (String) CBDateDropdown.getModel().getSelectedItem());
94          CBTimeDropdown = new JComboBox(time);
95          CBTimeDropdown.addItemListener(this);
96          panel1.add(CBTimeDropdown);
97
98          // label for Seats
99
100         JLabel labelSeats = new JLabel("No of Seats",
                                  SwingConstants.LEFT);
101         panel1.add(labelSeats);
102
103         // Dropdown box for No of Seats
104
105
106         String seats[] = server.getFilmDateTimeSeats((String)
                             CBFilmDropdown.getModel().getSelectedItem(),
107                  (String) CBDateDropdown.getModel().getSelectedItem(),
108                  (String) CBTimeDropdown.getModel().getSelectedItem());
109         CBSeatsDropdown = new JComboBox(seats);
110         panel1.add(CBSeatsDropdown);
111
112         // button for submit
113
114         JButton SUBMIT = new JButton("SUBMIT");
```

```
115         SUBMIT.addActionListener(this);
116         SUBMIT.setActionCommand("create");
117         panel1.add(SUBMIT);
118
119         // adding tab to tabbed gui
120
121         tabbedGUI.addTab("Create Booking", null, panel1, "First
                             Panel");
122
123         // pane 2
124
125         JPanel panel2 = new JPanel();
126
127         // adding Booking label
128
129         JLabel labelBooking = new JLabel("Booking",
                                  SwingConstants.LEFT);
130         panel2.add(labelBooking);
131
132         // adding booking dropdown box
133
134         String booking[] = server.getAllReservationsAsStrings();
135         ABBookingDropdown = new JComboBox(booking);
136         panel2.add(ABBookingDropdown);
137
138         // adding Booking label
139
140         JLabel labelAmendSeats = new JLabel("New No of Seats",
                                        SwingConstants.LEFT);
141         panel2.add(labelAmendSeats);
142
143         // No of seats spinner
144
145         ABSeatsSpinner = new JSpinner();
146
147         String noOfSeats[] = {"0"};
148         String ABseats = "0";
149         ABSeatsSpinner.setModel(new SpinnerListModel(noOfSeats));
150         ABSeatsSpinner.getModel().setValue(noOfSeats[0]);
151         panel2.add(ABSeatsSpinner);
152
153         String reservation = (String)
                   ABBookingDropdown.getModel().getSelectedItem();
154         if (!reservation.equals("")) {
155             String data[] = reservation.split(",");
156             String ABfilm = data[0].trim();
157             String ABdate = data[1].trim();
158             String ABtime = data[2].trim();
159             ABseats = data[3].trim();
160
161             ABSeatsSpinner.setModel(new
           SpinnerListModel(server.getFilmDateTimeSeats(ABfilm, ABdate,
           ABtime)));
162             //ABSeatsSpinner.getModel().setValue(ABseats);
163         }
164
165
166         // amend booking submit button
167
168         JButton SUBMIT2 = new JButton("SUBMIT");
169         SUBMIT2.addActionListener(this);
```

```
170          SUBMIT2.setActionCommand("amend");
171          panel2.add(SUBMIT2);
172
173          tabbedGUI.addTab("Amend Booking", null, panel2, "Second
                               Panel");
174
175          // pane 3
176
177          JPanel panel3 = new JPanel();
178
179          // label for Delete Booking
180
181          JLabel labelDeleteBooking = new JLabel("Booking",
                     SwingConstants.CENTER);
182          panel3.add(labelDeleteBooking);
183
184          // adding delete booking dropdown box
185
186
187          String Dbooking[] = server.getAllReservationsAsStrings();
188          DBBookingDropdown = new JComboBox(Dbooking);
189          panel3.add(DBBookingDropdown);
190
191          // adding submit button
192
193
194          JButton SUBMIT3 = new JButton("SUBMIT");
195          SUBMIT3.addActionListener(this);
196          SUBMIT3.setActionCommand("delete");
197          panel3.add(SUBMIT3);
198
199          tabbedGUI.addTab("Delete Booking", null, panel3, "Third
                               Panel");
200
201          // pane 4
202
203          JPanel panel4 = new JPanel();
204
205          // adding deals label
206
207          JLabel labelDeals = new JLabel("Deals", SwingConstants.CENTER);
208          panel4.add(labelDeals);
209
210          // adding text area
211
212          DealsText = new JTextArea(6, 30);
213          DealsText.setEditable(false);
214          panel4.add(DealsText);
215
216          Request request = new Request(Request.REFRESH_OFFERS);
217          Response response = server.sendRequest(request);
218          if (response.getSuccess()) {
219              DealsText.setText(response.getResponse());
220          } else {
221              DealsText.setText("");
222          }
223
224          // adding refresh button
225          JButton Refresh = new JButton("Refresh");
226          Refresh.addActionListener(this);
227          Refresh.setActionCommand("refresh");
```

```
228         panel4.add(Refresh);
229
230         tabbedGUI.addTab("Deals", null, panel4, "Forth Panel");
231
232         this.add(tabbedGUI);
233
234
            setDefaultCloseOperation(javax.swing.WindowConstants.DO_NOTHING
            _ON_CLOSE);
235         addWindowListener(this);
236         setLocationRelativeTo(null);
237         setSize(450, 220);
238     }
239
240     /**
241      * Not implemented.
242      * @param we
243      */
244     @Override
245     public void windowOpened(WindowEvent we) {
246     }
247
248     /**
249      * Logs off the server, disposes of this window and creates a new
         instance of Login.
250      * @param we
251      */
252     @Override
253     public void windowClosing(WindowEvent we) {
254         server.logoff();
255         messages.stop();
256         this.setVisible(false);
257         try {
258             Login l = new Login();
259         } catch (Exception e) {
260             JOptionPane.showMessageDialog(null, e.getMessage());
261             System.exit(0);
262         }
263         this.dispose();
264
265     }
266
267     /**
268      * Not implemented.
269      * @param we
270      */
271     @Override
272     public void windowClosed(WindowEvent we) {
273     }
274
275     /**
276      * Not implemented.
277      * @param we
278      */
279     @Override
280     public void windowIconified(WindowEvent we) {
281     }
282
283     /**
284      * Not implemented.
285      * @param we
```

```
286         */
287        @Override
288        public void windowDeiconified(WindowEvent we) {
289        }
290
291        /**
292         * Not implemented.
293         * @param we
294         */
295        @Override
296        public void windowActivated(WindowEvent we) {
297        }
298
299        /**
300         * Not implemented.
301         * @param we
302         */
303        @Override
304        public void windowDeactivated(WindowEvent we) {
305        }
306
307        /**
308         * Performs the action based on which action command is passed from
              a button on each tab.
309         * @param e
310         */
311        @Override
312        public void actionPerformed(ActionEvent e) {
313            String command = e.getActionCommand();
314            if (command.equals("create")) {
315                String film = (String)
                          CBFilmDropdown.getModel().getSelectedItem();
316                String date = (String)
                          CBDateDropdown.getModel().getSelectedItem();
317                String time = (String)
                          CBTimeDropdown.getModel().getSelectedItem();
318
319                //cast dropdown object to String then parse String to int
320                int seats = 0;
321                try {
322                    seats = Integer.parseInt((String)
                              CBSeatsDropdown.getModel().getSelectedItem());
323                } catch (NumberFormatException ne) {
324                    JOptionPane.showMessageDialog(null, "Looks like there
                          are no eats available for this showing.");
325                    return;
326                }
327
328                Request request = new Request(Request.MAKE);
329
330                request.setFilm(film);
331                request.setDate(date);
332                request.setTime(time);
333                request.setSeats(seats);
334
335                Response response = server.sendRequest(request);
336                if (response.getSuccess()) {
337                    JOptionPane.showMessageDialog(null, "Success!");
338                } else {
339                    JOptionPane.showMessageDialog(null,
                                  response.getReason());
```

```
340                 }
341                 return;
342
343
344             } else if (command.equals("amend")) {
345
346                 Request request = new Request(Request.AMEND);
347
348                 String booking = (String)
                          ABBookingDropdown.getModel().getSelectedItem();
349                 if (booking.equals("")) {
350                     JOptionPane.showMessageDialog(null, "You have no
                                                bookings, make one first.");
351                     return;
352                 }
353                 String data[] = booking.split(",");
354                 String film = data[0].trim();
355                 String date = data[1].trim();
356                 String time = data[2].trim();
357                 int seats = Integer.parseInt(data[3].trim());
358                 int newSeats = Integer.parseInt((String)
                              ABSeatsSpinner.getModel().getValue());
359
360                 request.setFilm(film);
361                 request.setDate(date);
362                 request.setTime(time);
363                 request.setSeats(seats);
364                 request.setNewSeats(newSeats);
365
366                 Response response = server.sendRequest(request);
367
368                 String[] r = server.getAllReservationsAsStrings(true);
369                 ABBookingDropdown.setModel(new DefaultComboBoxModel(r));
370                 ABBookingDropdown.validate();
371                 ABBookingDropdown.repaint();
372
373                 if (response.getSuccess()) {
374                     JOptionPane.showMessageDialog(null, "Success!");
375                 } else {
376                     JOptionPane.showMessageDialog(null,
                                                    response.getReason());
377                 }
378                 return;
379
380             } else if (command.equals("delete")) {
381                 Request request = new Request(Request.DELETE);
382
383                 String booking = (String)
                          DBBookingDropdown.getModel().getSelectedItem();
384                 if (booking.equals("")) {
385                     JOptionPane.showMessageDialog(null, "You have no
                                                bookings, make one first.");
386                     return;
387                 }
388                 String data[] = booking.split(",");
389                 String film = data[0].trim();
390                 String date = data[1].trim();
391                 String time = data[2].trim();
392                 int seats = Integer.parseInt(data[3].trim());
393
394                 request.setFilm(film);
```

```
395                request.setDate(date);
396                request.setTime(time);
397                request.setSeats(seats);
398
399                Response response = server.sendRequest(request);
400
401                DBBookingDropdown.setModel(new
                     DefaultComboBoxModel(server.getAllReservationsAsStrings(
                     true)));
402                DBBookingDropdown.validate();
403                DBBookingDropdown.repaint();
404
405                if (response.getSuccess()) {
406                    JOptionPane.showMessageDialog(null, "Success!");
407                } else {
408                    JOptionPane.showMessageDialog(null,
                                         response.getReason());
409                }
410                return;
411
412            } else if (command.equals("refresh")) {
413                Request request = new Request(Request.REFRESH_OFFERS);
414
415                Response response = server.sendRequest(request);
416                if (response.getSuccess()) {
417                    DealsText.setText(response.getResponse());
418                    return;
419                } else {
420                    JOptionPane.showMessageDialog(null,
                                           response.getReason());
421                    return;
422                }
423
424            } else {
425                JOptionPane.showMessageDialog(null, "Something bad
                                                happened!");
426                return;
427            }
428        }
429
430        /**
431         * Refreshes data on the selected tab after changing tab.
432         * Gets the new data and updates the model of the dropdown boxes.
433         * @param evt
434         */
435        @Override
436        public void stateChanged(ChangeEvent evt) {
437            if (evt.getSource() instanceof JTabbedPane) {
438                JTabbedPane pane = (JTabbedPane) evt.getSource();
439                String name = pane.getTitleAt(pane.getSelectedIndex());
440
441                if (name.equals("Create Booking")) {
442                    CBFilmDropdown.setModel(new
                         DefaultComboBoxModel(server.getFilmNames()));
443                    itemStateChanged(new ItemEvent(CBFilmDropdown, 0, null,
                                                   0));
444                } else if (name.equals("Amend Booking")) {
445                    ABBookingDropdown.setModel(new
                        DefaultComboBoxModel(server.getAllReservationsAsStri
                        ngs(true)));
446                    String noOfSeats[] = {"0"};
```

```
447                     ABSeatsSpinner.setModel(new
                                SpinnerListModel(noOfSeats));
448                     ABSeatsSpinner.getModel().setValue(noOfSeats[0]);
449
450                     String reservation = (String)
                              ABBookingDropdown.getModel().getSelectedItem();
451                     if (!reservation.equals("")) {
452                         String data[] = reservation.split(",");
453                         String ABfilm = data[0].trim();
454                         String ABdate = data[1].trim();
455                         String ABtime = data[2].trim();
456                         String ABseats = data[3].trim();
457
458                         ABSeatsSpinner.setModel(new
                               SpinnerListModel(server.getFilmDateTimeSeats(AB
                               film, ABdate, ABtime)));
459                         ABSeatsSpinner.getModel().setValue(ABseats);
460                     }
461                 } else if (name.endsWith("Delete Booking")) {
462                     DBBookingDropdown.setModel(new
                        DefaultComboBoxModel(server.getAllReservationsAsStrings
                        (true)));
463                 }
464                 pane.repaint();
465             }
466         }
467
468         /**
469          * Refreshes the next dropdown box when one has been changed.
470          * Cascades the updating of the models of dropdown boxes.
471          * @param ie
472          */
473         @Override
474         public void itemStateChanged(ItemEvent ie) {
475             JComboBox box = (JComboBox) ie.getSource();
476             if (box == CBFilmDropdown) {
477                 String film = (String) box.getModel().getSelectedItem();
478                 String[] dates = server.getFilmDates(film);
479                 CBDateDropdown.setModel(new DefaultComboBoxModel(dates));
480                 itemStateChanged(new ItemEvent(CBDateDropdown, 0, null,
                                                                    0));
481             } else if (box == CBDateDropdown) {
482                 String film = (String)
                          CBFilmDropdown.getModel().getSelectedItem();
483                 String date = (String) box.getModel().getSelectedItem();
484                 String[] times = server.getFilmDateTimes(film, date);
485                 CBTimeDropdown.setModel(new DefaultComboBoxModel(times));
486                 itemStateChanged(new ItemEvent(CBTimeDropdown, 0, null,
                                                                    0));
487             } else if (box == CBTimeDropdown) {
488                 String film = (String)
                          CBFilmDropdown.getModel().getSelectedItem();
489                 String date = (String)
                          CBDateDropdown.getModel().getSelectedItem();
490                 String time = (String) box.getModel().getSelectedItem();
491                 String[] seats = server.getFilmDateTimeSeats(film, date,
                            time);
492                 CBSeatsDropdown.setModel(new DefaultComboBoxModel(seats));
493
494             }
495         }
```

```
496 }
497
```

```
  1 /*
  2  * To change this template, choose Tools | Templates
  3  * and open the template in the editor.
  4  */
  5 package Client;
  6
  7 import java.awt.BorderLayout;
  8 import java.awt.Container;
  9 import java.awt.Dimension;
 10 import java.awt.event.ActionEvent;
 11 import java.awt.event.ActionListener;
 12 import java.io.BufferedReader;
 13 import java.io.IOException;
 14 import java.io.InputStreamReader;
 15 import java.net.Socket;
 16 import javax.swing.JButton;
 17 import javax.swing.JFrame;
 18 import javax.swing.JLabel;
 19 import javax.swing.JOptionPane;
 20 import javax.swing.JTextArea;
 21 import javax.swing.WindowConstants;
 22
 23 /**
 24  *The threaded window that listens for urgent messages from the server
     and updates the text box accordingly.
 25  * @author Robert
 26  */
 27 public class Urgent extends JFrame implements Runnable, ActionListener
          {
 28
 29     private static final long serialVersionUID = 1L;
 30     private BufferedReader MsgServer;
 31     private boolean quit;
 32     private JTextArea msgs;
 33
 34     /**
 35      * Constructs the window and sets up the server connections.
 36      */
 37     public Urgent() {
 38         try {
 39             Socket s = new Socket("localhost", 2001);
 40             s.setSoTimeout(5);
 41             MsgServer = new BufferedReader(new
                            InputStreamReader(s.getInputStream()));
 42
 43         } catch (IOException e) {
 44             JOptionPane.showMessageDialog(null, e.getMessage());
 45             System.exit(0);
 46         }
 47
 48         quit = false;
 49         JLabel lbl1 = new JLabel("Urgent Messages:");
 50         msgs = new JTextArea();
 51         msgs.setEditable(false);
 52         msgs.setText("");
 53         msgs.setPreferredSize(new Dimension(200, 200));
 54         JButton hideBtn = new JButton("Hide");
 55         Container panel = this.getContentPane();
```

```
56
57          panel.add(lbl1, BorderLayout.PAGE_START);
58          panel.add(msgs, BorderLayout.CENTER);
59          panel.add(hideBtn, BorderLayout.PAGE_END);
60
61          hideBtn.addActionListener(this);
62          setTitle("Urgent Messages!");
63
64          setDefaultCloseOperation(WindowConstants.HIDE_ON_CLOSE);
65          setSize(200, 250);
66          this.pack();
67      }
68
69      /**
70       * Listens for messages and updates the text box.
71       */
72      @Override
73      public void run() {
74          this.setVisible(true);
75          while (!quit) {
76              String m = msgs.getText();
77              String n = null;
78              try {
79                  n = MsgServer.readLine();
80              } catch (IOException e) {
81                  continue;
82              }
83              if (n != null) {
84                  msgs.setText(n + "\n" + m);
85                  this.setVisible(true);
86              }
87          }
88          this.dispose();
89      }
90
91      /**
92       * Starts a new thread from this instance of runnable.
93       */
94      public void start() {
95          Thread t = new Thread(this);
96          t.start();
97      }
98
99      /**
100      * Sets the quit flag to tell the thread to finish.
101      */
102     public void stop() {
103         this.quit = true;
104     }
105
106     /**
107      * Listens for the 'Hide' button to be presses and sets the window
           to invisible when pressed.
108      * @param ae
109      */
110     @Override
111     public void actionPerformed(ActionEvent ae) {
112         this.setVisible(false);
113     }
114 }
115
```

3.3 Commented Default Class

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package Default;
6
7  /**
8   * Default class.
9   * @author Robert
10  */
11 public class Main {
12     /**
13      * The Default entry point of the project. If no class path is
          passed then this will run illustrating to the user the correct
          usage.
14      * @param args no arguments are needed
15      */
16     public static void main(String[] args){
17         System.err.println("Error!!");
18         System.err.println("To run the server use:");
19         System.err.println("java -cp Task2.jar Server.Server\n");
20         System.err.println("To run the client use:");
21         System.err.println("java -cp Task2.jar Client.Client <host of
                                the server>\n");
22         System.err.println("To run the admin application use:");
23         System.err.println("java -cp Task2.jar Admin.Editor <host of the
                                server>");
24         System.exit(0);
25     }
26
27 }
28
```

3.4 Commented Server Code

```
1  /*
2   * To change this template, choose Tools | Templates
3   * and open the template in the editor.
4   */
5  package Server;
6
7  import java.io.IOException;
8  import java.io.InputStreamReader;
9
10 /**
11  *Thread that checks for a quit or force quit command from the keyboard
      on STDIN.
12  * @author Robert and Nathan
13  */
14 public class CmdThread extends Thread {
15     private Server server;
16     private boolean quit = false;
17
18     /**
19      * Constructs and instance.
20      * @param s instance of the server
```

```
21       */
22      public CmdThread(Server s){
23          this.setName("CMDThread");
24          server = s;
25      }
26
27      /**
28       * Reads characters from STDIN and sets quit/force quit flags
           accordingly.
29       */
30      @Override
31      synchronized public void run() {
32          InputStreamReader in = new InputStreamReader(System.in);
33          int c;
34          while(!quit){
35              try{
36              c = in.read();}
37              catch (IOException e){ continue;}
38              if(c == (int)'q'){
39                  quit = true;
40                  server.setQuit("q");
41              }
42              if(c == (int)'f'){
43                  quit = true;
44                  server.setQuit("f");
45              }
46          }
47      }
48
49 }
50
```

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package Server;
 6
 7 import java.util.Collections;
 8 import java.util.Iterator;
 9 import java.util.LinkedList;
10 import java.util.List;
11 import shared.Booking;
12 import shared.Film;
13
14 /**
15  *The class to hold all server data and deal with requests of the data.
16  * @author Robert and Nathan
17  */
18 public class Data {
19
20     private Film[] films;
21     private List<Booking> reservations;
22     private final UrgentMsgThread urgent;
23     private String offers;
24
25     /**
26      * Constructs the instance
27      * @param u instance of the Urgent Message Thread
28      */
29     public Data(UrgentMsgThread u){
```

```
30      this.urgent = u;
31      }
32
33      /**
34       * Makes the list synchronized and keeps it in the instance.
35       * @param ll a LinkedList of bookings from the data file
36       */
37      public void addReservationsLL(LinkedList<Booking> ll){
38          this.reservations = Collections.synchronizedList(ll);
39      }
40
41      /**
42       * Stores the array in the instance.
43       * @param films array of Films from the data file
44       */
45      public void addFilmsArray(Film[] films){
46          this.films = films;
47      }
48
49      /**
50       * find all reservations that were made by customerName
51       * @param customerName the customer's name
52       * @return an array of Bookings that belong to the customer
53       */
54      public Booking[] getReservations(String customerName) {
55          synchronized (reservations) {
56              LinkedList<Booking> r = new LinkedList<Booking>();
57              Iterator<Booking> it = reservations.iterator();
58              while (it.hasNext()) {
59                  Booking res = it.next();
60                  if (res.getName().equals(customerName)) {
61                      r.add(res);
62                  }
63              }
64              if (r.size() > 0) {
65
66                  Booking[] a = new Booking[r.size()];
67                  Object[] objArray = r.toArray();
68                  int i = 0;
69                  for (Object o : objArray) {
70                      a[i++] = (Booking) o;
71                  }
72
73                  return a;
74              }
75              return null;
76
77          }
78      }
79
80      /**
81       * Searches for a film in the array, returns null if not found
82       * @param name the Films name
83       * @param date the date of the Film "dd/mm/yyyy"
84       * @param time the time of the Film "hh:mm"
85       * @return Film instance
86       */
87      public synchronized Film findFilm(String name, String date, String
                time) {
88          if(films == null){
89              return null;
```

```
90          }
91              for(int x=0;x<films.length;x++){
92                  if (films[x].getName().equals(name) &&
                            films[x].getDate().equals(date)
93                          && films[x].getTime().equals(time)) {
94                      return films[x];
95                  }
96              }
97          return null;
98      }
99
100     /**
101      * makes a reservation and stores it in the linked list
102      * @param customer the customer's name
103      * @param film the film name
104      * @param date the film date
105      * @param time the time of the film
106      * @param no the number of seats
107      * @return true if made, false if couldn't make reservation
108      */
109     public boolean makeReservation(String customer, String film,
110             String date, String time, int no) {
111         Film f = findFilm(film, date, time);
112         if (f == null) {
113             return false;
114         }
115         synchronized (reservations) {
116             if (f.space() >= no) {
117                 f.book(no);
118                 Booking b = new Booking(customer, f, no);
119                 reservations.add(b);
120
121                 if (f.space() == 0) {
122                     urgent.send(film + " is now fully booked at "
123                             + date + " " + time);
124                 }
125
126                 return true;
127             }
128         }
129         return false;
130     }
131
132     /**
133      * Finds the customer's reservation and removes it from the linked
          list, also frees space from the film
134      * @param customer the customer's name
135      * @param film the film name
136      * @param date the film date
137      * @param time the film time
138      * @param no the number of seats
139      */
140     public void cancelReservation(String customer, String film,
141             String date, String time, int no) {
142         Film f = findFilm(film, date, time);
143         Booking[] b = getReservations(customer);
144         synchronized (reservations) {
145             for (int x = 0; x < b.length; x++) {
146                 if ((b[x].getFilm() == f) && (b[x].getSeats() == no)) {
147                     reservations.remove(b[x]);
148                 }
```

```
149
150                 }
151             }
152             int s = f.space();
153             f.free(no);
154             if (s == 0) {
155                 urgent.send(film + " now has " + f.space()
156                         + " seats available at " + date + " " + time);
157             }
158 
159         }
160 
161         /**
162          * Removes old booking and creates a new booking
163          * @param customer the customer name
164          * @param film the film name
165          * @param date the film date
166          * @param time the film time
167          * @param oldSeats the number of seats the old reservation had
168          * @param newSeats the number of seats for the new reservation to
           have
169          * @return true if able to make new booking and false if could not.
170          */
171         public boolean changeReservation(String customer, String film,
172                 String date, String time, int oldSeats, int newSeats) {
173             this.cancelReservation(customer, film, date, time, oldSeats);
174             return this.makeReservation(customer, film, date, time,
                                              newSeats);
175         }
176 
177         /**
178          * Checks if a film is fully booked.
179          * @param film the film name
180          * @param date the film date
181          * @param time the film time
182          * @return true if film is fully booked, false if not
183          */
184         public boolean isFull(String film, String date, String time) {
185             Film f = findFilm(film, date, time);
186             if (f.space() == 0) {
187                 return true;
188             } else {
189                 return false;
190             }
191 
192         }
193 
194         /**
195          * Gets the amount of free space that the film has.
196          * @param film the film name
197          * @param date the film date
198          * @param time the film time
199          * @return the number of seats available for booking
200          */
201         public int getSpace(String film, String date, String time) {
202             Film f = findFilm(film, date, time);
203             return f.space();
204         }
205 
206         /**
207          * Gets a new iterator of the reservations LinkedList
```

```
208         * @return a new iterator
209         */
210        public Iterator<Booking> getBookingIt() {
211            synchronized (reservations) {
212                return reservations.iterator();
213            }
214        }
215
216        /**
217         * Gets all of the films to string and creates a CSV format string
             ready to be written to file
218         * @return the films to a csv string
219         */
220        public synchronized String allFilmsToString() {
221            String r = "";
222            if (films != null) {
223                for (int x = 0; x < films.length; x++) {
224                    r += films[x].toString() + Server.endLine;
225                }
226                return r;
227            } else {
228                return null;
229            }
230
231        }
232
233        /**
234         * Gets a String array of the names of all the films
235         * @return Film names
236         */
237        public String[] getFilmNames(){
238            if(films == null){
239                return null;
240            }
241            String[] r = new String[films.length];
242            for(int x=0; x<r.length;x++){
243                r[x] = films[x].getName();
244            }
245            return r;
246        }
247
248        /**
249         * Gets an array of films whos name match the parameter
250         * @param film the film name
251         * @return an Object array of Films
252         */
253        public Object[] findFilms(String film) {
254            LinkedList<Film> r = new LinkedList<Film>();
255            if(films == null){
256                return null;
257            }
258            for(int x=0;x<films.length;x++){
259                if(films[x].getName().equals(film)){
260                    r.add(films[x]);
261                }
262            }
263            if(r.isEmpty()){
264                return null;
265            }
266            return r.toArray();
267        }
```

```
268
269     /**
270      * Gets an array of films whos name and date match the parameters
271      * @param film the films name
272      * @param date the films time
273      * @return an Object array of Films
274      */
275     public Object[] findFilms(String film, String date){
276          LinkedList<Film> r = new LinkedList<Film>();
277          if(films == null){
278              return null;
279          }
280         for(int x=0;x<films.length;x++){
281             if(films[x].getName().equals(film) &&
                             films[x].getDate().equals(date)){
282                 r.add(films[x]);
283             }
284         }
285         if(r.isEmpty()){
286             return null;
287         }
288         return r.toArray();
289
290     }
291
292     /**
293      * Sets the offers for use by the clients
294      * @param off the offer string
295      */
296     public void setOffers(String off){
297         this.offers = off;
298     }
299
300     /**
301      * Get the offers from the data structure
302      * @return the offers string
303      */
304     public String getOffers(){
305         return this.offers;
306     }
307 }
308
```

```
  1 /*
  2  * To change this template, choose Tools | Templates
  3  * and open the template in the editor.
  4  */
  5 package Server;
  6
  7 import java.io.File;
  8 import java.io.FileWriter;
  9 import java.io.IOException;
 10 import java.text.SimpleDateFormat;
 11 import java.util.Calendar;
 12
 13 /**
 14  *A log file writer.
 15  * @author Robert
 16  */
 17 public class Log {
 18
```

```
19      private Calendar now = null;
20      private File logfile = null;
21      private FileWriter log = null;
22
23      /**
24       * Constructs a Log instance, sets up the file named by the date or
          server.log if appending
25       * @param append boolean if true, then data will be written to the
          end of the file rather than the beginning
26       */
27      public Log(boolean append) {
28          try {
29              if (append) {
30                  logfile = new File("logs" + File.separator +
                                            "server.log");
31              } else {
32                  logfile = new File("logs" + File.separator + getTime()
                                    + ".log");
33              }
34              if (!logfile.getParentFile().exists()) {
35                  logfile.getParentFile().mkdirs();
36              }
37              log = new FileWriter(logfile, append);
38          } catch (IOException ex) {
39              System.err.println(ex.getMessage());
40          }
41      }
42
43      /**
44       * Constructs a Log instance, sets up the file named by the date.
45       */
46      public Log() {
47          try {
48              logfile = new File("logs" + File.separator + getTime() +
                                ".log");
49              if (!logfile.getParentFile().exists()) {
50                  logfile.getParentFile().mkdirs();
51              }
52              log = new FileWriter(logfile);
53          } catch (IOException ex) {
54              System.err.println(ex.getMessage());
55          }
56      }
57
58      /**
59       * Gets the date and time formatted as "yyyy-MM-dd HH:mm:ss" to be
          used within the instance.
60       * @return the formatted date/time
61       */
62      private String getTime() {
63          now = Calendar.getInstance();
64          SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH-mm-
                                            ss");
65          String time = sdf.format(now.getTime());
66          return time;
67      }
68
69      /**
70       * Writes a message to the file (and optionally to stdout) in the
          format "<i>TIME</i>:: <i>MESSAGE</i>"
71       * @param msg the message to be written
```

```
72        * @param stdout use true if the Message should be mirrored to
            stdout, false for file only.
73        * @return true if file was successfully written to, false
            otherwise
74        */
75       public boolean writeMessage(String msg, boolean stdout) {
76           try {
77               String message = getTime() + ":: " + msg + Server.endLine;
78               if (stdout) {
79                   System.out.print(message);
80               }
81               log.write(message);
82               log.flush();
83               return true;
84           } catch (IOException e) {
85               System.err.println(getTime() + ":: Could not write message:
                                     " + msg);
86               return false;
87           }
88       }
89
90       /**
91        * Writes an error to the file (and optionally to stderr) in the
            format "<i>TIME</i>:: ERROR: <i>MESSAGE</i>"
92        * @param err the error message
93        * @param stdout use true if the Message should be mirrored to
            stderr, false for file only.
94        * @return true if file was successfully written to, false
            otherwise
95        */
96       public boolean writeError(String err, boolean stdout) {
97           System.err.println(getTime() + ":: ERROR:" + err);
98           return writeMessage("ERROR: " + err, false);
99       }
100
101      /**
102       * Writes an event to the file (and optionally to stdout) in the
            format "<i>TIME</i>:: EVENT: <i>MESSAGE</i>"
103       * @param e the event message
104       * @param stdout use true if the Message should be mirrored to
            stdout, false for file only.
105       * @return true if file was successfully written to, false
            otherwise
106       */
107      public boolean writeEvent(String e, boolean stdout) {
108          return writeMessage("EVENT: " + e, stdout);
109      }
110 }
111
```

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package Server;
 6
 7 import java.io.File;
 8 import java.io.FileInputStream;
 9 import java.io.FileOutputStream;
10 import java.io.IOException;
11 import java.net.ServerSocket;
```

```
12 import java.net.Socket;
13 import java.util.Arrays;
14 import java.util.Collections;
15 import java.util.Iterator;
16 import java.util.LinkedList;
17 import java.util.List;
18 import shared.Booking;
19 import shared.Film;
20
21 /**
22  *The server's entry point and core server functionality
23  * @author Robert
24  */
25 public class Server {
26
27     private static final String filmFile = "data" + File.separator +
                                         "films.txt";
28     private static final String reservationFile = "data" +
                                 File.separator + "reservations.txt";
29     private static final String userFile = "data" + File.separator +
                                         "users.txt";
30     private static final String offersFile = "data" + File.separator +
                                           "special_offers.txt";
31     /**
32      * Global access to the end line character (not OS specific) for
         use in file read/write
33      */
34     public static final String endLine = "\r\n";
35
36     /**
37      * A non-blocking timeout constant for use in the server
38      */
39     public static final int TIMEOUT = 10;
40     /**
41      * A blocking timeout for the server.
42      */
43     public static final int TIMEOUT_BLOCK = 0;
44
45     private Data data;
46     private List<Session> _clients;
47     private boolean quit;
48     private boolean forcequit;
49     private List<String> users;
50     private String[] offers;
51
52     /**
53      * Entry point for the server
54      * @param args
55      */
56     public static void main(String[] args) {
57         Server server = new Server();
58         File films = new File(Server.filmFile);
59         File reservations = new File(Server.reservationFile);
60         File users = new File(Server.userFile);
61         File offers = new File(Server.offersFile);
62         server.readFile(films, reservations, users, offers);
63
64         ServerSocket s = null;
65         try {
66             s = new ServerSocket(2000);
67             s.setSoTimeout(Server.TIMEOUT);
```

```
68          } catch (IOException e) {
69              server.log.writeError(e.getMessage(), true);
70              System.exit(0);
71          }
72
73          server.log.writeEvent("Running", true);
74          while (!server.quitting()) {
75              Socket cl = null;
76              try {
77                  cl = s.accept();
78              } catch (IOException e) {
79                  continue;
80              }
81              try{
82               server.addClient(cl);
83              } catch (IOException e) {
84                  continue;
85              }
86
87          }
88          server.urgent.send("The server is shutting down!");
89          if (server.forceQuitting()) {
90              server.closeAllClients();
91          } else {
92              server.waitOnClients();
93          }
94
95          server.writeFile(films, reservations, users);
96
97          server.log.writeEvent("Shutdown", true);
98      }
99      /**
100      * The instance of the urgent message thread. Should only be used
         for .send(message)
101      */
102     public final UrgentMsgThread urgent;
103     private final CmdThread cmd;
104     /**
105      * The instance of the log file writer. Should only be used to
         write messages/errors/events
106      */
107     public final Log log;
108
109     /**
110      * Constructs a new server instance
111      */
112     public Server() {
113         log = new Log();
114         log.writeEvent("Started", false);
115         _clients = Collections.synchronizedList(new
                        LinkedList<Session>());
116         users = Collections.synchronizedList(new LinkedList<String>());
117         quit = false;
118         forcequit = false;
119
120         urgent = new UrgentMsgThread(this);
121         urgent.start();
122
123         cmd = new CmdThread(this);
124         cmd.start();
125
```

```
126            data = new Data(urgent);
127        }
128
129        /**
130         * Reads the parameter files into the internal data structures.
131         * Splits on end lines, then by commas to retrieve each 'cell' of
               information.
132         * @param f Films file
133         * @param r Reservations file
134         * @param u Users file
135         * @param s Special Offers file
136         */
137        public void readFile(File f, File r, File u, File s) {
138            LinkedList<Booking> bll = new LinkedList<Booking>();
139            LinkedList<String> ull = new LinkedList<String>();
140            Film[] fl = null;
141            String ofrs = null;
142            try {
143
144                if (u.exists()) {
145                    FileInputStream uf = new FileInputStream(u);
146                    byte[] us = new byte[uf.available()];
147                    uf.read(us);
148                    ull.addAll(Arrays.asList(new
                               String(us).split(Server.endLine)));
149                    uf.close();
150                } else {
151                    u.getParentFile().mkdirs();
152                    u.createNewFile();
153                }
154
155                if (f.exists()) {
156                    FileInputStream ff = new FileInputStream(f);
157                    byte[] fileBytes = new byte[ff.available()];
158                    ff.read(fileBytes);
159                    String films = new String(fileBytes);
160                    String[] film = films.split(Server.endLine);
161                    fl = new Film[film.length];
162                    if (!film[0].equals("")) {
163                        for (int x = 0; x < film.length; x++) {
164                            String[] items = film[x].split(",");
165                            fl[x] = new Film(items[0], items[1],
166                                    items[2], Integer.parseInt(items[3]),
167                                    Integer.parseInt(items[4]));
168                        }
169                    } else {
170                        fl = null;
171                    }
172                    ff.close();
173                } else {
174                    f.createNewFile();
175                }
176
177                data.addFilmsArray(fl);
178
179                if (r.exists()) {
                       FileInputStream fis = new FileInputStream(r);
181                    byte[] t = new byte[fis.available()];
182                    fis.read(t);
183                    String text = new String(t);
184                    String[] records = text.split(Server.endLine);
```

```
185                 if(!records[0].equals("")){
186                 for (int x = 0; x < records.length; x++) {
187                     String[] row = records[x].split(",");
188                     Booking b = new Booking(row[5],
                                                    data.findFilm(row[0],
189                             row[1], row[2]), Integer.parseInt(row[4]));
190                     bll.add(b);
191                 }}
192                 fis.close();
193             } else {
194                 r.createNewFile();
195             }
196
197             data.addReservationsLL(bll);
198
199             if(s.exists()){
200                 FileInputStream of = new FileInputStream(s);
201                 byte[] o = new byte[of.available()];
202                 of.read(o);
203                 ofrs = new String(o);
204             }
205         } catch (IOException ioe) {
206             log.writeError(ioe.getMessage(), true);
207         }
208         users = ull;
209         data.setOffers(ofrs);
210     }
211
212     /**
213      * Writes the internal data out the the parameter files in (rough)
           CSV format.
214      * @param ff films file
215      * @param rf reservations file
216      * @param uf users file
217      */
218     public void writeFile(File ff, File rf, File uf) {
219         try {
220             FileOutputStream fos = new FileOutputStream(ff);
221             String fs = data.allFilmsToString();
222             if(fs!=null){
223                 byte[] fb = fs.getBytes();
224                 fos.write(fb);
225             }
226             fos.close();
227
228             fos = new FileOutputStream(rf);
229             Iterator<Booking> bit = data.getBookingIt();
230             while(bit.hasNext()){
231                 Booking b = bit.next();
232                 String res = b.getFilm().getName() + ","
233                         + b.getFilm().getDate() + ","
234                         + b.getFilm().getTime()
235                         + "," + b.getFilm().getBooked()
236                         + "," + b.getSeats()
237                         + "," + b.getName() + Server.endLine;
238                 fos.write(res.getBytes());
239             }
240             fos.close();
241
242             fos = new FileOutputStream(uf);
243             synchronized (users) {
```

```
244                 Iterator<String> uit = users.iterator();
245                 while (uit.hasNext()) {
246                     String u = uit.next();
247                     String name = u + Server.endLine;
248                     fos.write(name.getBytes());
249                 }
250             }
251             fos.close();
252 
253         } catch (IOException ioe) {
254             log.writeError(ioe.getMessage(), true);
255         }
256 
257     }
258 
259     /**
260      * Constructs a new Session, adds it to the linked list and starts
        the session
261      * @param s A Socket to the client
262      * @throws IOException Couldn't add the client to the linked list
263      */
264     public void addClient(Socket s) throws IOException {
265         synchronized (_clients) {
266             Session c = new Session(s, this);
267             if (!_clients.add(c)) {
268                 throw new IOException("Could not add client to list.");
269             }
270             c.start();
271         }
272     }
273 
274     /**
275      * Force quits the session
276      * @param c the Session to remove
277      */
278     public void removeClient(Session c) {
279         synchronized (_clients) {
280             this._clients.remove(c);
281         }
282         c.forceQuit();
283     }
284 
285     /**
286      * Is the server shutting down?
287      * @return true if it is, false if it's not
288      */
289     public synchronized boolean  quitting() {
290         return quit;
291     }
292 
293     /**
294      * Set the server to quit (pass "q" in) or force quit (pass "f" in)
295      * @param q String indicating quit level
296      */
297     public synchronized void setQuit(String q) {
298         if (q.equals("q")) {
299             quit = true;
300         }
301         if (q.equals("f")) {
302             quit = true;
303             forcequit = true;
```

```
304             }
305         }
306
307         /**
308          * Is the server force quitting?
309          * @return true if the server is force quitting, false if not
310          */
311         public boolean forceQuitting() {
312             return forcequit;
313         }
314
315         /**
316          * Blocks the server from shutting down until all clients are
               disconnected.
317          */
318         public void waitOnClients() {
319             log.writeMessage("Waiting for clients to disconnect", false);
320             Object[] clients = null;
321             synchronized (_clients) {
322                 clients = _clients.toArray();
323                 if (clients == null || clients.length == 0) {
324                     return;
325                 }
326             }
327             boolean loop = true;
328             while (loop) {
329                 loop = false;
330                 for (int x = 0; x < clients.length; x++) {
331                     Session c = (Session) clients[x];
332                     if (c.isConnected()) {
333                         loop = true;
334                     }
335                 }
336             }
337         }
338
339         /**
340          * Closes the connection to all the clients, used for force
               quitting the server.
341          */
342         public void closeAllClients() {
343             log.writeMessage("Disconnecting all clients", false);
344             synchronized (_clients) {
345                 Iterator<Session> it = _clients.iterator();
346                 while (it.hasNext()) {
347                     it.next().forceQuit();
348                 }
349             }
350         }
351
352         /**
353          * Gets the Data instance
354          * @return the instance of the Data class
355          */
356         public Data getData(){
357             return this.data;
358         }
359
360         /**
361          * Adds users to the list if they're new
362          * @param user the user's name
```

```
363      * @return true if user is new, false if already registered
364      */
365     public boolean addUser(String user) {
366         synchronized (users) {
367             if (users.contains(user)) {
368                 return false;
369             } else {
370                 users.add(user);
371                 return true;
372             }
373         }
374     }
375 
376     /**
377      * Gets the Strings detailing the offers
378      * @return each offer as a String in the array
379      */
380     public String[] getOffers(){
381         return this.offers;
382     }
383 }
384 
```

```
  1 /*
  2  * To change this template, choose Tools | Templates
  3  * and open the template in the editor.
  4  */
  5 package Server;
  6 
  7 import java.io.IOException;
  8 import java.io.ObjectInputStream;
  9 import java.io.ObjectOutputStream;
 10 import java.net.Socket;
 11 import java.util.Arrays;
 12 import java.util.LinkedList;
 13 import shared.Booking;
 14 import shared.Film;
 15 import shared.Request;
 16 import shared.Response;
 17 
 18 /**
 19  * The client's threaded session with the server
 20  * @author Robert
 21  */
 22 public class Session extends Thread {
 23 
 24     private Server server;
 25     private Data data;
 26     private String name;
 27     private Socket IP;
 28     private ObjectInputStream in;
 29     private ObjectOutputStream out;
 30     private LinkedList<Booking> reservations;
 31     private boolean quit;
 32 
 33     /**
 34      * Creates a session ready for the thread to be started.
 35      * @param ip a socket to the client
 36      * @param s the instance of the server
 37      * @throws IOException if any setup error occurs
 38      */
```

```
39      public Session(Socket ip, Server s) throws IOException {
40          server = s;
41          data = server.getData();
42          IP = ip;
43          out = new ObjectOutputStream(IP.getOutputStream());
44          in = new ObjectInputStream(IP.getInputStream());
45          quit = false;
46          reservations = new LinkedList<Booking>();
47      }
48
49      /**
50       * Loop of interaction with the server.
51       * Parse requests, process data, write back a response.
52       */
53      @Override
54      synchronized public void run() {
55          try {
56              IP.setSoTimeout(Server.TIMEOUT_BLOCK);
57              name = (String) in.readObject();
58              this.setName(name + ":Thread");
59              server.log.writeEvent("Connected: " + name, false);
60              if (server.addUser(name)) {
61                  out.writeObject(null);
62              } else {
63                  Booking[] b = data.getReservations(name);
64                  if (b != null) {
65                      reservations.addAll(Arrays.asList(b));
66                      out.writeObject(reservations.toArray());
67                  } else {
68                      out.writeObject(null);
69                  }
70              }
71              //_ip.setSoTimeout(Server.TIMEOUT);
72          } catch (IOException ex) {
73              if(name == null) {name = "Unknown"; }
74              server.log.writeError(name + " session: " +
                                        ex.getMessage(), true);
75              return;
76          } catch (ClassNotFoundException ex) {
77              if(name == null) {name = "Unknown"; }
78              server.log.writeError(name + " session: " +
                                        ex.getMessage(), true);
79              return;
80          }
81
82          while (!quit) {
83              try {
84                  Request req = (Request) in.readObject();
85
86                  String command = req.getRequest();
87                  server.log.writeEvent(name + " session: requested " +
                                            command, false);
88
89                  String name = req.getName();
90                  String film = req.getFilm();
91                  String date = req.getDate();
92                  String time = req.getTime();
93                  int seats = req.getSeats();
94                  int newSeats = req.getNewSeats();
95
96                  Response r = new Response();
```

```
 97
 98                 if (command.equals(Request.MAKE)) {
 99                     if (data.makeReservation(name, film, date, time,
                            seats)) {
100                         r.setSuccess(true);
101                     } else {
102                         r.setSuccess(false);
103                         r.setReason("Capacity of the film is full. Try
                                  a different showing.");
104                     }
105                 } else if (command.equals(Request.AMEND)) {
106                     if (data.changeReservation(name, film, date, time,
                            seats, newSeats)) {
107                         r.setSuccess(true);
108                     } else {
109                         r.setSuccess(false);
110                         r.setReason("Capacity of the film is full. Try
                                  a different showing.");
111                     }
112                 } else if (command.equals(Request.DELETE)) {
113                     data.cancelReservation(name, film, date, time,
                                            seats);
114                     r.setSuccess(true);
115
116                 } else if (command.equals(Request.REFRESH_OFFERS)) {
117                     if (data.getOffers() == null) {
118                         r.setSuccess(false);
119                         r.setReason("Offers not found");
120                     } else {
121                         r.setResponse(data.getOffers());
122                         r.setSuccess(true);
123                     }
124                 } else if (command.equals(Request.MY_RESERVATIONS)) {
125                     Booking[] b = data.getReservations(this.name);
126                     if (b != null) {
127                         r.setSuccess(true);
128                         r.setResponseObjects(b);
129                         reservations.removeAll(reservations);
130                         reservations.addAll(Arrays.asList(b));
131                     } else {
132                         r.setSuccess(false);
133                         r.setResponseObjects(null);
134                         r.setReason("No reservations found");
135                     }
136                 } else if(command.equals(Request.FILMS)){
137                     r.setResponseObjects(data.getFilmNames());
138                     r.setSuccess(true);
139
140                 } else if (command.equals(Request.FILM_DATES)){
141
142                     Object[] films = data.findFilms(film);
143                     if (films != null) {
144                         String[] dates = new String[films.length];
145                         for (int x = 0; x < dates.length; x++) {
146                             dates[x] = ((Film) films[x]).getDate();
147                         }
148                         r.setResponseObjects(dates);
149                         r.setSuccess(true);
150                     } else {
151                         r.setSuccess(false);
152                         r.setReason("No dates found for the specified
```

```
                                    film.");
153                      }
154                  } else if (command.equals(Request.FILM_DATE_TIMES)){
155                      Object[] films = data.findFilms(film, date);
156                      if(films!=null){
157                      String[] times = new String[films.length];
158                      for(int x=0;x<times.length;x++){
159                          times[x] = ((Film)films[x]).getTime();
160                      }
161                      r.setResponseObjects(times);
162                      r.setSuccess(true);
163                      } else {
164                          r.setSuccess(false);
165                          r.setReason("No dates found for the specified
                                    film at the specified time.");
166                      }
167                  } else if
                         (command.equals(Request.FILM_DATE_TIME_SEATS)) {
168                      String[] seatsStrings;
169                      Film seatsFilm = data.findFilm(film, date, time);
170                      if (seatsFilm != null) {
171                          if (seatsFilm.space() == 0) {
172                              seatsStrings = new
                                         String[seatsFilm.getCapacity()];
173                              for (int x = 0; x < seatsStrings.length;
                                                                   x++) {
174                                  seatsStrings[x] = Integer.toString(x +
                                                                   1);
175                              }
176                          } else {
177                              seatsStrings = new
                                               String[seatsFilm.space()];
178                              for (int x = 0; x < seatsStrings.length;
                                                                   x++) {
179                                  seatsStrings[x] = Integer.toString(x +
                                                                   1);
180                              }
181                          }
182
183
184                      r.setResponseObjects(seatsStrings);
185                      r.setSuccess(true);
186                      } else {
187                          r.setSuccess(false);
188                          r.setReason("Could not find film");
189                      }
190
191                  }else if (command.equals(Request.LOG_OFF)) {
192                      server.removeClient(this);
193                      quit = true;
194                      return;
195                  } else {
196                      r.setSuccess(false);
197                      r.setReason("Command not understood by the
                                server");
198                  }
199
200                  out.writeObject(r);
201              } catch (IOException e) {
202                 server.log.writeError(name + " session: " +
                                     e.getMessage(), true);
```

```
203                } catch (ClassNotFoundException cnf) {
204                    server.log.writeError(name + " session: " +
                                              cnf.getMessage(), true);
205                }
206            }
207        }
208
209        /**
210         * Is the session connected?
211         * @return if the session is connected
212         */
213        public boolean isConnected() {
214            return !quit;
215        }
216
217        /**
218         * Close the socket.
219         */
220        public synchronized void forceQuit() {
221            try {
222                IP.close();
223            } catch (IOException e) {
224                server.log.writeError(name + " session: ", false);
225            }
226        }
227    }
228
```

```
  1 /*
  2  * To change this template, choose Tools | Templates
  3  * and open the template in the editor.
  4  */
  5 package Server;
  6
  7 import java.io.BufferedWriter;
  8 import java.io.IOException;
  9 import java.io.OutputStreamWriter;
 10 import java.net.ServerSocket;
 11 import java.net.Socket;
 12 import java.util.Collections;
 13 import java.util.LinkedList;
 14 import java.util.List;
 15
 16 /**
 17  * The thread that deals with urgent messages.
 18  * @author Robert
 19  */
 20 public class UrgentMsgThread extends Thread {
 21
 22     private Server server;
 23     private ServerSocket soc;
 24     private List<Socket> clients;
 25
 26     /**
 27      * constructs a new urgent message thread ready to be started
 28      * @param server the server's instance
 29      */
 30     public UrgentMsgThread(Server server) {
 31         this.setName("URGNT Thread");
 32         this.server = server;
 33         try {
```

```
34              soc = new ServerSocket(2001);
35              soc.setSoTimeout(Server.TIMEOUT);
36          } catch (IOException e) {
37              server.log.writeError("Urgent Messages: " + e.getMessage(),
                                                                    true);
38          }
39          clients = Collections.synchronizedList(new
                                            LinkedList<Socket>());
40      }
41
42      /**
43       * Thread to gets new connections
44       */
45      @Override
46      public void run() {
47          if (soc == null) {
48              return;
49          }
50          while (!server.quitting()) {
51              Socket cl = null;
52              try {
53                  cl = soc.accept();
54              } catch (IOException e) {
55                  continue;
56              }
57              synchronized(clients){
58              clients.add(cl);}
59          }
60      }
61
62      /**
63       * Send an urgent message to all clients.
64       * @param msg message to send to all clients
65       */
66      public synchronized void send(String msg) {
67              Object[] socs = null;
68              try{
69              socs = clients.toArray();
70              } catch (ClassCastException cce){
71                  server.log.writeError("Urgent Messages: " +
                                        cce.getMessage(), true);
72              }
73              UMSender m = new UMSender(socs, msg);
74              m.start();
75
76      }
77
78      /**
79       * Urgent message sender thread
80       */
81      class UMSender extends Thread {
82
83          private Object[] clients;
84          private String message;
85
86          public UMSender(Object[] c, String m) {
87              this.setName("UM Sending: " + m);
88              clients = c;
89              message = m;
90          }
91
```

```
 92         @Override
 93         public void run() {
 94             if(clients == null){ return; }
 95             for (int x = 0; x < clients.length; x++) {
 96                 Socket s = (Socket) clients[x];
 97                 try {
 98                     BufferedWriter bw = new BufferedWriter(new
                                 OutputStreamWriter(s.getOutputStream()));
 99                     bw.write(message);
100                     bw.newLine();
101                     bw.flush();
102                 } catch (IOException e) {
103                     continue;
104                 }
105             }
106 
107         }
108     }
109 }
110 
```

3.5 Commented Shared Code

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package shared;
 6 
 7 import java.io.Serializable;
 8 
 9 /**
10  * The storage class for a reservation
11  * @author Robert
12  */
13 public class Booking implements Serializable {
14     private static final long serialVersionUID = 1L;
15     private String customerName;
16     private Film film;
17     private int seats;
18 
19     /**
20      * constructs a new instance of a booking, setting instance fields
21      * @param name the customer's name
22      * @param film a Film object for the reservation
23      * @param no the number of seats
24      */
25     public Booking(String name, Film film, int no) {
26         this.customerName = name;
27         this.film = film;
28         this.seats = no;
29     }
30 
31     /**
32      * Get the name of the customer from this instance
33      * @return the customer's name
34      */
35     public String getName(){
36         return customerName;
37     }
```

```
38
39      /**
40       * Returns the film this is a reservation of
41       * @return a Film object
42       */
43      public Film getFilm(){
44          return this.film;
45      }
46
47      /**
48       * Get the number of seats reserved in this booking
49       * @return the number of seats
50       */
51      public int getSeats(){
52          return this.seats;
53      }
54  }
55
```

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package shared;
 6
 7 import java.io.Serializable;
 8
 9 /**
10  * The storage class for a Film
11  * @author Robert
12  */
13 public class Film implements Serializable{
14     private static final long serialVersionUID = 1L;
15     private String name;
16     private String time;
17     private int capacity;
18     private int booked;
19     private String date;
20
21     /**
22      * Constructs a new film
23      * @param name the films name
24      * @param time the films time
25      * @param capacity  the films capacity
26      */
27     public Film(String name, String time, int capacity){
28         this.name = name;
29         this.time = time;
30         this.capacity = capacity;
31         this.booked = 0;
32     }
33
34     /**
35      * Constructs a new film
36      * @param name the films name
37      * @param date the films date
38      * @param time the films time
39      * @param capacity the films capacity
40      * @param booked the number of seats booked already
41      */
42     public Film(String name, String date, String time, int capacity,
```

```
                    int booked){
43          this.name = name;
44          this.date = date;
45          this.time = time;
46          this.capacity = capacity;
47          this.booked = booked;
48      }
49
50      /**
51       * Get this films name
52       * @return the films name
53       */
54      public String getName(){
55          return this.name;
56      }
57
58      /**
59       * Get this films date
60       * @return the films date
61       */
62      public String getDate(){
63          return this.date;
64      }
65
66      /**
67       * Get this films capacity
68       * @return the films capacity
69       */
70      public int getCapacity(){
71          return this.capacity;
72      }
73
74      /**
75       * Get the number of seats booked for this film
76       * @return the number of seats booked
77       */
78      public int getBooked(){
79          return this.booked;
80      }
81
82      /**
83       * The amount of free space in this film
84       * @return the number of free seats
85       */
86      public int space(){
87          return this.capacity - this.booked;
88      }
89
90      /**
91       * Book the number of seats passed in. This will return false if
           there is not enough space to carry out the request
92       * @param c the number of seats to book
93       * @return true if booked, false if not
94       */
95      public boolean book(int c){
96          if(this.space()-c < 0){
97              return false;
98          } else {
99              this.booked += c;
100             return true;
101         }
```

```
102     }
103 
104     /**
105      * Frees up the number of seats passed in
106      * @param c the number of seats
107      */
108     public void free(int c){
109         this.booked -= c;
110     }
111 
112     /**
113      * Converts this Film instance to a string where each field is
         separated by a comma
114      * @return this instance to a comma separated string
115      */
116     @Override
117     public String toString(){
118         return this.name + "," + this.date + ","
119                 + this.time + "," + this.capacity + "," + this.booked;
120     }
121 
122     /**
123      * Gets this films time
124      * @return the films time
125      */
126     public String getTime() {
127        return this.time;
128     }
129 
130 }
131 
```

```
  1 /*
  2  * To change this template, choose Tools | Templates
  3  * and open the template in the editor.
  4  */
  5 package shared;
  6 
  7 import java.io.Serializable;
  8 
  9 /**
 10  * The class that the client will send to the server to request a
       command to be run.
 11  * @author Robert
 12  */
 13 public class Request implements Serializable {
 14 
 15     /**
 16      * static field to be passed into the constructor of a new Request.
 17      */
 18     public static final String LOG_OFF = "LOG_OFF";
 19     /**
 20      * static field to be passed into the constructor of a new Request.
 21      */
 22     public static final String MAKE = "MAKE_RESERVATION";
 23     /**
 24      * static field to be passed into the constructor of a new Request.
 25      */
 26     public static final String AMEND = "CHANGE_RESERVATON";
 27     /**
 28      * static field to be passed into the constructor of a new Request.
```

```
29      */
30      public static final String DELETE = "DELETE_RESERVATION";
31      /**
32       * static field to be passed into the constructor of a new Request.
33       */
34      public static final String REFRESH_OFFERS =
        "REFRESH_SPECIAL_OFFERS";
35      /**
36       * static field to be passed into the constructor of a new Request.
37       */
38      public static final String FILMS = "GET_ALL_FILMS";
39      /**
40       * static field to be passed into the constructor of a new Request.
41       */
42      public static final String MY_RESERVATIONS =
        "GET_ALL_MY_RESERVATIONS";
43      /**
44       * static field to be passed into the constructor of a new Request.
45       */
46      public static final String FILM_DATES = "GET_FILM_DATES";
47      /**
48       * static field to be passed into the constructor of a new Request.
49       */
50      public static final String FILM_DATE_TIMES = "GET_FILM_DATE_TIMES";
51      /**
52       * static field to be passed into the constructor of a new Request.
53       */
54      public static final String FILM_DATE_TIME_SEATS =
        "GET_FILM_DATE_TIMESEATS";
55
56
57      private static final long serialVersionUID = 1L;
58
59      private String name;
60      private String request;
61      private String film;
62      private String date;
63      private String time;
64      private int seats;
65      private int newSeats;
66
67      /**
68       * getter for newSeats
69       * @param s number of new seats
70       */
71      public void setNewSeats(int s){
72          this.newSeats = s;
73      }
74
75      /**
76       * setter for newSeats
77       * @return number of new seats
78       */
79      public int getNewSeats(){
80          return newSeats;
81      }
82
83      /**
84       * setter for name
85       * @param n the customer name
86       */
```

```
87      public void setName(String n){
88          this.name = n;
89      }
90
91      /**
92       * getter for name
93       * @return the customer name
94       */
95      public String getName(){
96          return name;
97      }
98
99      /**
100      * getter for request. This should be one of the static Strings
101      * @return the request
102      */
103     public String getRequest() {
104         return this.request;
105     }
106
107     /**
108      * setter for request. This should be one of the static Strings
109      * @param r the request
110      */
111     public void setRequest(String r) {
112         this.request = r;
113     }
114
115     /**
116      * constructs a request. Should pass in a static string of this
           class to ensure compatibility.
117      * @param r the request. Always use one of the static Strings
118      */
119     public Request(String r) {
120         this.request = r;
121     }
122
123     /**
124      * getter for film
125      * @return the film name
126      */
127     public String getFilm() {
128         return film;
129     }
130
131     /**
132      * setter for film
133      * @param film the film name
134      */
135     public void setFilm(String film) {
136         this.film = film;
137     }
138
139     /**
140      * getter for date
141      * @return the date
142      */
143     public String getDate() {
144         return date;
145     }
146
```

```
147     /**
148      * setter for date
149      * @param date the date
150      */
151     public void setDate(String date) {
152         this.date = date;
153     }
154
155     /**
156      * getter for time
157      * @return the film
158      */
159     public String getTime() {
160         return time;
161     }
162
163     /**
164      * setter for time
165      * @param time the time
166      */
167     public void setTime(String time) {
168         this.time = time;
169     }
170
171     /**
172      * getter for seats
173      * @return the number seats
174      */
175     public int getSeats() {
176         return seats;
177     }
178
179     /**
180      * setter for seats
181      * @param seats the number of seats
182      */
183     public void setSeats(int seats) {
184         this.seats = seats;
185     }
186 }
187
```

```
 1 /*
 2  * To change this template, choose Tools | Templates
 3  * and open the template in the editor.
 4  */
 5 package shared;
 6
 7 import java.io.Serializable;
 8
 9 /**
10  * The Response class that the server will always send to the client.
11  * @author Robert
12  */
13 public class Response implements Serializable {
14
15     private static final long serialVersionUID = 1L;
16     private String response;
17     private Object[] object;
18     private boolean success;
19     private String reason;
```

```
20
21      /**
22       * construct a new response. Sets an error message in this.reason
           (as a fall back) which should be reset as a success or error
           message.
23       */
24      public Response() {
25          this.reason = "Something bad happened";
26      }
27
28      /**
29       * setter for response
30       * @param r the response
31       */
32      public void setResponse(String r) {
33          this.response = r;
34      }
35
36      /**
37       * Setter for success.
38       * @param s Set to true if response has no errors, true if an error
           occured.
39       */
40      public void setSuccess(boolean s) {
41          this.success = s;
42      }
43
44      /**
45       * getter for response
46       * @return the response
47       */
48      public String getResponse() {
49          return this.response;
50      }
51
52      /**
53       * Setter for an object array. This will be the payload for any
           response.
54       * @param o Any array that can be cast as Object
55       */
56      public void setResponseObjects(Object[] o) {
57          this.object = o;
58      }
59
60      /**
61       * Getter for the object array. This is the payload from the server.
62       * This should be parsed out as what ever objects are expected.
63       * Could be of any length and any class type.
64       * @return the object array from the server
65       */
66      public Object[] getResponseObjects() {
67          return this.object;
68      }
69
70      /**
71       * Getter for success. This will be true if the server encountered
           no errors in running the request.
72       * If this is false, check this.reason for an error message.
73       * @return true if no errors, false otherwise.
74       */
75      public boolean getSuccess() {
```

```
76          return success;
77      }
78
79      /**
80       * This should be called passing in any non-null String before
           sending back to the server.
81       * If there is no error message, pass a success message
82       * @param r a reason why the request was not successful
83       */
84      public void setReason(String r) {
85          this.reason = r;
86      }
87
88      /**
89       * Get the reason why a request did not succeed. If this.success is
           true then this should contain a success message.
90       * @return the reason why the request did not succeed
91       */
92      public String getReason() {
93          return reason;
94      }
95  }
96
```

4 Explanation of the System Design

4.1 Nature of the Network Connection

The type of network connection chosen for the system was a connection using Streaming Sockets utilising a TCP connection with an Object Input and Output Stream wrapper. This enables serialised objects to be written and read from the stream. Using this method makes sending data that is of an unknown size easier as it can be encapsulated in an object.

The server holds an open connection between the server and the client. This method was chosen to allow whole objects to be transferred. The server creates a new Session (which extends thread) in which the connection to the client is maintained.

The system creates connections to clients on the fly; this allows an improved level of flexibility and scalability in comparison to a pooled connection style. If pooled connections were used in the connections between the server and clients would be limited to the size assigned to the pool. On the fly on the other hand has more elasticity creating connections when needed.

Using this style of network connection supports the protocol that was devised for this project. Two classes, Request and Response, were designed for sending to and from the server and client. A formalised and standardised method of communication will simplify synchronising the two programs. The client can set flags, messages, and payloads in the Request object ready to be received and interpreted by the server. Then the server will set flags, messages, and payloads can be set in the Response object to be received and processed by the client.

4.2 Server Data Structure

The session stores the name of the user and the server holds a linked list of sessions. The reason for using a linked list is that linked lists provide a higher degree of flexibility in comparison to static arrays when it comes to adjustments to the structure. Static arrays have a fixed size and do not provide this level of flexibility. Linked lists are not stored in contiguous memory locations. In a linked list every node contains an element and a link to the next element. The allocation is not static instead it is dynamic. Another disadvantage of using static arrays is the complexity of position-based insertion; if you wanted to insert an element at a position already in use, the elements in the array

would need shifting to allow space to insert the new element at the desired position. Whereas linked lists allow insertion at arbitrary points in the list.

A Film class and a Booking class were designed to hold and pass around the data relevant for each entity. This made manipulating and storing the data much easier.

4.3 Client Server asynchronous communication

To deal with asynchronous communication the server created a separate thread that sets up new a listening port to accept urgent message connections from clients. The client connects to this separate listening port and waits for the urgent message in the new thread and updates the urgent message GUI JFrame. The asynchronous communication uses a separate thread and listening port to ensure that urgent messages do not get mixed up with normal request/response messages due to the asynchronous nature of them being able to come at any point. The protocol could have been designed to incorporate the urgent message information but this approach seemed overly complicated and so was not implemented.

4.4 Additional Features and Enhancements

Extra features have been added outside the requirements of the specification. A description of these features is listed below.

4.4.1 Graceful Quit

A graceful quit feature has been added to allow all clients to finish their session before shutting down the server. This works using a loop that checks all the clients in the session list until all clients have finished their sessions. Alternatively there is a force close option that terminates the server regardless of the client's state. This method iterates through the list of clients terminating their connection. The graceful quit feature ensures that any clients already connected can carry on with their session before shutting down the server but no longer accepts new clients. An urgent message is also sent to all connected clients notifying them that the server would like to shut down.

4.4.2 Server Log File

Another additional feature implemented is the server log file. The system created an instance of the log class each time the server is started. This creates new file, which is named by the current time

and date. Whilst the server is running it logs all activities such as errors, connected clients and client requests. The logging methods were designed to allow any message to be mirrored to standard out or standard error (based on the message type). When instantiating the Log object, a boolean can be passed into the constructor to specify whether to append to one log file or to create a new file named by the current time. Refer to Appendix A for an example of a log file saved by the system.

4.4.3 Graphical User Interface

To implement a Graphical user interface (GUI) in java it is necessary to use the Abstract Window Toolkit (AWT) and the Swing packages. These packages provide a variety of GUI components that can be positioned within a Frame. These components can perform different event driven actions using the listener methods.

The GUI uses a JFrame with a tabbed pane inside it. It has four panes: Create Booking, Amend Book and Delete booking. Refer to Appendix C1 for examples of the tabbed panes. The 'create booking' tab uses JCombo boxes to list: films, dates, times and number of seats. Appendix C1 shows examples of the JCombo Box. The interface uses a JButton as the submit button to send a booking request to the server as seen in Appendix C1. Other components used in the interface were the JSpinner and JTextField. See Appendix C2 & C3 for examples of the mentioned components.

4.4.4 Administration facilities

To enable easy editing of the films data in the system, an administrative command line interface has been designed which allows adding and deleting of film information from the text files. When run, this interface will ensure that it is safe to edit the text files by polling the server to see if it has been closed down. Once the server has been shut down the data will be at a consistent state and so the text files can be edited. This facility allows a film to be chosen from a list to be deleted, or a new film to be added.

4.4.5 Amend Bookings

For a user-friendly client, an amend booking feature was added so that users can quickly and easily alter the number of seats they would like for a specified booking. The list of bookings is requested from the server when the tab is switched to so that the information is always up to date. The stepper then allows the user to choose a new amount of seats. This feature was implemented

by utilising the delete and make booking methods that had already been designed for the respective features.

6 References

GitHub Help. (2012). GitHub Bootcamp. Available: http://help.github.com/. Last accessed 23/04/2012.

Read, N and Shippey, R. (2012). Systems Software JavaDocs. Available: http://robertshippey.github.com/Task2/. Last accessed 23/04/2012.

Read, N and Shippey, R. (2012). Systems Software Repository. Available: https://github.com/RobertShippey/Task2. Last accessed 23/04/2012.

7 Appendices

Appendix A: GitHub Help Sheet – Systems Software

Intro

GitHub is a version control system (VCS). This means that, used correctly, it will keep a history of all changes to files. Actually it's a distributed VCS, which means that the repository you have locally is a whole repository whereas other systems only keep recent changes locally. That allows you to easily view past revisions of a file, and revert to previous version and manage different 'branches' of changes.

Before attempting to use GitHub for Systems Software coursework, I would suggest running through the introduction that GitHub provides. It will walk you through the extreme basics of installing, setting up your SSH keys, and other things. This document is purely meant as a quick reference for common tasks.

Basics

Cloning

As the Systems Software repository is already created, you only have to clone them to your local machine.

```
$ git clone https://github.com/RobertShippey/Task2.git
$ cd Task2
```

This 'git clone' downloads all the relevant repo files into a new folder of the same name. You can then cd into that folder.

Branches

Can you start making edits to the code now? Yes, and no. You could, but you would be making edits on the default branch. A branch is a sequence of commits that are based on code from one time, but are separate from other commits. Standard practice dictates that master (this is what we call the default branch) is always stable and edits are made in branches of master. This helps to make sure your changes reflect up to date code, and you don't accidentally break the system with a bad commit.

```
$ git checkout master
$ git branch mynewbranch
```

By checking out master, you're making sure that your new branch will be based on stable code. You create your new branch with a name that describes what you plan to change. Now you've made it, checkout your branch to switch to it.

```
$ git checkout mynewbranch
```

Committing

After you've made some changes to your code, you'll want to commit them locally. You will remember this concept from SDI1 last year. Firstly you need to stage files before you commit (akin to ticking the checkbox next to your files in the Tortoise Commit pane). You only stage the files that you have changed to stop commits from getting too big.

```
$ git add mychangedfile.java anotherchangedfile.java
```

You can check which files have been staged with:

```
$ git status
```

This will show you files that have been modified and if they are staged or not.

Once all the files you have changed are staged you can commit your changes locally. Remember, VCS's only save changes to files not the whole file every time. This saves space and allows branches to be merged with relative ease.

```
$ git commit -m "improved search algorithm"
```

I suppose some explanation of that is needed. The -m flag means that you want to add a message to your commit, this is highly recommended to let others know what you've been up to, the string in double quotes is your message. Using this method, your commit message is limited to about 50 chars. If you want to write more about what you've done you can use a text file as your commit message.

For this, keep line widths to 50 chars for the title and 70 for the descriptions. Also, the description and title should have one blank line between them. Now you can use the -F flag to use the file you've created as the message.

```
$ git commit -F message.txt
```

Push and Pull

You can commit locally as many times as you like, and it's better to commit more often than less often. I'd recommend getting into the habit of committing regularly because it makes you think about what changed you've actually made. After a while though, you want to push your commits to the server. This makes your changes available to everyone else and obviously keep another copy of the code incase your hardware fails. Here origin is the name of the server to push to (automatically named) which is the same as what we cloned from. Also, the -u just keeps all branches up to date.

```
$ git push -u origin mynewbranch
```

If you want to update your local repository with changes that others have made then you use the pull command. This should automatically update all the branches that you've been working with.

```
$ git pull
```

If you want to work on a branch that someone else has been working on, you can check that out in the same way as before.

```
$ git checkout someoneElsesBranch
```

This will pull the commits from the server because you don't already have them locally.

You should remember clone, pull, commit, and pull from SDI1 last year. We used Mercurial which is another VCS so the ideas should be quite similar.

That should cover your minimal requirements. As mentioned, go through GitHubs bootcamp because it explains things a lot clearer than I can. As a slightly relevant side note, all the help pages on GitHub are stores in a repo that the community can clone, branch, fork, commit, pull, update, push.

Advanced

Pull Requests

Okay, so you've got to the point where the branch you've been working on is finished and tested or you're stuck on something and would really like other people to look at your code and add some commits. This is where GitHub.com's Pull Request feature shines.

Pull Requests were built as a way of controlling how people merge code into others branches but they're also a great way of getting feedback.
Log on, navigate to the repo, and switch to the branch in question
Hit the 'Pull Request' button
Ensure that your commits are going 'into master from <yourBranch>'
Write a description, mention that you think your new feature is stable or that you can't get something to work and assign people if you like.
Then hit send.
Now othwe people (namely; me) will see that you've submitted a pull request and look at the things you've changed. They can commit to your branch, which will show up in the pull request. They can add comments on the website to give feedback. If you were proposing a new features to be added into master then once someone has reviewed your work and given you the thumbs up you can hit 'Merge pull request'. Having someone approve you merge into master isn't required but it's very good practice. GitHub will then automatically take your commits and apply all the changes to the master branch, therefore adding your new feature or bug fix to the stable branch.
If everyone gets into this habit of creating a branch, making edits, pull request, review, merge, then we will have a very smooth and clean development process, and the only thing that any one person can cock up is their own branch.

Issues

As you know, we're expert coders and eagle eyes debuggers so we won't ever have any problems with our code. LOLJKS, we're human. But luckily, GitHub has another features on it's website to help us mere mortals identify and detail the problems that will arise in the code. GitHub call it Issues, I call it a godsend.

If you see something wrong someone else's code, or have an issue in yours that you think someone else could help with you can create a new Issue on the repo to let them know.

People can commit to the issue and add comments to discuss the problem (it's not a slagging match about peoples code). The web interface is simple so I won't patronise you by explaining it. What I will highlight though, is how to associate a commit with an issue. Each issue has a number which you can see from the URL on GitHub, if you include #n in your commit message GitHub will kindly mention your changes in the issue page. Thank you GitHub.

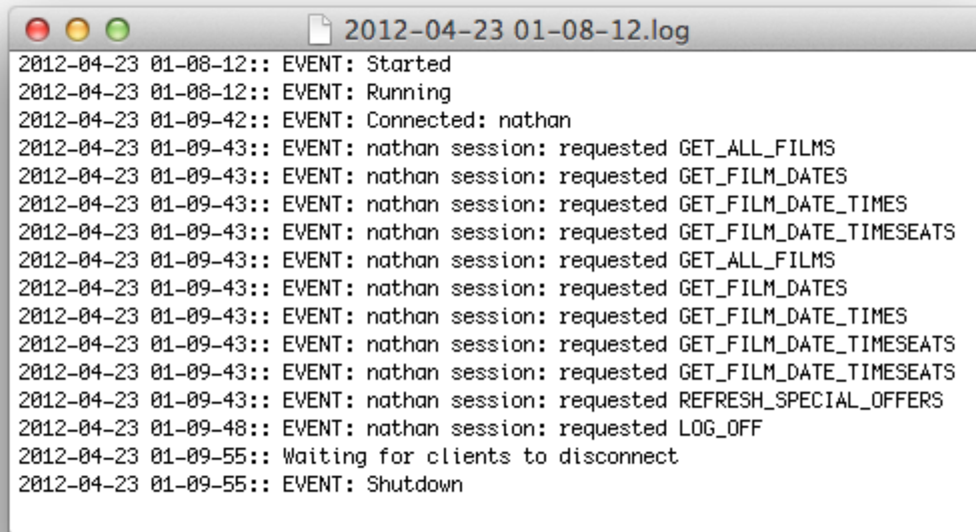
If you're sure that your commit fixes the problem you can mention 'closes #3' in your commit message and the Issue will say that your commit fixed it and close the discussion so that everyone can buy you a pint (no promises).

Outro

Places that you can get help are (in order of preference); GitHub help, Google, and Robert. Good luck and happy coding!

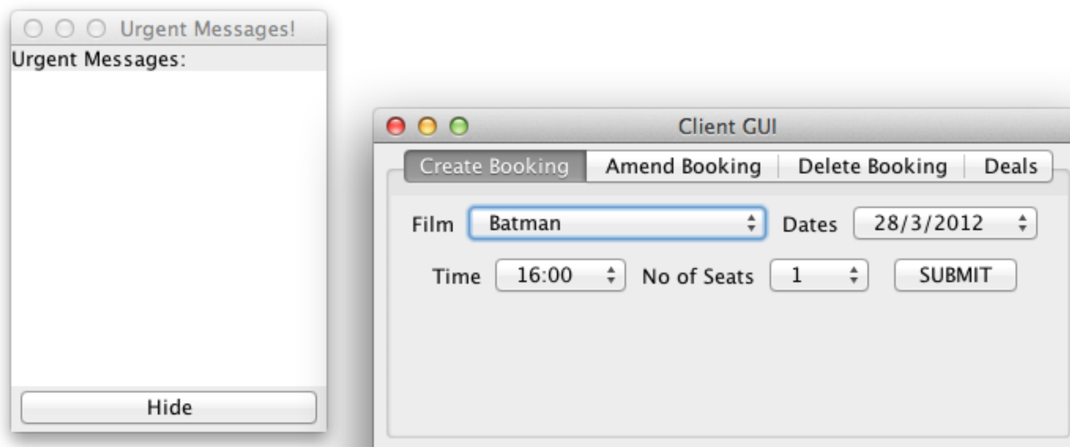
return 0;

Appendix B: Example server log



```
2012-04-23 01-08-12:: EVENT: Started
2012-04-23 01-08-12:: EVENT: Running
2012-04-23 01-09-42:: EVENT: Connected: nathan
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_ALL_FILMS
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATES
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATE_TIMES
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATE_TIMESEATS
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_ALL_FILMS
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATES
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATE_TIMES
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATE_TIMESEATS
2012-04-23 01-09-43:: EVENT: nathan session: requested GET_FILM_DATE_TIMESEATS
2012-04-23 01-09-43:: EVENT: nathan session: requested REFRESH_SPECIAL_OFFERS
2012-04-23 01-09-48:: EVENT: nathan session: requested LOG_OFF
2012-04-23 01-09-55:: Waiting for clients to disconnect
2012-04-23 01-09-55:: EVENT: Shutdown
```

Appendix C1: JFrame with tabbed Pane



Appendix C2: JSpinner Example

Appendix C3: JTextField Example