EEN1035 — OOP With Embedded Systems Assignment: Java GUI-based Client/Server Sensor Aggregation

Name: Mohammed AL shuaili

ID: 20106181

Date: 26/11/2024

Introduction:

This assignment involves creating a Java-based client/server application to simulate and manage environmental sensor data. The application will consist of multiple client devices, represented by GUI applications, sending simulated sensor data to a central server. The server, also GUI-based, will aggregate, analyse, and display the data using graphical representations, such as dials and charts. The simulation includes three chosen sensor types (temperature, humidity, sound levels), with clients providing updates every five seconds.

The objective is to implement this system while following specific design constraints and functional requirements, including multithreading, object serialization for communication, and data aggregation. Additional goals include implementing innovative features, maintaining user-friendly interfaces, and demonstrating testing with multiple simulated client connections.

Design:

The client interface is designed with three sliders and corresponding textboxes, each representing a sensor type (e.g., temperature, sound level, and humidity level) displayed as a percentage out of 100. A checkbox is included to simulate real-life signals; when selected, it adds random noise to the recorded sensor data before sending it to the server.

Additionally, the interface features two buttons: one for connecting to the server and another for editing the device (client) name. When the server is running and the client successfully connects, the "Connect" button changes to green to indicate successful communication. The client then sends the three sensor values as objects to the server every five seconds. The application requires two command-line arguments before execution: one for the server's IP address and another for the client's name. Figures 1 and 2 illustrate the interface's design.

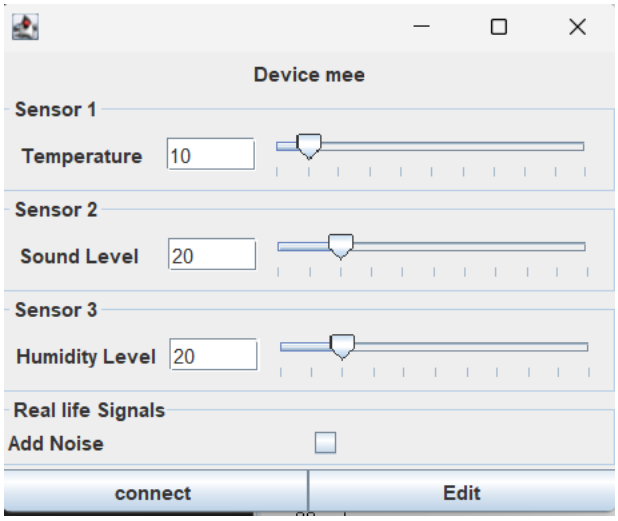

Figure 2 Client interface before it's connected

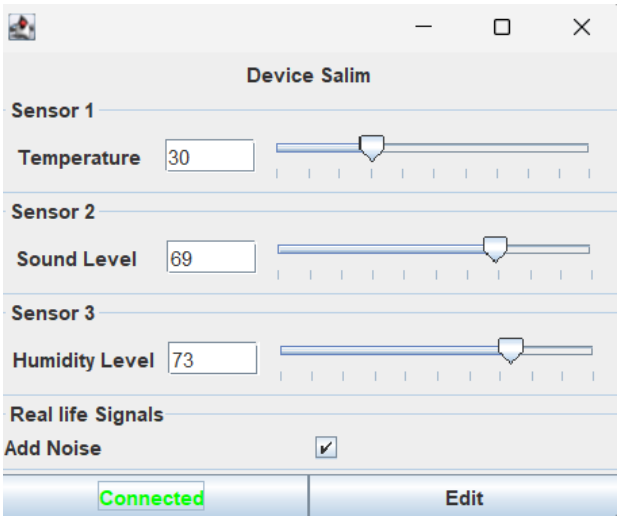

Figure 1 Client GUI after it's connected

The server GUI features a list displaying all connected clients along with an entry for the overall averages of all connected devices. It includes a button, similar to the client-side, for starting the server. When the server is actively listening for incoming connections, the button turns green. Selecting a client from the list displays the client's name and shows graphical gauges for the three sensor readings, each representing the average values of that client.

Additionally, the interface has three checkboxes, one for each sensor type. When a checkbox is selected, a plot of the last 10 readings for that sensor is displayed, with each sensor represented by a different colour. The "average" option in the client list displays the overall average of all connected clients' sensor values, as well as the average readings for each sensor, which are used for plotting. See Figures 3 and 4 for a visual representation.

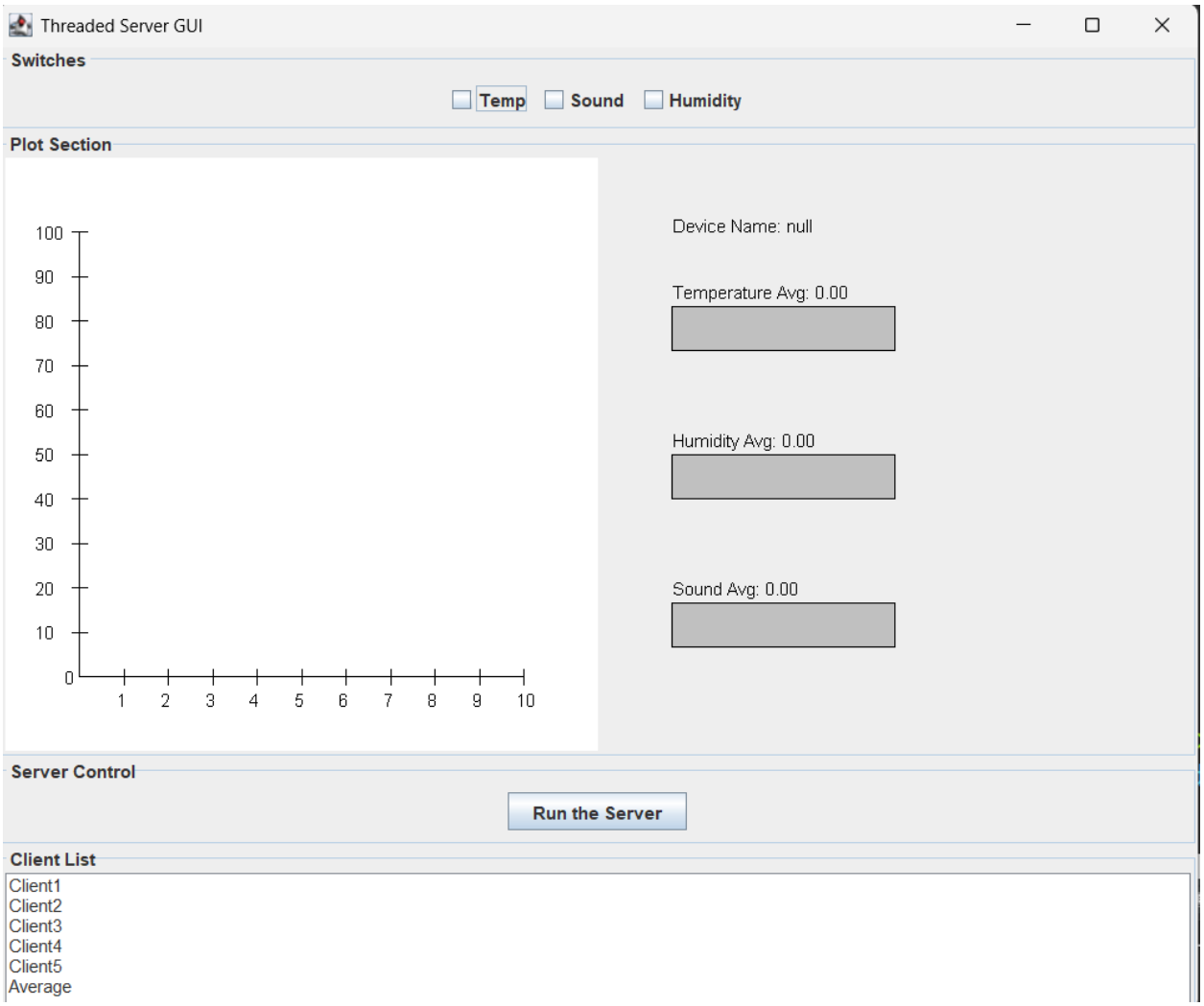

Figure 3 Server GUI Showing Client List and Default State (No Data Plotted)

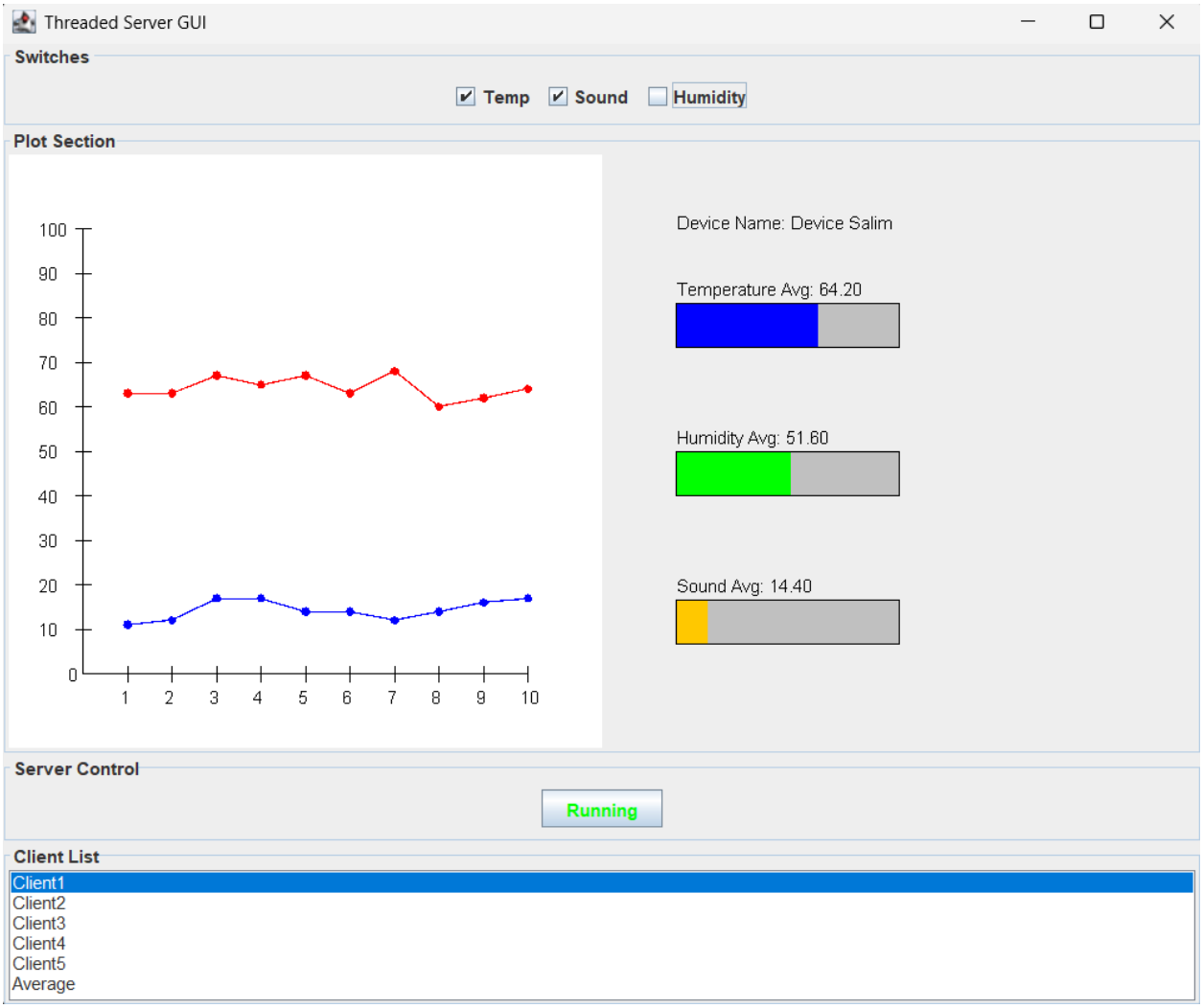

Figure 4 Server GUI Displaying Selected Client's Sensor Averages and Sensor Data Plots with Active Connections

Implementation:

- Client

The client implementation provides a GUI application for simulating environmental sensors and facilitating communication with the server. Built using Java Swing, the client interface is designed to be user-friendly, functional, and adaptable.

Key features of the client implementation include:

1. Graphical User Interface (GUI):
    - Sliders and textboxes to represent three sensor types: temperature, sound level, and humidity. Users can adjust the sliders, which automatically update the corresponding textbox, ensuring synchronization between components.

- A checkbox is provided for real-life simulation, introducing random noise to the sensor readings to simulate real-world variations. This checkbox enables dynamic toggling of the noise features.
- Two buttons are included: one to establish a connection with the server ("Connect") and another to dynamically edit the device name. The interface provides real-time feedback to users, such as changing the "Connect" button to green and updating its label to "Connected" when communication is successful.

2. Dynamic Sensor Value Input:
    - The textboxes allow direct input for sensor values as an alternative to adjusting sliders. The implementation ensures only numeric values between 0 and 100 are accepted, as the sensor readings are expressed as percentage levels.
    - If users input values exceeding this range, the program automatically resets the values to the nearest valid limit (0 or 100). This prevents erroneous data from being sent to the server, maintaining the integrity of the transmitted information.
3. Threaded Communication:
    - To ensure continuous and asynchronous updates, the client runs a background thread that sends sensor data to the server every five seconds (5000 ms). This design ensures the GUI remains responsive and user-friendly while the client communicates with the server in real time.
4. Device Name Customization:
    - Users can edit the device name dynamically through a pop-up dialog box, enabling better differentiation of clients in multi-client setups. The new name is immediately updated in the GUI and included in subsequent data transmissions.

Communication

The client communicates with the server using a socket-based connection. On pressing the "Connect" button, the client establishes a connection to the server at the specified IP address and port. It uses ObjectOutputStream to serialize and send SensorObject instances to the server, encapsulating the device name, sensor readings, and connection status.

The "Connect" button also allows users to disconnect from the server at any time. If the button is pressed while already connected, it resets the connection and updates its label and colour, indicating the disconnection status. Users can reconnect at any point, ensuring flexibility and adaptability.

- Server

The server, implemented in the ThreadedServer class, serves as the central hub for receiving, processing, and visualising sensor data from multiple clients. It uses multithreaded architecture, where each client connection is handled by a separate thread (ThreadedConnectionHandler). The server's GUI provides real-time monitoring and control capabilities, including toggling sensor plots, viewing client data, and displaying averages.

Key features of the server implementation include:

1. Dynamic Client Management:
    - The server maintains a list of connected clients, dynamically updating the GUI to reflect incoming and disconnected clients.
    - Each client has a unique identifier (ID) based on its port number, enabling the server to associate received data with the correct client.
2. Data Visualisation:
    - The server uses a custom canvas (MyCanvas) to display sensor data as graphical plots. Users can select a specific client from the list or choose "Average" to view the average sensor values across all connected clients.
    - Toggle switches allow users to enable or disable plots for temperature, sound, and humidity, providing a flexible and customizable visualisation experience.
3. Control Features:
    - The server can be started and stopped dynamically using the "Run the Server" button. When running, the button turns green and updates its label to "Running".
4. Multithreading:
    - Each client connection is assigned a dedicated thread via the ThreadedConnectionHandler class. This ensures that the server can handle multiple clients simultaneously without blocking operations.

Connection Handler

The ThreadedConnectionHandler class is responsible for managing the communication between the server and each individual client. It receives serialized SensorObject instances, processes the data, and updates the server's data structures and GUI accordingly.

Key features of the connection handler implementation include:

1. Sensor Data Processing:
    - The connection handler receives SensorObject instances from clients. Each object contains the client's name, sensor values, and connection status.
    - Sensor data is stored in a stack associated with the client's unique ID. This stack allows the server to maintain a history of up to 10 recent data points for each client, enabling moving averages and historical analysis.
2. Dynamic Stack Management:
    - The findOrCreateStack method locates the stack corresponding to a client's ID or creates a new one if no such stack exists (new client). This ensures that the server can handle new and existing clients and store all values associated with them.
    - When a client disconnects, its stack is removed, and the server GUI updates to reflect the change (the name is set to "Disconnected").
3. Real-Time GUI Updates:
    - When the connection handler receives a SensorObject, it checks if the data corresponds to the currently selected client in the server's GUI. If it does, the plot is updated immediately to reflect the new data. Otherwise, the data is stored in the appropriate stack for later retrieval or processing.
4. Graceful Disconnection:
    - When a client disconnects (indicated by its status in the SensorObject), the connection handler closes the socket and removes the client's data from the server's data structures. If the disconnected client was selected in the GUI, the plot resets to reflect the disconnection.

Communication

Communication between the server and clients is facilitated through Java sockets. The server listens for incoming connections on a specified port (5050) and spawns a new ThreadedConnectionHandler for each connection. The handler uses an ObjectInputStream to receive serialized SensorObject instances from the client.

The SensorObject structure ensures that all transmitted data is well-organized and easy to process. Each SensorObject contains an ID set by the server to uniquely identify the client.

Achievements

- Real-Time Data Handling:
    - The server processes and visualises incoming sensor data in real time, ensuring that users have up-to-date information at all times.
- Scalable Architecture:
    - The multithreaded design allows the server to handle multiple clients simultaneously, with independent data stacks for each client.
- Disconnection Handling:
    - Clients can disconnect gracefully, with their data removed from the server's GUI and data structures.

Novel Features

- Dynamic Client Visualisation:
    - Users can view data from individual clients or system-wide averages. This flexibility provides insights into both client-specific and overall performance.
- Historical Data Management:
    - The use of stacks for sensor data enables the server to maintain a history of the 10 most recent data points for each client. This supports moving averages and trend analysis.
- Interactive Sensor Selection:
    - Toggle switches allow users to customize the displayed plots by selecting which sensors (temperature, sound, humidity) to visualise.
- Dynamic Server Control:
    - The server can be started and stopped dynamically, with clear visual feedback indicating its status.
- Real-Time Input Validation:
    - Users can input sensor values directly into textboxes, but only numeric values between 0 and 100 are accepted. If users attempt to input invalid values (e.g., negative numbers, non-numeric characters, or values exceeding 100), the program resets these to the nearest valid range.
- Dynamic Connection Management:

- The "Connect" button provides users with the ability to establish or terminate the connection with the server at any time. Disconnection is handled gracefully, with a final data packet sent to inform the server of the client's status.

- Device Name Editing:
    - The ability to dynamically update the device name enhances usability in multi-client scenarios. The feature ensures that the server always receives the latest name in transmitted data packets.

Testing**:**

Tests Performed

1. Object Transmission and Serialization:
    - Initially, attempts to send objects between the client and server resulted in errors due to missing serialization in the SensorObject class. This was resolved by implementing the Serializable interface, ensuring that objects could be transmitted and deserialized correctly.
2. Data Synchronization:
    - Observed issues where some data points were missing on the server side, particularly with simultaneous updates from multiple clients. This was addressed by synchronizing access to shared resources, such as the sensorStacksList, ensuring thread-safe operations and consistent data handling.
3. Plotting Updates:
    - Encountered issues with incorrect plot updates when multiple clients were connected. This was fixed by ensuring that the plot only updates if the index of the selected client in the GUI matches the index of the stack related to the received object.
4. Server Shutdown Handling:
    - Initially, exceptions occurred when the server was closed while clients were still connected, as threads attempted to access closed or null sockets. This was resolved by implementing null checks and ensuring graceful disconnection of clients during server shutdown.
5. Single Client Connection Test:

- Verified successful communication between a single client and the server. It confirmed that sensor data was received and plotted in real time, and the client could disconnect and reconnect without issues.

6. Multiple Clients Test:
    - Simulated multiple simultaneous client connections to ensure the server could handle independent data streams. Verified that switching between clients in the GUI updated the plot correctly, and selecting "Average" displayed the overall averages of all connected clients.
7. Real-Time Updates:
    - Tested dynamic updates of sensor values on the client side, including noise simulation, to ensure accurate and immediate reflections on the server's GUI.
8. Invalid Data Handling:
    - Checked the system's behaviour when invalid sensor values (e.g., out of range or non-numeric) were input. Verified that the client reset these to valid limits and sent correct data to the server.
9. Server Start/Stop:
    - Repeatedly started and stopped the server to test proper initialization and graceful shutdown. Ensured no lingering sockets or threads remained after closure.

Does it Work Correctly with Multiple Clients?

Yes, the server works correctly with multiple simultaneous clients. Each client operates independently, and the server processes their data accurately. Switching between clients or viewing overall averages in the GUI functions as intended, with no data conflicts or delays.

Limitations

1. Scalability:
    - The system performs well with a small number of clients but may face performance challenges with a larger number due to the overhead of thread management and data synchronization.
2. Visualisation:

- The current plotting system is basic and may not effectively convey complex trends or detailed analytics. More advanced visualization methods could improve usability.

3. Disconnection Handling:
    - Abrupt client disconnections (e.g., due to network issues) may not be detected immediately, leaving stale data in the server until cleanup occurs.

Conditions Under Which It Fails

1. Server Shutdown During Active Connections:
    - If the server is closed while clients are actively transmitting data, exceptions might occur as threads attempt to access closed sockets. These are now logged and do not disrupt the server's functionality.
2. Heavy Client Load:
    - Handling a large number of clients simultaneously may result in slower response times or potential resource exhaustion due to threading overhead.

Conclusions:

This assignment provided valuable experience in building a real-time client-server system, strengthening skills in network communication, multithreading, and GUI design. It was rewarding to implement features such as dynamic visualisation, synchronized sensor inputs, and error handling, which made the system interactive and practical. Key challenges included managing thread synchronization, handling client disconnections gracefully, and maintaining real-time responsiveness, all of them required careful debugging and design adjustments.

The modular structure of the system, with separate responsibilities for the client, server, and connection handler, worked well and facilitated development. However, the visualisation could be enhanced with more advanced graphs or charts, and the system could be scaled to support a larger number of clients more efficiently. Overall, this project was an excellent opportunity to apply theoretical concepts to a practical scenario, providing insights that will be valuable for future development tasks.

Appendices.

Client.java

```java
package Assignement;

import java.net.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

import java.awt.*;
import java.awt.event.*;
import java.io.*;

@SuppressWarnings("serial")
public class Client extends JFrame implements ChangeListener, ActionListener,
WindowListener,KeyListener, Runnable{

     private static int portNumber = 5050;
    private Socket socket = null;
    private ObjectOutputStream os = null;
    private ObjectInputStream is = null;
    private JButton Connect, ChangeName;
     private JTextField Sensor1,Sensor2,Sensor3;
     private JLabel CN, temp, Sound, Humidity, noiseL;
     private JSlider slider1, slider2, slider3;
     private String serverIP;
     private Boolean Status = false;
     private Thread thread;
     private JCheckBox  noiseBox;
     // the constructor expects the IP address of the server - the port is fixed
    public Client(String serverIP, String DeviceName) {
     this.serverIP = serverIP;
               JPanel hp = new JPanel();
               hp.setLayout(new BoxLayout(hp, BoxLayout.Y_AXIS));
               this.getContentPane().add(hp);
               JPanel p;
               p = new JPanel();
               p.setLayout(new FlowLayout());
               CN = new JLabel("Device "+DeviceName);
               p.add(CN);
               hp.add(p);
               // Sensor 1 temperature
               temp = new JLabel("Temperature   ");
               Sensor1 = new JTextField(5);
               slider1 = new JSlider(-10, 100, 20);
               p = new JPanel();
               p.setLayout(new FlowLayout());
               p.setBorder(new TitledBorder("Sensor 1"));
               slider1 = new JSlider(0, 100, 10);
               slider1.setPaintTicks(true);
               slider1.setMajorTickSpacing(10);
               p.add(temp);
               p.add(Sensor1);
               p.add(slider1);
               hp.add(p);
               // Sensor 2 Sound Level
               Sound = new JLabel("Sound Level    ");
               Sensor2 = new JTextField(5);
               slider2 = new JSlider(-10, 100, 20);
               p = new JPanel();
               p.setLayout(new FlowLayout());
               p.setBorder(new TitledBorder("Sensor 2"));
               slider2 = new JSlider(0, 100, 20);
               slider2.setPaintTicks(true);
```

```
                slider2.setMajorTickSpacing(10);
                p.add(Sound);
                p.add(Sensor2);
                p.add(slider2);
                hp.add(p);
                // Sensor 3 humidity
                Humidity = new JLabel("Humidity Level ");
                Sensor3 = new JTextField(5);
                slider3 = new JSlider(-10, 100, 20);
                p = new JPanel();
                p.setLayout(new FlowLayout());
                p.setBorder(new TitledBorder("Sensor 3"));
                slider3 = new JSlider(0, 100, 20);
                slider3.setPaintTicks(true);
                slider3.setMajorTickSpacing(10);
                p.add(Humidity);
                p.add(Sensor3);
                p.add(slider3);
                hp.add(p);
                //noise
                p = new JPanel();
                p.setLayout(new GridLayout(1, 2));
                p.setBorder(new TitledBorder("Real life Signals"));
                noiseL = new JLabel("Add Noise");
                noiseBox = new JCheckBox();
                p.add(noiseL);
                p.add(noiseBox);
                hp.add(p);
                // buttons
                Connect = new JButton("connect");
                ChangeName = new JButton("Edit");
                p = new JPanel();
                p.setLayout(new GridLayout(1,2));
                p.add(Connect);
                p.add(ChangeName);
                hp.add(p);
                this.pack();
                this.slider1.addChangeListener(this);
                this.slider2.addChangeListener(this);
                this.slider3.addChangeListener(this);
                this.Connect.addActionListener(this);
                this.ChangeName.addActionListener(this);
                this.updateText(Sensor1, slider1);
                this.updateText(Sensor2, slider2);
                this.updateText(Sensor3, slider3);
                this.setVisible(true);
                this.addWindowListener(this);
                this.Sensor1.addKeyListener(this);
                this.Sensor2.addKeyListener(this);
                this.Sensor3.addKeyListener(this);
                this.thread = new Thread(this);

    }
    private void updateText(JTextField sensor, JSlider slider) {
                sensor.setText(String.valueOf(slider.getValue()));
     }
    private int addNoise(int value) {
        int noise = (int) (Math.random() * 10) - 4; // Generates a random number
        int noisyValue = value + noise;
        return noisyValue;}

    private boolean connectToServer(String serverIP) {
     try { // open a new socket to the server
                this.socket = new Socket(serverIP,portNumber);
                this.os = new ObjectOutputStream(this.socket.getOutputStream());
```

```
127.                this.is = new ObjectInputStream(this.socket.getInputStream());
128.                System.out.println("00. -> Connected to Server:" + this.socket.getInetAddress()
129.                                   + " on port: " + this.socket.getPort());
130.                System.out.println("    -> from local address: " + this.socket.getLocalAddress()
131.                                   + " and port: " + this.socket.getLocalPort());
132.      }
133.          catch (Exception e) {
134.                System.out.println("XX. Failed to Connect to the Server at port: " +
portNumber);
135.                System.out.println("    Exception: " + e.toString());
136.                return false;
137.          }
138.                return true;
139.      }
140.
141.      private void SendObject() {
142.       int tempValue = Integer.parseInt(this.Sensor1.getText());
143.          int soundValue = Integer.parseInt(this.Sensor2.getText());
144.          int humidityValue = Integer.parseInt(this.Sensor3.getText());
145.
146.          // Add noise if the checkbox is selected
147.          if (noiseBox.isSelected()) {
148.              tempValue = addNoise(tempValue);
149.              soundValue = addNoise(soundValue);
150.              humidityValue = addNoise(humidityValue);
151.          }
152.       SensorObject MyObject = new SensorObject(this.CN.getText(),tempValue,
soundValue,humidityValue, this.Status);
153.       this.send(MyObject);
154.
155.      }
156.
157.      // method to send a generic object.
158.      private void send(Object o) {
159.                try {
160.                    System.out.println("02. -> Sending an object...");
161.                    os.writeObject(o);
162.                    os.flush();
163.                }
164.           catch (Exception e) {
165.                    System.out.println("XX. Exception Occurred on Sending:" +  e.toString());
166.                }
167.      }
168.
169.
170.      public static void main(String args[])
171.      {
172.       System.out.println("**. Java Client Application - EE402 OOP Module, DCU");
173.       if(args.length>1){
174.
175.                new Client(args[0],args[1]);
176.                    //theApp.getDate();
177.                }
178.       else
179.       {
180.                System.out.println("Error: you must provide the address of the server and client
name");
181.                System.out.println("Usage is:  java Client x.x.x.x ClientName  (e.g. java Client
192.168.7.2 Mohammed)");
182.                System.out.println("      or:  java Client hostname ClientName (e.g. java Client
localhost Salim)");
183.       }
184.      }
185.
186.       @Override
```

```
187.      public void windowOpened(WindowEvent e) {}
188.
189.      @Override
190.      public void windowClosing(WindowEvent e) { if(socket !=null| Status) {Status = false;
this.SendObject();}
191.      System.exit(0);}
192.
193.      @Override
194.      public void windowClosed(WindowEvent e) {}
195.
196.      @Override
197.      public void windowIconified(WindowEvent e) {}
198.
199.      @Override
200.      public void windowDeiconified(WindowEvent e) {}
201.
202.      @Override
203.      public void windowActivated(WindowEvent e) {}
204.
205.      @Override
206.      public void windowDeactivated(WindowEvent e) {}
207.
208.      @Override
209.      public void actionPerformed(ActionEvent e) {
210.                if(e.getSource().equals(Connect)) {
211.                          if(!this.Status) {
212.                          if (!this.connectToServer(serverIP)) {
213.                          System.out.println("XX. Failed to open socket connection to: " +
serverIP);
214.                }
215.                          else {
216.                                    Status = true;
217.                                    this.Connect.setText("Connected");
218.                                    this.Connect.setForeground(Color.green);
219.                                    this.thread.start();
220.                          }}
221.                          else {
222.                                    Status = false;
223.                                    this.Connect.setText("Connect");
224.                                    this.Connect.setForeground(Color.black);
225.                                    //this.thread.suspend();
226.                                    this.SendObject();
227.                          }
228.                }
229.                if(e.getSource().equals(ChangeName)) {
230.                          String s = JOptionPane.showInputDialog(this, "Enter the device Name?",
231.                                              "A Question", JOptionPane.QUESTION_MESSAGE);
232.                          this.CN.setText("Device "+ s);
233.                }
234.      }
235.      @Override
236.      public void keyTyped(KeyEvent e) {
237.                if(!Character.isDigit(e.getKeyChar())) {
238.                          e.consume();
239.                }
240.      }
241.      @Override
242.      public void keyPressed(KeyEvent e) {
243.                if(e.getKeyCode()==KeyEvent.VK_ENTER) {
244.                          JTextField source = (JTextField) e.getSource();
245.                          int value = Integer.parseInt(source.getText());
246.                          if(value>100) {
247.                                    source.setText("100");
248.                          }
249.                          else if(value<0){
```

```
                                    source.setText("0");
                            }
                            if (source == Sensor1) {
                        slider1.setValue(value);
                    } else if (source == Sensor2) {
                        slider2.setValue(value);
                    } else if (source == Sensor3) {
                        slider3.setValue(value);
                    }
                    }
        }
        @Override
        public void keyReleased(KeyEvent e) {
        }
        @Override
        public void run() {
                    while(Status) {
                            this.SendObject();
                            try {
                                    Thread.sleep(5000);
                            }catch(InterruptedException e) {
                                    System.out.println("Thread was Interrupted");
                            }
                    }
        }
        @Override
        public void stateChanged(ChangeEvent e) {
                    if(e.getSource().equals(slider1)) {
                            updateText(Sensor1, slider1);
                    }
                    if(e.getSource().equals(slider2)) {
                            updateText(Sensor2, slider2);
                    }
                    if(e.getSource().equals(slider3)) {
                            updateText(Sensor3, slider3);
                    }
        }

}

```

SensorObject.java

```
package Assignement;

import java.io.Serializable;

@SuppressWarnings("serial")
public class SensorObject implements Serializable {
        private String Name;
        private Integer ID, V1, V2, V3;
        private Boolean s;
        public SensorObject(String Name, Integer V1, Integer V2, Integer V3, Boolean s) {
                    this.Name = Name; this.V1= V1; this.V2= V2; this.V3= V3; this.s = s;

        }
      public String getName() {
          return Name;
      }
      public Boolean getStatus() {
        return s;
      }
      public void setID(int ID) {
        this.ID = ID;
      }
```

```
23.     public Integer getID() {
24.       return this.ID;
25.     }
26.     public int getValue(int SensorNm) {
27.       if (SensorNm ==1) {return V1;}
28.       if (SensorNm ==2) {return V2;}
29.       if (SensorNm ==3) {return V3;}
30.         return 0;
31.     }
32.
33. }
34.
```

ThreadedServer.java

```
 1. /* The Date Server Class - Written by Derek Molloy for the EE402 Module
 2.  * See: ee402.eeng.dcu.ie
 3.  */
 4.
 5. package Assignement;
 6. import java.net.*;
 7. import javax.swing.*;
 8. import java.awt.event.*;
 9. import java.awt.*;
10. import java.io.*;
11. import java.util.Stack;
12.
13. @SuppressWarnings("serial")
14. public class ThreadedServer extends JFrame implements ActionListener, WindowListener, Runnable
15. {
16.      private static int portNumber = 5050;
17.      boolean listening = true;
18.     ServerSocket serverSocket = null;
19.     private MyCanvas plot = new MyCanvas();
20.     private JButton start;
21.     private Thread thread;
```

```
22.      private List  ClientList;
23.      int currentSelectedClient = -1;
24.      private JCheckBox  tempSwitch, soundSwitch, humiditySwitch;
25.      public ThreadedServer() {
26.                // Main panel with vertical layout to stack sections
27.          JPanel mainPanel = new JPanel();
28.          mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));
29.          this.getContentPane().add(mainPanel);
30.
31.          // Section 1: Switch Panel
32.          JPanel switchPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
33.          switchPanel.setBorder(BorderFactory.createTitledBorder("Switches"));
34.          tempSwitch = new JCheckBox ("Temp");
35.          soundSwitch = new JCheckBox ("Sound");
36.          humiditySwitch =new JCheckBox ("Humidity");
37.          switchPanel.add(tempSwitch);
38.          switchPanel.add(soundSwitch);
39.          switchPanel.add(humiditySwitch);
40.          mainPanel.add(switchPanel);
41.          this.tempSwitch.addActionListener(this);
42.          this.soundSwitch.addActionListener(this);
43.          this.humiditySwitch.addActionListener(this);
44.          // Section 2: Plot Panel
45.          JPanel plotPanel = new JPanel(new BorderLayout());
46.          plotPanel.setBorder(BorderFactory.createTitledBorder("Plot Section"));
47.          plotPanel.add(plot, BorderLayout.CENTER);
48.          mainPanel.add(plotPanel);
49.
50.       // Section 3: Control Panel
51.          JPanel controlPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
52.          controlPanel.setBorder(BorderFactory.createTitledBorder("Server Control"));
53.          start = new JButton("Run the Server");
54.          this.start.addActionListener(this);
55.          controlPanel.add(start);
56.          mainPanel.add(controlPanel);
57.
58.          // Section 4: Client List Panel
59.          JPanel clientPanel = new JPanel(new BorderLayout());
60.          clientPanel.setBorder(BorderFactory.createTitledBorder("Client List"));
61.          ClientList = new List(6, false);
62.          for (int i = 0; i < 5; i++) {
63.              ClientList.add("Client" + (i + 1));
64.          }
65.          ClientList.add("Average");
66.          ClientList.addActionListener(this);
67.          clientPanel.add(ClientList, BorderLayout.CENTER);
68.          mainPanel.add(clientPanel);
69.          this.pack();
70.          this.setTitle("Threaded Server GUI");
71.          this.setVisible(true);
72.
73.
74.          this.thread = new Thread(this);
75.          this.addWindowListener(this);
76.
77.      }
78.      public void rePlot(Stack<SensorObject> stack) {
79.         plot.rePlot(stack); // Update canvas with new averages
80.      }
81.      public void FindAvg() {
82.      plot.FindAvg(ThreadedConnectionHandler.sensorStacksList);}
83.      public void disconnectClient() {
84.                plot.disconnectClient();
85.      }
86.      public static void main(String args[]) {
```

```
 87.                 new ThreadedServer();
 88.     }
 89.
 90.     @Override
 91.     public void run() {
 92.
 93.                 try
 94.         {
 95.             serverSocket = new ServerSocket(portNumber);
 96.             System.out.println("New Server has started listening on port: " + portNumber );
 97.         }
 98.         catch (IOException e)
 99.         {
100.             System.out.println("Cannot listen on port: " + portNumber + ", Exception: " + e);
101.             System.exit(1);
102.         }
103.
104.         // Server is now listening for connections or would not get to this point
105.         while (listening) // almost infinite loop - loop once for each client request
106.         {
107.             Socket clientSocket = null;
108.             try{
109.                System.out.println("**. Listening for a connection...");
110.                 clientSocket = serverSocket.accept();//get stuck here until a client connect
111.                 System.out.println("00. <- Accepted socket connection from a client: ");
112.                 System.out.println("    <- with address: " +
clientSocket.getInetAddress().toString());
113.                 System.out.println("    <- and port number: " + clientSocket.getPort());
114.                 ThreadedConnectionHandler con = new ThreadedConnectionHandler(clientSocket,
this);
115.                 con.start();
116.             }
117.             catch (IOException e){
118.                if (!listening) {
119.                     System.out.println("Server has stopped listening for connections.");
120.                     break; // Exit the loop if the server is no longer listening
121.                 }
122.                 System.out.println("XX. Accept failed: " + e.getMessage());
123.             }
124.
125.         }
126.     closeSocket();
127.     }
128.
129.     private void closeSocket() {
130.                 try
131.         {
132.             System.out.println("04. -- Closing down the server socket gracefully.");
133.             serverSocket.close();
134.         }
135.         catch (IOException e)
136.         {
137.             System.err.println("XX. Could not close server socket. " + e.getMessage());
138.         }
139.     }
140.      private void onClientSelected() {
141.             String selectedClient = ClientList.getSelectedItem();
142.             if (selectedClient != null) {
143.                 try {
144.                         int index = ClientList.getSelectedIndex();
145.                         if("Average".equals(selectedClient))
146.                         {  plot.FindAvg(ThreadedConnectionHandler.sensorStacksList);
147.                         currentSelectedClient = index ; // 5 is the avg ,
148.                         }
149.
```

```
                    else if (index >= 0 && index <
ThreadedConnectionHandler.sensorStacksList.size()) {
                        Stack<SensorObject> stack =
ThreadedConnectionHandler.sensorStacksList.get(index);
                        currentSelectedClient = index;
                        plot.rePlot(stack); // Update the canvas with the selected client's
averages
                    } else {
                        currentSelectedClient = -1;
                        plot.disconnectClient(); // No data for the selected client
                    }
                } catch (Exception e) {
                    System.err.println("Error selecting client: " + e.getMessage());
                    plot.updateAverages(0, 0, 0);
                }
            }
        }


    @Override
    public void windowOpened(WindowEvent e) {
            // TODO Auto-generated method stub

    }

    @Override
    public void windowClosing(WindowEvent e) { if (serverSocket!=null) {listening = false;
}System.exit(0);}

    @Override
    public void windowClosed(WindowEvent e) {
            // TODO Auto-generated method stub

    }

    @Override
    public void windowIconified(WindowEvent e) {
            // TODO Auto-generated method stub

    }

    @Override
    public void windowDeiconified(WindowEvent e) {
            // TODO Auto-generated method stub

    }

    @Override
    public void windowActivated(WindowEvent e) {
            // TODO Auto-generated method stub

    }

    @Override
    public void windowDeactivated(WindowEvent e) {
            // TODO Auto-generated method stub
    }

    @Override
    public void actionPerformed(ActionEvent e) {
            if(e.getSource().equals(start)) {
                    if(serverSocket ==null) {
                    listening = true;
                    start.setText("Running");
                    this.start.setForeground(Color.green);
```

```
                        this.thread.start();

                        }
                        else {
                                listening = false;
                                closeSocket();
                                start.setText("Run the Server");
                                this.start.setForeground(Color.BLACK);
                        }
                }
                if (e.getSource().equals(tempSwitch) || e.getSource().equals(soundSwitch) ||
e.getSource().equals(humiditySwitch)) {
                         plot.setPlotVisibility(tempSwitch.isSelected(),
soundSwitch.isSelected(), humiditySwitch.isSelected());

                    }
                if (e.getSource().equals(ClientList)) {
                        onClientSelected();
                }
        }
}

```

MyCanvas.java

```
package Assignement;

import java.awt.*;
import java.util.*;
import java.util.List;

@SuppressWarnings("serial")
public class MyCanvas extends Canvas {
     private double [] tempVal= new double[10];
     private double [] humVal= new double[10];
     private double [] soundVal= new double[10];
     private double tempAvg, humidityAvg, soundAvg;
    private boolean showTemp = false;
    private boolean showHumidity = false;
    private boolean showSound = false;
    private String dName;
     public MyCanvas () {

                this.setPreferredSize(new Dimension(800,400));

     }
    public void updateAverages(double temp, double sound, double humidity) {
        this.tempAvg = temp;
        this.soundAvg =  sound;
        this.humidityAvg = humidity;
        repaint();
    }
    public void rePlot(Stack<SensorObject> stack) {
     this.tempVal = new double[10];  // Reset to an empty array
        this.soundVal = new double[10];
        this.humVal = new double[10];
     double temp=0,sound=0,hum=0;
     int i=0;
     for (Object o : stack) {
                SensorObject sensor = (SensorObject) o;
                temp += sensor.getValue(1);
                sound += sensor.getValue(2);
                hum +=sensor.getValue(3);
                tempVal[i] = sensor.getValue(1); // Temperature
            humVal[i] = sensor.getValue(3); // Humidity
```

```
            soundVal[i] = sensor.getValue(2); // Sound
            i++;
            this.dName = sensor.getName();
        }
        temp = temp/stack.size();
        sound = sound/stack.size();
        hum = hum/stack.size();
        updateAverages(temp, sound, hum);
       }
       public void  disconnectClient() {
        this.dName = "Device is disconnected";
        this.tempVal = new double[10];  // Reset to an empty array
           this.soundVal = new double[10];
           this.humVal = new double[10];
           updateAverages(0,0,0);
        repaint();
       }

       public void setPlotVisibility(boolean showTemp, boolean showSound, boolean showHumidity) {
           this.showTemp = showTemp;
           this.showHumidity = showHumidity;
           this.showSound = showSound;
           repaint();
       }
       public synchronized void FindAvg(List<Stack<SensorObject>> sensorStacksList) {
           double totalTempAvg = 0, totalSoundAvg = 0, totalHumidityAvg = 0;
           int clientCount = 0;
           double [] temp= new double[10];
           double [] sound= new double[10];
           double [] hum= new double[10];
               for (Stack<SensorObject> stack : sensorStacksList) {
                   if (!stack.isEmpty()) {
                       double tempSum = 0, soundSum = 0, humSum = 0;
                       int i = 0;
                       for (SensorObject sensor : stack) {
                           tempSum += sensor.getValue(1);
                           soundSum += sensor.getValue(2);
                           humSum += sensor.getValue(3);
                           temp[i] += sensor.getValue(1);
                           sound[i]+= sensor.getValue(2);
                           hum [i] += sensor.getValue(3);
                           i++;
                       }
                       int stackSize = stack.size();
                       totalTempAvg += tempSum / stackSize;
                       totalSoundAvg += soundSum / stackSize;
                       totalHumidityAvg += humSum / stackSize;
                       clientCount++;
                   }
               }
               for(int i=0; i<temp.length;i++) {
                  temp[i] = temp[i]/clientCount;
                  sound[i] = sound[i]/clientCount;
                  hum[i] = hum[i]/clientCount;
               }
               totalTempAvg = totalTempAvg/clientCount;
               totalSoundAvg = totalSoundAvg /clientCount;
               totalHumidityAvg = totalHumidityAvg/clientCount;
               this.tempVal = temp; this.soundVal = sound; this.humVal = hum;
               this.dName = "Average";
               updateAverages(totalTempAvg, totalSoundAvg,totalHumidityAvg);
       }
        public void paint(Graphics g) {
                  int width = 400;
           int height = 400;
```

```
106.
107.         // Draw background
108.         g.setColor(Color.WHITE);
109.         g.fillRect(0, 0, width, height);
110.
111.         // Draw axes
112.         g.setColor(Color.BLACK);
113.         g.drawLine(50, height - 50, width - 50, height - 50); // x-axis
114.         g.drawLine(50, height - 50, 50, 50); // y-axis
115.         //int xAxisLength = width-100;
116.         // Draw axis labels
117.         g.drawString("0", 40, height - 45); // Origin
118.         // Draw tick marks and labels on x-axis
119.        for (int i = 1; i <= 10; i++) {
120.             int x = 50 + (i * (width - 100) / 10);
121.             g.drawLine(x, height - 55, x, height - 45);
122.             g.drawString(Integer.toString(i), x - 5, height - 30);
123.         }
124.
125.         // Draw tick marks and labels on y-axis
126.         for (int i = 1; i <= 10; i++) {
127.             int y = height - 50 - (i * (height - 100) / 10);
128.             g.drawLine(45, y, 55, y);
129.             g.drawString(Integer.toString(i * 10), 20, y + 5);
130.         }
131.         g.setColor(Color.BLACK);
132.         if(dName == "Device is disconnected") {g.setColor(Color.RED);}
133.         g.drawString("Device Name: "+ dName,width + 50,50);
134.         if (showTemp) drawPlot(g, tempVal, Color.RED);
135.         if (showHumidity) drawPlot(g, humVal, Color.GREEN);
136.         if (showSound) drawPlot(g, soundVal, Color.BLUE);
137.      // Draw the gauge for average values
138.         drawGauge(g, width + 50, 100, "Temperature Avg", tempAvg, Color.RED);
139.         drawGauge(g, width + 50, 200, "Humidity Avg", humidityAvg, Color.GREEN);
140.         drawGauge(g, width + 50, 300, "Sound Avg", soundAvg, Color.BLUE);
141.
142.      }
143.     private void drawGauge(Graphics g, int x, int y, String label, double value, Color color) {
144.         int gaugeWidth = 150;
145.         int gaugeHeight = 30;
146.
147.         // Draw background of the gauge
148.         g.setColor(Color.LIGHT_GRAY);
149.         g.fillRect(x, y, gaugeWidth, gaugeHeight);
150.
151.         // Draw filled part of the gauge based on value (scale 0-100)
152.         int filledWidth = (int) (value * gaugeWidth / 100); // Scale value to gauge width
153.         g.setColor(color);
154.         g.fillRect(x, y, filledWidth, gaugeHeight);
155.
156.         // Draw gauge border
157.         g.setColor(Color.BLACK);
158.         g.drawRect(x, y, gaugeWidth, gaugeHeight);
159.
160.         // Draw label and value
161.         g.drawString(label + ": " + String.format("%.2f", value), x, y - 5);
162.     }
163.     private void drawPlot(Graphics g, double []RecivedValues, Color color) {
164.      g.setColor(color);
165.       for (int i = 0; i < RecivedValues.length; i++) {
166.                int x = 50 + ((i+1) * (300) / 10); // Start from 1
167.               int y = (int) (400 - 50 - (RecivedValues[i] * (400 - 100) / 100)); // Scale y (0-
100)
168.               g.fillOval(x - 3, y - 3, 6, 6); // Draw point as a small circle
169.          }
```

```
170.       for (int i = 0; i < RecivedValues.length - 1; i++) {
171.             int x1 = 50 + ((i+1) * (300) / 10); // x position for the i-th value
172.             int y1 = (int) (400 - 50 - (RecivedValues[i] * (400 - 100) / 100)); // y position
for the i-th value
173.
174.             int x2 = 50 + ((i + 2) * (300) / 10); // x position for the (i+1)-th value
175.             int y2 = (int) (400 - 50 - (RecivedValues[i + 1] * (400 - 100) / 100)); // y
position for the (i+1)-th value
176.
177.             // Draw the line between the two points
178.             g.drawLine(x1, y1, x2, y2);
179.          }
180.     }
181.
182. }
183.
```

ThreadedConnectionHandeller.java

```
  1. /* The Connection Handler Class - Written by Derek Molloy for the EE402 Module
  2.  * See: ee402.eeng.dcu.ie
  3.  */
  4.
  5. package Assignement;
  6.
  7. import java.net.*;
  8. import java.util.*;
  9. import java.io.*;
 10.
 11. public class ThreadedConnectionHandler extends Thread
 12. {
 13.     private Socket clientSocket = null;                                    // Client socket
object
 14.     private ObjectInputStream is = null;                          // Input stream
 15.     private ObjectOutputStream os = null;                         // Output stream
 16.     static List<Stack<SensorObject>> sensorStacksList = new ArrayList<>();
 17.     private ThreadedServer server;
 18.      // The constructor for the connection handler
 19.     public ThreadedConnectionHandler(Socket clientSocket, ThreadedServer server) {
 20.         this.clientSocket = clientSocket;
 21.         this.server = server;
 22.     }
 23.
 24.     // Will eventually be the thread execution method - can't pass the exception back
 25.     public void run() {
 26.          try {
 27.             this.is = new ObjectInputStream(clientSocket.getInputStream());
 28.             this.os = new ObjectOutputStream(clientSocket.getOutputStream());
 29.             while (this.readCommand()) {}
 30.          }
 31.          catch (IOException e)
 32.          {
 33.                System.out.println("XX. There was a problem with the Input/Output
Communication:");
 34.             e.printStackTrace();
 35.          }
 36.     }
 37.
 38.     // Receive and process incoming string commands from client socket
 39.     private synchronized boolean readCommand() {
 40.        //Object s = null; //don't use this for the assignment use obj
 41.         try {
 42.             Object s =  is.readObject();
 43.             if(s instanceof SensorObject) {
 44.                SensorObject data = (SensorObject) s;
```

```
45.                 data.setID(clientSocket.getPort());
46.                 if(data.getStatus() == false) {
47.                           System.out.println("Closing the Socket");
48.                           this.closeSocket();
49.                           Stack<SensorObject> stack = findOrCreateStack(data.getID());
50.                           if (sensorStacksList.indexOf(stack) == server.currentSelectedClient) {
51.                                     server.disconnectClient();
52.                           }
53.                           sensorStacksList.remove(stack);
54.                           return false;
55.                 }
56.
57.                       Stack<SensorObject> stack = findOrCreateStack(data.getID());
58.                       if (stack.size()==10) {stack.remove(0);}
59.                       stack.push(data);
60.                       if (sensorStacksList.indexOf(stack) == server.currentSelectedClient) {
61.                           server.rePlot(stack);
62.                       }
63.                       else if(server.currentSelectedClient == 5) {
64.                           server.FindAvg();
65.                       }
66.                       //server.findAvg(stack);
67.
68.             } else {
69.                 System.out.println("XX. Received an unknown object type.");
70.             }
71.             System.out.println("----------------------------------------");
72.         }
73.         catch (Exception e){   // catch a general exception
74.             System.out.println("XX. Error in readCommand: " + e);
75.               this.closeSocket();
76.             return false;
77.         }
78.
79.         return true;
80.     }
81.     // don't change this even with objects
82.     // Use our custom DateTimeService Class to get the date and time
83.
84.     // Send a generic object back to the client
85.     private void send(Object o) {
86.         try {
87.             System.out.println("02. -> Sending (" + o +") to the client.");
88.             this.os.writeObject(o);
89.             this.os.flush();
90.         }
91.         catch (Exception e) {
92.             System.out.println("XX." + e.getStackTrace());
93.         }
94.     }
95.     private Stack<SensorObject> findOrCreateStack(int id) {
96.      displaySensorStacks();
97.         for (Stack<SensorObject> stack : sensorStacksList) {
98.             if (!stack.isEmpty() && stack.peek().getID() == id) {
99.                 return stack;
100.            }
101.        }
102.        // Create a new stack if not found
103.        Stack<SensorObject> newStack = new Stack<>();
104.        sensorStacksList.add(newStack);
105.        return newStack;
106.    }
107.    private void displaySensorStacks() {
108.        System.out.println("### Current Sensor Stacks ###");
109.        synchronized (sensorStacksList) {
```

```
110.             for (Stack<SensorObject> stack : sensorStacksList) {
111.                 if (!stack.isEmpty()) {
112.                     SensorObject top = stack.peek(); // Get the top object to identify the
stack
113.                     System.out.println("Stack for ID: " + top.getID());
114.                     for (SensorObject sensor : stack) {
115.                         System.out.println("    " + sensor);
116.                     }
117.                 }
118.             }
119.         }
120.         System.out.println("##############################");
121.     }
122.     // Send a pre-formatted error message to the client
123.     public void sendError(String message) {
124.         this.send("Error:" + message);      //remember a String IS-A Object!
125.     }
126.
127.     // Close the client socket
128.     public void closeSocket() { //gracefully close the socket connection
129.         try {
130.             this.os.close();
131.             this.is.close();
132.             this.clientSocket.close();
133.         }
134.         catch (Exception e) {
135.             System.out.println("XX. " + e.getStackTrace());
136.         }
137.     }
138. }
139.
```