JUST DO IT
HÖHERE ABTEILUNG FÜR
INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Laborprotokoll

IoT Visualisierung

Systemtechnik Labor
5BHIT 2017/18

Martin Wölfer

Version 0.1
Begonnen am 30. November 2017
Beendet am 14. Dezember 2017

Note:
Betreuer: WEIJ

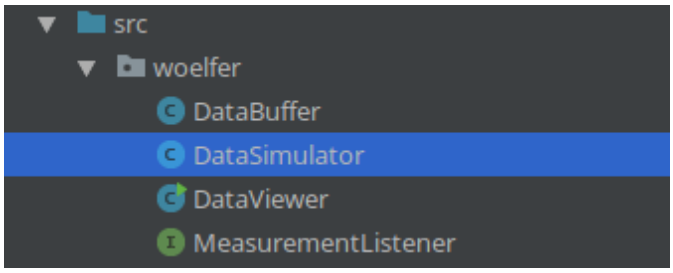
Inhaltsverzeichnis

1 Aufgabenstellung

Im Abschnitt "Übungen EK: Embedded Devices - Weiser"gibt es im Ordner Ünterlagen zu den Übungenëin Dokument namens "Datenvisualisierung.pdf". Darin ist eine Aufgabe für die Grundkompetenzen als auch eine Aufgabe für die erweiterten Kompetenzen beschrieben. Führe die Übungen der zweiten Aufgabe (erweiterte Kompetenzen) hier durch und gib dann das entsprechende Protokoll und den Code hier ab.

Es gibt auchim selben Ordner noch ein entsprechendes zip-File mit Java-Klassen, welche du benötigst. Weiters benötigst du auchdas Ergebnis der ersten Aufgabe zm Thema IoT-Visualisierung.

2 Projektaufbau



Die Klassen `Databuffer`, `DataSimulator` und `MeasurementListener` wurden bereits vorgegeben.

2.1 MeasurementListener

Das **Interface** definiert die Methode, welche aufgerufen wird sobald eine neue **Messung** eintrifft, `measurementReceived()`

Dieser wird im `DataBuffer` mit der Methode `addMeasurementListener` hinzugefügt.

Wenn nun die Methode `addList()` vom Buffer aufgerufen wird, wird automatisch `measurementReceived()` des Listeners aufgerufen.

2.2 DataBuffer

Diese Klasse würde in einem klassischen **MVC-Modell** das **Model** darstellen. Der `DataBuffer` hält die Liste, für die Werte welche simuliert werden. Der Buffer bestimmt auch die Anzahl der ausgerechneten Werte mit der Konstante `SIZE`.

2.2.1 addList

Diese Methode ”überschreibt” die momentane Liste der Werte, und ersetzt mit jener welche übergeben wird als Parameter. Zusätzlich wird noch die `measurementReceived()` Methode des Listeners aufgerufen.

2.2.2 getNext

Diese Methode kopiert das momentane Array mit allen Werten, und übergibt es an die Variable welcher übergeben wird - auch **Callback-Variable** genannt.

Der Grund dafür, dass die Liste nicht einfach **return**ed wird sondern **kopiert**, liegt darin dass Konflikte entstehen könnten, da der `DataSimulator`, als Thread, immer parallel läuft und Zugriffsprobleme entstehen könnten, da der Simulator auch auf die Liste zugreift.

2.3 DataSimulator

Der `DataSimulator` stellt einen Thread dar. Deswegen erweitert diese Klasse die Klasse `Thread` und **überschreibt** die `run()` Methode des Threads.

Diese Klasse übernimmt folgende Parameter, um Eigenschaften der Sinuskurve zu definieren:

- minFrequency (Standard: 0.5)
- maxFrequency (Standard: 5)
- deltaFrequency (Standard: 0.015)

Der Simulator nimmt zusätzlich noch eine Instanz vom Typen `DataBuffer`, in welcher er die simulierten Daten "speichert".

2.3.1 run

Wie schon erwähnt, liegt die gesamte Logik in der run Methode, da diese parallel zu den anderen Prozessen laufen kann.

Es wird nun eine `values` Liste angelegt, in welcher die Werte für eine Sinuskurve gespeichert werden. Wenn die Berechnung vollständig durchgeführt wurde und die vorübergehende Liste gefüllt ist, wird die Liste der DataBuffer-Instanz gesetzt.

Anschließend wird die Frequenz verändert, damit bei der Visualisierung eine sich änderte Sinuskurve zu sehen ist. Danach wiederholt sich alles da der gesamte Block mit einer While-true Schleife umschlossen ist.

2.4 DataViewer

Die Klasse `DataViewer` stellt ein `JComponent` dar, welches in ein `JFrame` gepackt wird um gezeichnet zu werden.

2.4.1 main

```
1  public static void main(String[] args){
       frame = new JFrame("DataViewer");
3      frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
       frame.getContentPane().add(new DataViewer());
5      frame.pack();
       frame.setSize(new Dimension(800, 400));
7      frame.setVisible(true);
   }
```

In der main-Methode, wird wie bereits erwähnt, ein JFrame erzeugt in welches, mit der Methode `frame.getContentPane().add()` das JComponent, also der `DataViewer`, hinzugefügt.

Anschließend werden noch Dimensionen und obligatorische JFrame Methoden gesetzt.

2.4.2 constructor

Zuerst wird ein `DataBuffer` Objekt erzeugt, welches anschließend mit zusätzlichen Parametern an den Simulator übergeben wird um diesen zu starten. Es wird auch eine die Liste `yPoints` erzeugt, welche die Werte des Simulators halten wird.

Ein wichtiger Schritt ist es, dem DataBuffer nun per `addMeasurementListener` einen `MeasurementListener` hinzuzufügen.

Um den Code einfacher darzustellen, wurde gleich in der Instanziierung definiert, was passieren soll wenn eine neue Messung erhalten wurde. Es soll nun die yPoints Liste gesetzt werden mit `next()` des Buffers, und das frame `repainted` werden, damit die Sinuskurve sich bei einer neuen Messung tatsächlich aktualisiert:

```
   DataBuffer db = new DataBuffer();
2  yPoints = new int[DataBuffer.SIZE];
   DataSimulator dt = new DataSimulator(0.5, 5, 0.015, db);
4  db.addMeasurementListener(new MeasurementListener(){
       @Override
6      public void measurementReceived() {
           db.getNext(yPoints);
8          frame.repaint();
       }
10 });
```

2.4.3 paint

Diese Methode wird von `JComponent` vererbt. Durch überschreiben dieser Methode, kann definiert werden was gezeichnet werden soll wenn die Komponente geladen wird. Um die Sinuskurve zu simulieren, wird das Package `GeneralPath` verwendet. Damit ist es möglich, Linien von einem Punkt zu einem anderen zu zeichnen.

Die Idee ist, da sehr viele Werte geliefert werden vom Simulator, zwischen jedem dieser Werte einen Strich zu ziehen um somit den Anschein, eine echte Sinuskurve zu zeichnen, zu erwecken.

Zuerst wird eine path Instanz erzeugt, und zum Mittelpunkt des Fenster navigiert:

```
GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD, DataBuffer.SIZE);

path.moveTo(0, frame.getHeight() / 2);
```

```
for (int i = 0; i < DataBuffer.SIZE; i++){
    path.lineTo(i*10, (yPoints[i]/70)+(this.frame.getHeight()/2));
}
```

In diesen Zeilen werden die einzelnen Linien dem Path hinzugefügt.

Die Methode `lineTo` nimmt 2 Punkte als Parameter. Es ist sich vorzustellen, der erste Punkt sei der Punkt auf der "x-Achse" und der zweite die Auswertung der "y-Achse". Wie bereits erwähnt, werden in `yPoints` die Werte des Simulators gespeichert.

Diese werden durch eine Konstante dividiert, um die Werte der Fenstergröße anzupassen, und um die Hälfte des Fenster nach unten geschoben damit die Kurve in der Mitte liegt.

Zum Schluss wird noch ein `Graphics2D` Objekt erzeugt, mit welchem tatsächlich der path gezeichnet wird:

```
Graphics2D g2 = (Graphics2D) g;
g2.draw(path);
```

3 Ergebnis

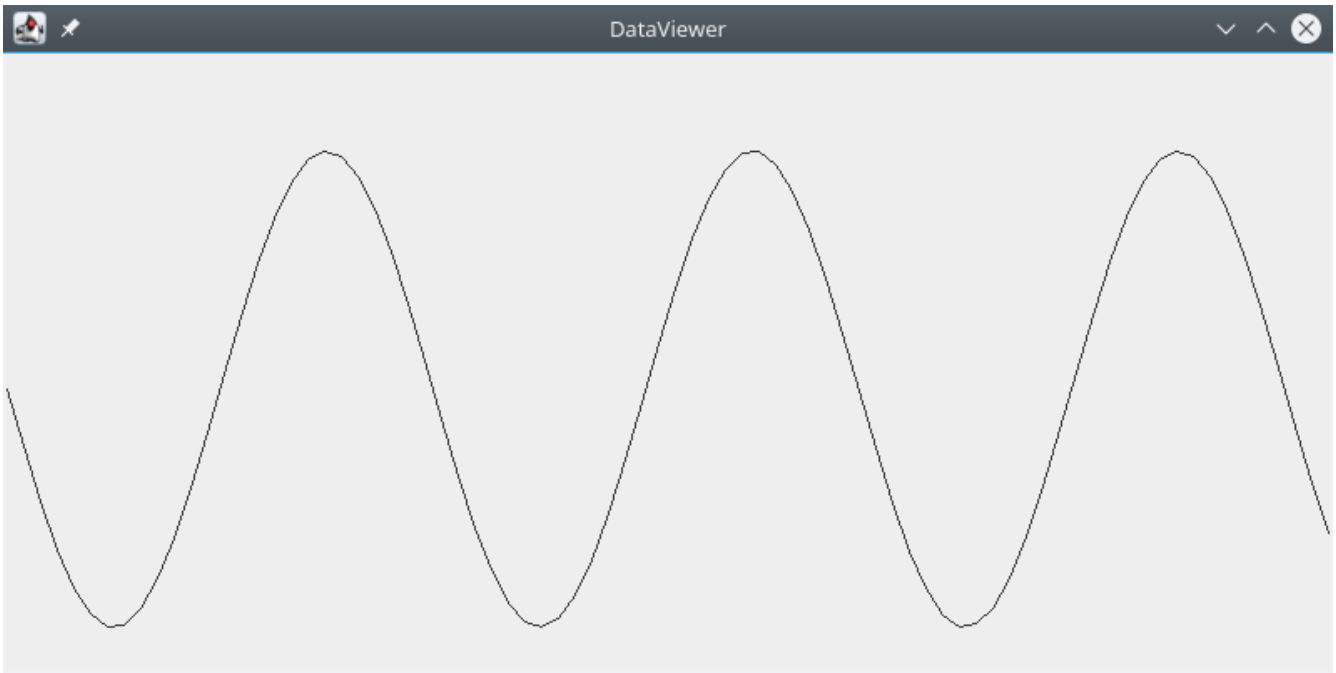


Abbildung 1: Sinuskurve wird simuliert in Java

Abbildungsverzeichnis