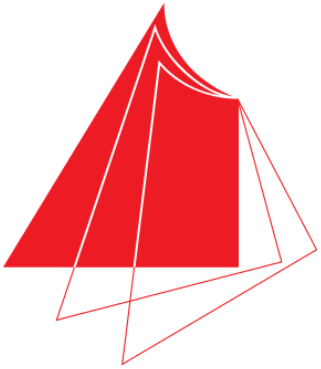

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Ausarbeitung Projektarbeit

Multithreadingfähiger Mergesort-Algorithmus in Scala mit Visualisierung über eine grafische Oberfläche

Tobias Kerst, 47646
Patrick König, 46910

Sommersemester 2016

Dozent: Prof. Dr. Heiko Körner

Inhaltsverzeichnis

1 Einleitung

1.1 Aufgabenbeschreibung

Die Aufgabe in unserer Projektarbeit besteht darin, den Mergesort-Algorithmus in der Programmiersprache Scala zu implementieren, sowie dessen Ablauf visuell darzustellen.

1.2 Motivation

Schon vor der Themenauswahl für unsere Projektarbeit waren wir von der Programmiersprache Scala begeistert. Um diese Programmiersprache zu erlernen, suchten wir nach einem Thema, welches sinnvollerweise in Scala implementiert werden könnte. Mit den Neuerungen von ScalaFX kam uns die Aufgabe zur Visualisierung eines parallelisierbaren Sortieralgorithmus sehr entgegen und verleitete uns schließlich zu unserer Entscheidung für dieses Thema.

Da der Mergesort ein sehr performanter und oft genutzter Sortieralgorithmus für eine große Menge an Elementen ist, fanden wir es interessant, etwas tiefer in die Materie einzutauchen und den Algorithmus in seinen Einzelheiten genau zu untersuchen, sowie diese visuell als Animationen darzustellen.
Desweiteren war es uns von großer Bedeutung, während der Projektarbeit an praktischer Programmiererfahrung dazuzugewinnen und auch theoretisch etwas Neues zu lernen. Deshalb haben wir uns auch dazu entschieden, die Aufgabe nicht in Java, sondern in einer anderen, modernen Programmiersprache zu implementieren, welche wir uns neu aneignen mussten. Darüber hinaus ist die Arbeitsweise mit `JavaFX und ScalaFX` sehr interessant, wenn es um das erstellen von Animationen und Transitionen geht. Zusätzlich war es uns wichtig, eine schönes User-Interface zu implementieren, um die dadurch erlangten Kenntnisse bei Gelegenheit in der Zukunft anwenden zu können.
Wir wissen, dass der Mergesort aufgrund seiner rekursiven Implementierung für viele nicht auf Anhieb zu verstehen ist. Deshalb besteht unser Ziel bei dieser Projektarbeit darin, den Benutzer beim Verstehen des Mergesort-Algorithmus so weit wie möglich zu unterstützen und ihm mit zahlreichen Features den Eintritt in die Welt der rekursiven Programmierung zu erleichtern. Zusätzlich zeigen wir auf, wie ein parallelisierbares Problem, in diesem Fall das Sortieren einer Liste, mit Hilfe von mehreren Threads effizienter gelöst werden kann.

1.3 Methodische Vorgehensweise

Da wir den Visual Mergesort in Scala implementiert haben, wird in dieser Dokumentation zuerst auf die Sprache an sich eingegangen. Im zweiten Kapitel werden also zunächst die grundlegenden Konzepte von Scala erläutert. Um das Einfinden in Scala zu erleichtern, werden wir zudem die Besonderheiten der Syntax genau erklären und gegebenenfalls einem vergleichbaren Beispiel der bekannten Programmiersprache Java gegenüberstellen. Im dritten Kapitel schaffen wir dem Leser in den Bereichen Rekursion und Mergesort-Algorithmus eine theoretische Grundlage, die benötigt wird, um den internen Systemablauf während der Animation nachvollziehen zu können. Im vierten Kapitel bekommt der Leser einen Einblick in den Aufbau unseres Programms. Darüber hinaus werden bestimmte Klassen und Codestücke, welche für die Implementierung eine zentrale Rolle spielen, ausführlich erklärt. Im fünften Kapitel befindet sich eine Bedienungsanleitung, welche sich ausschließlich mit der Bereitstellung und Benutzung der Applikation befasst. Zudem werden dort die nötigen Schritte erklärt, um das Programm selber zu kompilieren. Zuallerletzt werden wir unsere Vorgehensweise bei der Implementierung reflektieren und diese sowie unsere Ergebnisse einer kritischen Betrachtung unterziehen.

2 Scala

Wir haben uns entschieden, das Programm in Scala zu schreiben, einer Sprache, die bereits schon im Jahr 2001 von Martin Odersky[1] entwickelt wurde, jedoch erst seit Kurzem einen großen Bekanntheitsgrad erlangt hat. Grund dafür ist der Hype um die sogenannte *funktionale Programmierung* und das Bedürfnis, Anwendungen nebenläufig zu entwickeln.
Wir möchten nun zu Beginn die Gründe nennen, warum Scala sich als moderne Programmiersprache anbietet, die sogar von Javas Hauptentwickler James Gosling[2] als bevorzugte Java-Alternative betitelt wurde [3].

2.1 Laufzeit-Umgebung

Scala ist wie Java eine Programmiersprache, die zu Bytecode kompiliert wird. Dieser Bytecode wird dann von der *Java Virtual Machine* (`JVM`) benutzt, um daraus Maschinencode zu erstellen. Die JVM ist mittlerweile auf sehr vielen Rechnern installiert und sogar das Android Betriebssystem setzt auf eine Variante der JVM (`Dalvik`).

Die JVM hat den großen Vorteil, dass sie mittlerweile seit 20 Jahren aktiv entwickelt wird und die gesamte Java-Umgebung besonders im Enterprise–Bereich eingesetzt wird. Das Resultat ist ein sehr stabiles und vor allem perfomantes System, das aus dem heutigen IT-Markt nicht mehr wegzudenken ist.

2.2 Bibliotheken

Scala wird also nicht zu irgendeinem Bytecode kompiliert, sondern zu Java-Bytecode, um genau zu sein. Dies hat den großen Vorteil, dass man neben der JVM-Unterstützung auch auf etablierte Java Bibliotheken zugreifen kann. Beispielsweise ist der Einsatz von

[1] https://de.wikipedia.org/wiki/Martin_Odersky
[2] https://de.wikipedia.org/wiki/James_Gosling
[3] James Gosling wurde auf einer Konferenz gefragt, welche Programmiersprache er heutzutage anstelle von Java auf der JVM nutzen würde, worauf er entschieden mit “Scala” antwortete — http://www.adam-bien.com/roller/abien/entry/java_net_javaone_which_programming

Google GSON[4], das das Serialisieren von Objekten in JSON und zurück ermöglicht, über Scala so möglich, wie auch über Java.

So ist es dann auch kaum überraschend, dass der Einsatz von den üblichen Java Bibliotheken in Scala möglich ist. Ein Beispiel ist `JavaFX`, welches seit Java 8 Teil des *Java Development Kit* (`JDK`) ist.

2.3 JavaFX und ScalaFX

Um die Oberfläche (User Interface) zu gestalten, nutzt man heute Bibliotheken, die meistens so komplex sind, dass ihnen ganze Werke gewidmet werden. Wir haben uns für JavaFX entschieden, welches durch den Wrapper ScalaFX angesprochen wird. Was diese Bibliotheken machen und auszeichnet, wird in den folgenden Abschnitten erklärt.

2.3.1 JavaFX

JavaFX 8 ist der offzielle Nachfolger von Swing. Zu Beginn sollte es eine Scriptssprache werden, jedoch wurde dieser Fokus mit Version 2 aufgegeben und JavaFX wurde zu der GUI Bibliothek, wie man sie heute nutzt. In der aktuellen Version 8 (welcher der direkte Nachfolger von Version 2 ist und wegen der Einbindung in das JDK 8 diesen Versionssprung vollzogen hat), wurde die Bibliothek um wichtige Komponenten erweitert und bietet die folgendenden Vorteile:

Scene Graph

JavaFX ist besonders leicht zu entwickeln, da es auf den `Scene Graph` setzt. Der `Scene Graph` ist eine Baumstruktur, bei der die Elemente hierachisch angeordnet werden. Elemente im `Scene Graph` sind vom Type `Node` [VGC+14]. Wenn man also beispielsweise einen Button auf der Bildfläche platzieren möchte, dann hat man ein Button Objekt, welches von `Node` erbt.

Ein Vorteil ist, dass man so Objekte gruppieren kann und dadurch Operationen auf der Gruppe ausführen kann. So wird bei einer Verschiebe-Operation (auch *Translation* genannt) jedes Element in der Gruppe verschoben, was den Code lesbarer und wartbarer macht.
Außerdem ist somit eine zum Eltern Element relative Positionierung möglich.

Klares MVC

Bei JavaFX wird das Konzept für das Design in einer separaten Datei, einer **FXML** Datei geführt. So muss man das Layout nicht im Code generieren. Im Abschnitt zu FXML 2.3.3 wird dies ausführlicher beschrieben.

[4]Link zu der Projektseite von Google GSON: https://github.com/google/gson

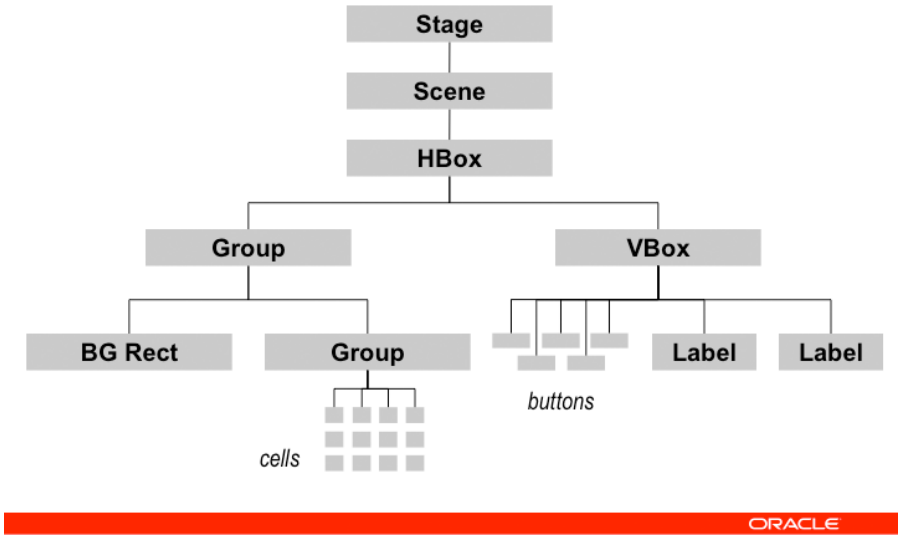

Abbildung 2.1: Darstellung der Baumstruktur im `Scene Graph`
http://www.smarks.org/svjugfx20100414/flood.html

Styling über CSS

Neben der Trennung von Ablauf, Logik und Gestaltung nach dem *MVC*-Prinzip, lässt sich das Styling durch eine CSS-ähnliche Syntax durchführen, was das Gestalten von Oberflächen durch Designer ermöglicht. Es wird jedoch eine leicht abgewandelte Syntax verwendet. Das Setzen der Textfarbe auf rot (`#ff0000` im RGB-Farbraum) wird durch die CSS-Syntax

```
.button: {
  -fx-color: #f00;
}
```

ermöglicht. Man erkennt, dass das Präfix `-fx-` vor dem CSS-Befehl steht.

Properties und Bindings

Das Koppeln von Eigenschaften einer Klasse an eine andere Klasse ist ein tolles Feature, welches das Entwickeln von komplexen Anwendungen noch weiter vereinfacht. Intern wird das `Observable`-Interface benutzt, die Komponenten aneinander zu binden.

Um Bindings und die dazu benutzten Properties besser zu verstehen, verweisen wir auf die sehr gute Zusammenfassung von [Cze].

2.3.2 ScalaFX

ScalaFX ist eine *Domain Specifig Language*[5] (DSL) für JavaFX. Als DSL bietet ScalaFX syntaktischen Zucker für die JavaFX–Bibliothek, und die folgenden Vorteile:

Lesbare Bind-Ausdrücke

Binding und Properties gehören zu den tollen Funktionen, die JavaFX bietet, jedoch ist die Syntax teilweise sehr umständlich.

Beispiel: Wenn man 3 Rechtecke hat (`rect1`, `rect2` und `reckt3`) und man möchte, dass das Rechteck `rect1` so hoch ist, wie `rect2` und `rect3`, dann bindet man die Höhe von `rect1` an die summierte Höhe von `rect2` und `rect3`.

Scala:

Listing 2.1: Scala Beispiel Code für natürliche Bindings

```
rect1.height <== rect2.height + rect3.height
```

Java:

Listing 2.2: Das selbe Beispiel in Java

```
rect1.heightProperty().bind(rect2.heightProperty().add(rect3
    .heightProperty()))
```

Der Scala Code ist wesentlich intuitiver und lesbarer, was genau der Sinn dieser DSL ist. Das Beispiel stammt aus [VGC+14] (Seite 574f.).

Angepasste Animations–Syntax

Da Animationen ein wichtiger Bestandteil von JavaFX sind, wurde die Syntax wesentlich verbessert. Auch hier möchten wir uns an [VGC+14] halten und Scala Code mit Java Code vergleichen.

Scala:

Listing 2.3: Scala Beispiel für eine einface Animation

```
Timeline(at (3 s) {radius -> 0}).play()
```

Java:

Listing 2.4: Das selbe Beispiel in Java

```
KeyValue collapse = new KeyValue(circle.radiusProperty(),0);
new Timeline(new KeyFrame(Duration.seconds(3), collapse))
  .play();
```

[5] Bei einer DSL handelt es sich um eine formale Sprache, die ein bestimmtes Problemfeld abdeckt. In dem Fall von ScalaFX handelt es sich um eine *UI DSL*, die als Wrapper um JavaFX gelegt wird.

Typsichere APIs

Ein Vorteil von Scala, wie wir noch kennen lernen werden, ist, dass die Sprache *statisch typisiert* ist. ScalaFX ist garantiert *typsicher*, was zur Folge hat, dass Fehler bereits beim Kompilieren auftreten und nicht erst zur Laufzeit zu Fehlern führen wird. [Pie02]

Interoperabilität zwischen ScalaFX und JavaFX

ScalaFX liegt wie ein Wrapper um JavaFX und bietet Funktionen an, die die Arbeit mit JavaFX erleichtern. Dies wird dadurch vereinfacht, dass man ganz einfach zwischen JavaFX und ScalaFX wechseln kann. Ein besonders hilfreiches Konzept sind hierbei *Implizite Konvertierungen*. So kann man beispielsweise eine `javafx.scene.Shape.button` an eine Methode übergeben, die ein `scalafx.scene.Shape.button` als Argument erwartet.

Da es sich bei ScalaFX primär um syntaktische Verbesserungen handelt, kann man argumentieren, dass die Benutzung obsolet ist. Wir haben uns bewusst für die Integration entschieden, da wir näher an der Scala Syntax bleiben wollten und die Lesbarkeit von Source-Code für uns ein wichtiger Aspekt ist[6].

2.3.3 FXML

Wenn man sich mit den Entwicklung von Swing auskennt, dann weiß man, dass man alle darstellbaren Objekte im Code selber generiert und gestaltet. Möchte man diese dann später ändern, so muss man sich mit dem Programmcode auseinander setzen und hier die Syntax nachvollziehen und Änderungen über die Programmiersprache vornehmen.

Dieser Ansatz ist in JavaFX auch möglich, jedoch hat dies den Nachteil, dass die Oberfläche durch den teilweise umständlichen Programmierstil beschrieben wird. Außerdem müssen sich Oberflächen-Gestalter in die Programmiersprache einarbeiten.
FXML ist eine auf `XML` basierende Sprache, dass die Struktur des Layouts beschreibt. Man kann das FXML Layout ganz einfach auslagern, also in einer separaten Datei speichern, sodass Anwendungs-Logik und die Präsentationsschicht getrennt sind, so wie im `Model-View-Control` Pattern gefordert.

Dadurch, dass das Layout über den Scene**graph** organisiert wird, bietet sich XML durch seine verschachtelten Tags an, diese Knotenstruktur abzubilden. Somit wird diese Struktur auch gleichzeitig **transparent**.
FXML wird nicht kompiliert, somit können Änderungen ohne vorheriges Rekompilieren direkt getestet werden.
Ein großer Vorteil ist, dass man einzelnen Knoten `ID`s und `Class`es zuteilen kann, wie in HTML. Diese lassen sich dann über CSS stylen. Hierbei wird eine leicht veränderte CSS-Syntax verwendet, aber jeder der sich in der Webgestaltung auskennt, wird sich

[6] Eine Interessante Diskussion zu diesem Thema findet man auch hier http://stackoverflow.com/a/22744881

hier schnell zurecht finden. Neben dem Stylen von Elementen ist die **Lokalisierung** des Inhalts heutzutage auch extrem wichtig, da der Softwaremarkt mittlerweile kaum mehr auf einen Sprachraum beschränkt ist. Hier bietet FXML eingebaute Features, die diese Lokalisierung sehr einfach ermöglichen.
Und da JavaFX das Laden und verarbeiten von FXML Dateien erlaubt, kann man auch mit anderen JVM–Sprachen, wie in unserem Fall Scala, FXML nutzen. Wir haben jedoch eine zusätzliche Library genutzt, die das Benutzen und Arbeiten mit FXML Dateien erleichtert: ScalaFXML [vig].

Zuletzt muss man in Verbindung mit FXML noch den Scene Builder[7] nennen. Dieses Tool erleichtert die Gestaltung von einfachen User Interfaces, indem Elemente per Drag & Drop platziert werden können und man somit eine Oberfläche gestalten kann, ohne sich genauer mit dem FXML Code auseinander setzen zu müssen. Das Programm hat uns am Anfang geholfen, die Syntax besser zu verstehen und die Elemente besser zu platzieren. Da wir in unserer Arbeit jedoch immer öfter Kleinigkeiten im Layout anpassen mussten, haben wir im Verlauf immer mehr auf den Scene Builder verzichtet, da die Anpassung einer ID, oder Größe oder das Einfügen eine Buttons händisch tatsächlich schnell geht und der Scene Builder nicht fehlerfrei funktionierte. Bei uns wurde die Menüzeile des Scene-Builders, die man beispielsweise nutzen kann, um von dort aus das Layout als native Anwendung ohne Logik zu testen, durch das Menü unserer Anwendung überschrieben [8].

2.4 Grundlegendes und Syntax

Die Programmiersprache Scala ist sehr komplex und die Sprache hier vorzustellen und zu erklären würde den Umfang dieser Ausarbeitung sprengen. Dennoch möchten wir einige Grundlegende Spracheigenschaften vorstellen, sodass man den später vorgestellten Code besser nachvollziehen kann.
Man kann zwar Scala Code schreiben, der sehr an Java Code erinnert, jedoch wird davon abgeraten. Einer der grundlegenden Ansätze[9] ist, dass man sich die elementaren Konzepte von Scala verinnerlicht und Code schreibt, der funktioniert. Man kann dann immer noch die komplexeren Eigenschaften der Sprache lernen.

Da dies das erste Programm ist, das wir in Scala geschrieben haben, kann man bestimmt viele Sachen besser machen, wenn man entsprechend Erfahrung hat. Dennoch haben wir probiert, viele der tollen Features zu nutzen, die Scala bietet.

[7] Der Scene Builder ist mittlerweile Open Source und kann hier herunter geladen werden http://gluonhq.com/labs/scene-builder/. Weitere Informationen findet man noch auf der mittlerweile archivierten Oracle Seite unter http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html

[8] Der Fehler wird hier beschrieben: https://bugs.openjdk.java.net/browse/JDK-8089659. Wir haben das Tag `useSystemMenuBar` genutzt, um auf OSX die native Menüzeile zu nutzen, was das Layout natürlicher macht.

[9] Siehe: http://www.scala-lang.org/documentation/getting-started.html

Viele der folgenden Beispiele kann man in der **REPL** (`Read-Evaluate-Print-Loop`) testen. Die REPL ist so etwas wie eine interaktive Console, in der man Scala Code testen und ausprobieren kann. Da wir diese für unsere Projektarbeit nicht direkt benötigten, aber eine tolle Möglichkeit darstellt um Scala Code zu testen, verweisen wir auf die tolle Beschreibung von [Ale]

2.4.1 Typinferenz

Scala ist eine statisch typisierte Sprache. Das bedeutet, dass eine Variable von einem Typ ist, der sich auch nicht ändert. In Java gibt man den Typ bei der Deklaration direkt mit an. Wenn man Beispielsweise einen Variable `helloWorld` vom Type `String` hat, die den Wert `Hello World` hat, dann wird dies wie folgt angegeben:

```
String helloWorld = "Hello World";
```

In Scala kann man den Typ weglassen, den die Variable hat, dieser wird vom Compiler beim Kompilieren ausgewertet. Es reicht also, wenn man

```
val helloWorld = "Hello World"
```

schreibt.
Wenn man möchte, so kann man den Typ dennoch mit angeben, das sieht dann wie folgt aus:

```
val helloWorld: String = "Hello World"
```

In den obigen Beispielen fällt auf, dass es ein Schlüsselwort gibt, das es in Java so nicht gibt: `val`. Außerdem muss man kein Semikolon an das Ende der Zeilen machen. Wie so oft in Scala gilt für das Semikolon: Muss man nicht, kann man aber.
Was es mit dem Schlüsselwort `val` auf sich hat, möchten wir im nächsten Abschnitt aufzeigen.

2.4.2 Values und Variablen

Das `val` steht für einen Value, also einen unveränderlichen Wert. Scala ist eine funktionale Programmiersprache. Einer der wichtigen Ansätze hierbei ist die *Immuntability*, also die *Unveränderlichkeit* von Werten. Man möchte bei nebenläufigen Operationen keine Werte haben, die von anderen Threads verändert werden können. Um dies zu vermeiden, benutzt man unveränderliche Werte, also **Values**. Dies entspricht einer `final`-Variable in Java.
Das Code-Fragment

```
val number = 1
number = 2
```

führt also zu dem Fehler:

```
error: reassignment to val
```

An dieser Stelle benötigt man also eine Variable `var`:

```
var number = 1
number = 2
```

2.4.3 Methoden

Methoden werden ähnlich wie in Java geschrieben. Sie haben einen Rückgabetype und eine ParameterListe. Wir wollen uns im folgenden eine sehr einfache Methode angucken:

```
1 def isLarge(value: Int): Boolean = {
2   if (value > 15)
3     true
4   else
5     false
6 }
```

Die Methode mit dem Namen `isLarge` nimmt also als Parameter einen `Integer`, den man mit dem Namen `value` anspricht. Die Methode hat Rückgabetyp `Boolean`.

Die Methode zeigt auch schon einige Besonderheiten an Scala. Man erkennt, dass das Schlüsselwort `return` nicht angegeben werden muss. In Java würde man `return true` und `return false` schreiben müssen [10] [11]. Dies hat den Hintergrund, dass **jeder Ausdruck ausgewertet wird** und somit etwas zurück gegeben wird. Wenn man nichts zurückgeben möchte, so gibt man `Unit` an. Unit gibt an, dass man mit dem Ergebnis, das zurückgegeben wird, nichts anfangen kann und kommt dem `void` in Java nahe. Beispielsweise die `println()` Methode hat den Rückgabetyp `Unit`.

Um das obige Beispiel noch klarer zu machen, speichern wir den von dem If-Statement zurückgegebenen Wert in einem Value, den wir dann zurück geben werden:

```
1 def isLarge2(value: Int): Boolean = {
2   val isValueLarger = (if (value > 15) true else false)
3
4   isValueLarger
5 }
```

Die beiden Methoden machen beide das Selbe. Wenn wir die Methode in der `REPL` testen, bekommen wir folgendes Ergebnis:

[10] Man kann if Statements wie einen ternären Operator nutzen, siehe: [Ale13] Section 3.6 — *Using the if Construct Like a Ternary Operator*

[11] Return Statements sollten vermieden werden, wie in [Sue12] Section 2.2.1 — *Don't use return* beschrieben

```
scala> def isLarge(value: Int): Boolean = {
     |   val isValueLarger = (if (value > 15) true else false)
     |
     |   isValueLarger
     | }
isLarge: (value: Int)Boolean

scala> isLarge(9)
res0: Boolean = false

scala> isLarge(20)
res1: Boolean = true
```

Wir werden die Methode `isLarge` im weiterhin benutzen, um die Funktionen von Scala hervorzuheben.

2.4.4 Strings

Das Arbeiten mit Strings in Scala ist sehr einfach und es gibt einige sehr nützliche Funktionen, die dafür sorgen, dass der Code nicht nur lesbarer, sondern auch leichter zu warten ist.

Das Konkatenieren von Strings und Variablen in Java ist sehr praktisch und einfach, man fügt diese einfach durch ein + Zeichen aneinander.

```
System.out.println("Is␣the␣number␣" + number + "␣a␣large␣number:␣
    " isLarge(number));
```

In Scala muss man sich keine Gedanken mehr um das Konkatenieren an sich machen, der gleiche Code wie oben sieht so aus:

```
println(s"Is␣the␣number␣$number␣a␣large␣number:␣${isLarge(number)
    }")
```

Das Weglassen von `System.out` ist kein Fehler, sondern ein Scala nicht nötig. Durch das Voransetzen von `s` an den String, kann man Variablen durch `$` ganz einfach einsetzen. Ausdrücke werden durch `${}` eingefügt, müssen also geklammert werden.

Mehrzeilige Strings mit Sonderzeichen

Ein weiteres tolles Feature ist, dass man mehrzeilige Strings nicht manuell umbrechen muss. Durch das Setzen von drei Anführungszeichen zu Beginn des Strings, kann man den String einfach umbrechen:

```
1 val multiLineString = """This
2 ␣␣␣␣␣|is
3 ␣␣␣␣␣|an
4 ␣␣␣␣␣|example""".stripMargin
5 println(multiLineString)
```

Dies ergibt die Ausgabe:

```
This
is
an
example
```

Eine tolle Beschreibung für diese Funktionalität gibt es in [Ale13] unter Section 1.2.

Eine weitere syntaktische Neuerung ist, dass die Methode `stripMargin()` ohne die Klammern aufgerufen wird. Dies ist in Scala möglich, wenn eine Methode keine Argumente erwartet.

Tatsächlich ist es möglich, dass man sogar den Punkt zwischen dem String (also dem Objekt) und der Methode weglässt. Dies bezeichnet man als *Postfix Operator Notation*, ein Feature, was Code teilweise sehr lesbar und schön macht, aber auch zu unerwartetem Verhalten führen kann.[12]

2.4.5 Getter & Setter

Die Nutzung von Getter– und Setter–Methoden Zugriff auf die Attribute eines Objekts ist gang und gäbe in Java. In Scala wird auf die Felder so zugegriffen, als wären diese public, obwohl dies nicht der Fall ist. Scala generiert beim Kompilieren die Methoden, die den Zugriff auf die Variablen erlauben. Ein Beispiel[13] soll dies veranschaulichen:

Listing 2.5: Beispiel Klasse Person

```
1 class Person() {
2   var name = ""
3   var age = 0
4 }
```

Da die Klasse keine Methoden hat, könnte man meinen, dass der Zugriff auf `name` und `age` nicht möglich ist. Jedoch wurden die Getter– und Setter–Methoden von Scala automatisch erstellt und man kann nun auf die Felder zugreifen:

```
1 person = new Person()
2
```

[12] Diese Stackoverflow Diskussion klärt einige dieser Fragen, setzt jedoch voraus, dass man sich mit der Scala Syntax auskennt: http://stackoverflow.com/questions/13011204/scalas-postfix-ops

[13] Das Beispiel stammt von http://dustinmartin.net/getters-and-setters-in-scala/

```
3 println(person.age)
4 println(person.name)
```

Man kann die Getter und Setter auch überschreiben, siehe hierzu [Ale13] Section 4.6 und die Quelle dieses Beispiels.

2.4.6 For–Schleifen

Schleifen sind ein elementares Konstrukt aus der Programmierung. Bei den For–Schleifen wird dabei meistens eine Zählvariable benutzt, die bis zu einem gewissen Wert zählt und auf die man währenddessen zugreifen kann. Ein klassisches Beispiel ist das Befüllen eines Arrays mit Werten.

Listing 2.6: Typische For–Schleife aus der Java Programmierung

```
1 for (int i = 0; i <= array.length; i++) {
2   array[i] = i * i;
3 }
```

In dem obigen Beispiel wird das Quadrat vom Index als Wert in dem Array gespeichert. Dabei wird jedoch im Schleifen–Kopf manuell geprüft, ob das Ende des Arrays erreicht ist Um typische Fehler zu vermeiden (beispielsweise `ArrayIndexOutOfBoundsException`), wurden Iteratoren eingeführt und man durchläuft damit Listen etc. durch For-Each Schleifen. Wenn man jedoch einen Index-Zugriff ermöglichen will, so benötigt man wieder eine Zählvariable.

```
1 int i = 0;
2 for (int element : array) {
3   System.out.println(i + ": " + element); // Gibt Index und
       enthaltenes Element aus
4   i++;
5 }
```

Da die Zählvariable `i` somit *variabel* sein müsste, gibt es in Scala einen anderen Ansatz. Man kann durch ein Schleifenkonstrukt wie folgt, die obige Ausgabe erreichen:

```
1 val array = new Array(1,2,3,4,5,6)
2 for (i <- 0 until array.length) {
3   println(s"$i: ${array(i)"})
4 }
```

Wie man erkennt, greift man nicht mehr über `array[i]` auf ein Element in dem Array zu, sondern über `array(i)`. Ansonsten zählt man die i Variable hoch. Man kann auch eine For-Each Schleife kreiren, so wird aus dem Beispiel oben:

Listing 2.7: For–Each in Scala

```
1 var i = 0
```

```
2  for (element <- array) {
3    println(s"$i: $element)
4  ␣␣i␣+=␣1
5  }
```

Die Syntax sollte nun nicht mehr überraschen, man nimmt jedes Element aus dem Array und gibt die Anzahl aus. Eine schönere Methode, die wir auch in unserer Projektarbeit öfter benutzt haben, ist sich die Zählvariable generieren zu lassen. Hierzu nutzt man die `zipWithIndex` Methode.

```
1  for ((element, index) <- array.zipWithIndex) {
2    println(s"$index: $element")
3  }
```

Nun haben wir einige Schleifen-Konstrukte kennen gelernt, jedoch nutzt man in Scala üblicherweise die `foreach`–Methode, die direkt in der Collection angeboten wird. Hier macht sich bemerkbar, dass Scala eine funktionale Programmiersprache ist. Man übergibt an die `foreach`–Methode nämlich eine Funktion, die jedes Mal ausgeführt wird. Am besten betrachtet man ein Beispiel, um mit der Syntax vertraut zu werden.

```
1  val array = new Array(1,2,3,4,5,6)
2  array.foreach(e: Int => println(e))
```

Hier wird die an die For-Each Methode die Funktion

```
e: Int => println(e)
```

übergeben. Da in dem Array Elemente vom Typ enthalten sind, nimmt die Funktion Werte `e` von Typ `Int`. Für diese Werte wird dann die Funktion `println(e)` aufgerufen. Man kann diesen Ausdruck auch in einem Value speichern und so an die `forearch`–Methode übergeben.

```
val printElements = {e: Int => println(e)}
```

Nun gibt es in Scala noch einige Möglichkeiten, den Code weiter zu minimieren und somit lesbarer zu machen. So kann man die Typangabe von `e: Int` weglassen und lediglich `e` schreiben, da die `foreach`–Methode auf einer Liste von `Int`s aufgerufen wird.

```
array.foreach(e => println(e))
```

Und da klar ist, auf welches Element zugegriffen wird, kann man den Variablennamen ganz weglassen und einfach auf das Element zugreifen mit `_`. Das sieht das so aus:

```
array.foreach(println(_))
```

Benötigt man den Index, dann kann man auch hier wieder die `zipWithIndex` Methode benutzen:

```
array.zipWithIndex.foreach(e => println(s"${e._2}: ${e._1}"))
```

Was es mit dem `e._2` und `e._` auf sich hat, sehen wir im kommenden Abschnitt.

2.4.7 Tupel

Tupel sind ein in Scala eingebautes Feature, welches bei Java seit jeher vermisst wird. In einem Tupel werden zwei, oder mehr Elemente abgespeichert, auf die man im Anschluss zugreifen möchte. Der Urpsrung liegt darin, dass man in Methoden nur ein Objekt zurück geben kann, welches man im Methodenkopf auch angeben muss. Möchte man mehr als ein Elemente zurück geben, so muss man ein neues Objekt entwerfen, das die beiden Ergebnisse umschließt, ein sogenanntes `Wrapper`–Objekt. Das es solche Objekte, beispielsweise als `Pair`, `Tripel` oder anderweitig nicht in der Java–Standard–Bibliothek gibt, wird häufig bemängelt[14], auch wenn es Bibliotheken[15] gibt, die diese Funktionalität nachrüsten.

In Scala gibt es Tupel, die bis zu 22 Elemente beinhalten können. [16]

Beispielsweise kann man wie folgt ein Tupel zurückgeben:

```
def returnTuples(): (Int, String) = {
  (12, "Hello World")
}
```

Man kann nun auf die 12 zugreifen, indem man sich das erste Element aus dem Tupel über `_1` holt.

Wichtig! Das erste Element in einem Tupel hat den Index 1.

Listing 2.8: Zugriff auf die Elemente eines Tupels

```
val exampleTuple = returnTuples
println(exampleTuple._1) // Gibt 12 zurück
println(exampleTuple._2) // Gibt Hello World zurück.
```

2.4.8 Casting

Auch in Scala war es teilweise notwendig, dass wir Objekte von einem Typ in den anderen `casten` mussten. Dies war beispielsweise dann notwendig, wenn wir Polyphormie nutzten, aber dann eine bestimmte Methode auf dem abgeleiteten Objekt aufrufen wollten.

[14] https://dzone.com/articles/whats-wrong-java-8-part-v
[15] http://www.javatuples.org
[16] Auch wir haben uns die Frage gestellt, wieso gerade 22 Elemente die Obergrenze darstellen: http://stackoverflow.com/questions/4152223/why-are-scala-functions-limited-to-22-parameters

Beispiel Unsere Klasse `SortElement` erbt von `Node`. Wenn man auf einem `Group`–Objekt die Methode `getChildren` aufruft, so bekommt man eine Liste von `Node`s zurück geliefert, auch wenn in der Liste eigentlich `SortElement`s enthalten sind. Wenn man sich sicher ist, so kann man die Objekte casten, und zwar über den Aufruf der Methode `asInstanceOf[SortElement]`. Wir möchten hier wieder exemplarisch Java–Code mit Scala–Code vergleichen:

Listing 2.9: Casting in Java: Von Node zu SortElement

```
SortElement exSortElement = (SortElement) group.getChildren().get
    (0)
```

In Scala würde der selbe Code wie folgt aussehen:

Listing 2.10: Casting in Scala: Von Node zu SortElement

```
val exSortElement: SortElement = group.getChildren(0).
    asInstanceOf[SortElement]
```

Die Angabe des Typs von `exSortElement` kann man sich aufgrund der *Typinferenz* sparen, ist hier jedoch angegeben, um explizit zu sagen, welchen Typ das Objekt haben wird (`SortElement`).

2.4.9 Pattern Matching

Pattern Matching kann man als aufgebohrte Switch Statements verstehen, die extrem vielfältig einsetzbar und dennoch gut lesbar sind. Als in Java 7 eingeführt wurde, dass man `switch`-Statements auf Strings anwenden konnte, war dies ein echter Meilenstein, da man vorher mühselig `if-else-if` Konstrukte bauen musste. Pattern Matching ist die Evolution dieser Switch Statements.

Ein einfaches Beispiel ist die Überprüfung, welche Funktion ausgeführt werden soll, nachdem ein Programm mit gewissen Parametern gestartet wurde:

Listing 2.11: Pattern Matching mit Strings

```
1 def parseArgument(arg: String): Unit = arg match {
2   case "-h" | "--help" => displayHelp
3   case "-v" | "--version" => displayVersion
4   case unknown => throw new IllegalArgumentException(unknkown)
5 }
```

Ein tolles Beispiel für den Nutzen ist zu checken, von welchem Typ ein Objekt ist:

Listing 2.12: Type-Checking mit Pattern Matching

```
1 def getClassAsString(x: Any): String = x match {
2   case s: String => s + "␣is␣a␣String"
3   case i: Int => "Int"
```

```
4     case l: List[_] => "List"
5     case p: Person => "Person"
6     case _ => "Unknown"
```

Das Beispiel und eine Diskussion, wie man mit dem *default* Fall umgehen sollte kann man in [Ale13] (Section 3.7 — *Using a Match Expression Like a switch Statement* & Section 3.8 — *Matching Multiple Conditions with One Case Statement*) nachlesen.

Einige weitere Beispiel findet man auf der Seite https://kerflyn.wordpress.com/2011/02/14/playing-with-scalas-pattern-matching/.

2.4.10 Exception Handling & Optional

3 Theoretische Grundlgen

3.1 Rekursion

Um den Mergesort-Algorithmus in seiner eleganten Form zu verstehen, ist es unausweichbar, sich zuerst mit dem Konzept der Rekursion zu befassen. Generell bedeutet Rekursion, dass ein Regelwerk erneut auf die Dinge angewendet werden kann, welche mit diesem Regelwerk erzeugt wurden, was potenziell zu einer endlosen Produktionsschleife, wie beispielsweise bei einer Rückköpplung führen kann. In der Informatik dient Rekursion dazu, sonst sehr komplexe Sachverhalte elegant zu formulieren. Vereinfachterweise ist eine rekursive Methode eine Methode, die sich selbst aufruft und somit der Gegenspieler zu einer iterativen Methode. Hierbei spielt die korrekt formulierte Abbruchbedingung der Methode eine zentrale Rolle, da man sonst Gefahr läuft, eine Endlosschleife zu produzieren. Prinzipiell lässt sich sagen, dass iterative und rekursive Programmierung gleich mächtig sind, da sich jedes rekursiv lösbare Problem unter mehr oder weniger Umständen auch iterativ ausformulieren lässt und umgekehrt.

Ein Beispiel hierfür wäre die einfache Berechnung der Fakultät einer Zahl in Java. Die erste Lösung erfolgt rekursiv:

```
1    public static berechne_fakultaet_rekursiv(int n){
2
3      if(n <= 1){
4      return 1;
5      }
6      else{
7      return ( n * berechne_fakultaet_rekursiv(n-1));
8      }
9
10   }
```

Der rekursive Methodenaufruf befindet sich direkt hinter dem zweiten `return` Ausdruck. Hier wird die gleich Methode immer wieder mit einer um eins dekrementierten Zahl aufgerufen, bis die vordefinierte Abbruchbedingung $n <= 1$ eintritt.

Alternativ lässt sich die gleiche Funktion auch iterativ implementieren, wobei die Methode nur ein einziges Mal aufgerufen wird, und das Problem linear löst:

```
1    public static berechne_fakultaet_iterariv(int n){
2
```

```
3    int fakultaet = 1;
4    int faktor = 2;
5    while (faktor <= n){
6
7        fakultaet = fakultaet * faktor;
8        faktor ++;
9
10   }
11   return fakultaet;
12
13   }
```

Wie man sieht wurde der rekursive Methodenaufruf in diesem Fall durch eine while-Schleife ersetzt.

3.2 Merge–Sort

Der erstmals 1945 durch John von Neumann vorgestellte Mergesort ist ein Sortieralgotithmus, der nach dem Paradigma *divide–and–conquer* arbeitet. Bei diesem Prinzip wird das eigentliche, große Problem so lange rekursiv in kleinere Probleme unterteilt, bis diese lösbar sind. Im Anschluss wird aus allen Teillösungen die Endlösung rekonstruiert. Die genaue Funktionsweise des Mergesort erfolgt in zwei Schritten: Im ersten Schritt wird die ursprüngliche Liste in zwei Hälften zerlegt, die jeweils wieder in einer Liste gespeichert werden. Dieser Schritt wird, zusammen mit dem nachfolgenden Schritt, so lange rekursiv fortgesetzt, bis sich nur noch ein Element in jeder Liste befindet. Im zweiten Schritt werden die Hälften sortiert und zu einer Menge zusammengefügt, bis sich irgendwann wieder die Gesamtmenge mit allen enthaltenen Ursprungselementen ergibt. Hierbei werden immer die ersten Elemente der beiden Hälften verglichen, wobei das jeweils kleinere in die zusammengefügte Menge wandert.

3.2.1 Laufzeit

Da sich die Größe der Liste bei jedem Merge verdoppelt, werden $log(n)$ (n = Anzahl der zu sortierenden Elemente) Mergeschritte benötigt, um das Ergebnis vollständig zusammenzusetzen. Bei beispielsweise 8 Elementen ergeben sich insgesamt $log_2(8) = 3$ Schritte.

Jeder Mergeschritt benötigt wiederum n Schritte, um die Elemente der beiden Listen zu sortieren, da hierzu jedes einzelne Elemente betrachtet und eingeordnet werden muss

Die gesamte Laufzeit beträgt also $O(nlog(n))$. Im Vergleich zu anderen Sortieralgorithmen wie Bubblesort (Worst-Case-Laufzeit: $O(n^2)$) und Quicksort (Worst-Case-Laufzeit: $O(n^2)$) ist der Mergesort bei größeren Datenmengen sehr effizient, da dessen Laufzeiten im Best-Case- und im Worst-Case-Szenario kaum Unterschiede aufweisen.

3.2.2 Allgemeine Implementierung in Scala

Sortieren

```
1  def sort(list: List[SortElement]) {
2      if (list.size > 1 ){
3
4        val firstListLength = (list.size / 2.0).ceil.toInt
5        val splitList = list.splitAt(firstListLength)
6        val left = splitList._1
7        val right = splitList._2
8        sort(left)
9        sort(right)
10       merge(list, left, right)
11     }
12   }
```

Die Methode `sort` nimmt eine Liste von Elementen entgegen und teilt diese in der Mitte in zwei weitere Listen auf. Durch `list.splitAt` wird ein Tupel erzeugt, welches in der Value splitList gespeichert wird. Mit Hilfe eines Unterstrichs kann entweder auf das erste Feld oder auf das zweite Feld des Tupels zugegriffen werden. Nachdem die ursprüngliche Liste gesplittet wurde, wird auf dem linken Teil rekursiv die Methode `sort` aufgerufen. Dies sorgt zunächst dafür, dass die linke Liste so lange in zwei Teile aufgeteilt wird, bis sich nur noch ein einziges Element in dieser Liste befindet und die Abbruchbedingung der Rekursion greift. Zu diesem Zeitpunkt befindet sich im rechten Teil der Liste auch nur ein Element und es kommt zur ersten Zusammensetzung zweier sortierter Listen. Diese Methodik wird nun rekursiv für alle Teile der Liste durchgeführt, bis die zwei zu Anfang aufgeteilten Listen sortiert sind und es zum letzten Merge kommt.

Mergen

```
1  def merge(resultList:List[SortElement], leftList:List[SortElement
       ], rightList:List[SortElement]) {
2
3    val leftSize: Int = leftList.size
4    val rightSize: Int = rightList.size
5    val totalSize: Int  = leftSize + rightSize
6    var i = 0
7    var j = 0
8
9    for (k <- 0 until totalSize) {
10
11     if(i < leftSize && j < rightSize){
12
13       if (leftList(i) < rightList(j)){
14         resultList = resultList.updated(k, leftList(i))
```

```
15          i = i + 1
16        } else {
17          resultList = resultList.updated(k, rightList(j))
18          j = j + 1
19        }
20      } else if (i >= leftSize && j < rightSize){
21        resultList = resultList.updated(k, rightList(j))
22        j = j + 1
23      } else {
24        resultList = resultList.updated(k, leftList(i))
25        i = i + 1
26      }
27    }
28  }
```

Diese Methode nimmt zwei vorsortierte Listen an und setzt alle Elemente dieser Teillisten zu einer einzigen, sortierten Liste zusammen. Hierzu wird für jeden Index k der Ergebnisliste bestimmt, welches Element aus den beiden Listen an dieser Stelle einsortiert wird. Vorausgesetzt, die Ergebnisliste enthält noch nicht alle Elemente mindestens einer Teilliste, werden bei jedem Schritt die jeweils kleinsten Elemente der beiden Teillisten miteinander verglichen, die sich noch nicht in der Ergebnisliste befinden. Das kleinere wird an den aktuellen Index k der Ergebnisliste gesetzt und der Zeiger i bzw. j der Teilliste wird inkrementiert, um auf das nächstgrößere Element zu zeigen. An dieser Stelle macht man es sich zunutze, dass die beiden Teillisten vorsortiert sind, da das nächstgrößere Element mit dem Hochzählen des Indexes i bzw. j der Teillisten automatisch ausgewählt wird. In diesem Codebeispiel existiert also eine einzige Liste, die fortlaufend aktualisiert wird, bis sie das Ergebnis beinhaltet. Zur Visualisierung des Algorithmus in unserer Applikation werden wir später für jedes Zwischenergebnis, welches gesplittet oder zusammengesetzt wurde, eine eigene Gruppe anlegen.

4 Eigenleistung

4.1 Aufbau der GUI

Um die benötigten Zeichen- Kontroll- und Schaltflächen zu unserem Interface hinzuzufügen, haben wir uns für die Sprache FXML entschieden. Dabei hat unsere Applikation folgenden Aufbau: Auf der untersten Ebene befinden sich eine `MenuBar`, welche das Menü in der oberen Leiste einer Standardandwendung darstellt und eine `BorderPane`, die es erlaubt, das eigentliche Fenster zur Visualisierung des Mergesort in verschiedene Bereiche zu unterteilen. Im oberen Bereich der `BorderPane` befindet sich eine `AnchorPane`, die dazu dient, das Anordnen der Schalt- und Kontrollflächen zu erleichtern, da diese dort relativ zu den Rändern und anderen Elementen ausgerichtet werden können.

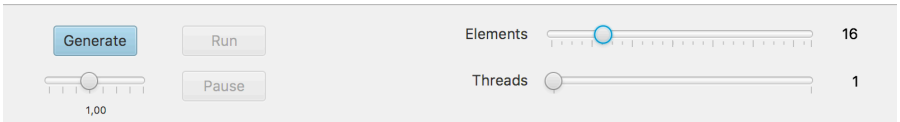

Abbildung 4.1: Anchorpane mit Schaltflächen

Im Zentrum der `BorderPane` befindet sich eine `ScrollPane`, die alle weiteren Elemente beinhaltet, die zur Darstellung der eigentlichen Zeichenfläche dienen. In diesem Teil finden ausschließlich die Animationen statt - somit sind keine Interaktionen mit der Benutzeroberfläche möglich.

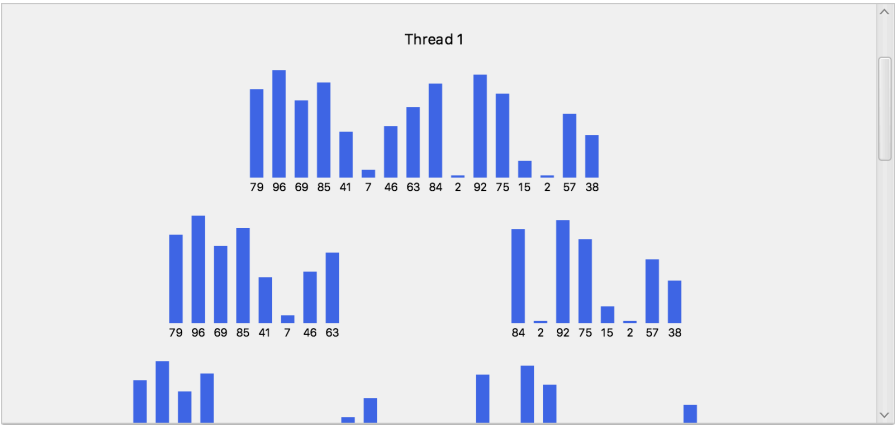

Abbildung 4.2: Scrollpane mit generierten Elementen

Zuletzt befindet sich im unteren Teil der `BorderPane` eine `TextArea`, die für alle textuellen Ausgaben in der Konsole verantwortlich ist. Hier kann der Benutzer optionalerweise das Geschehen, welches im Zentrum animiert wird, auf leserische Art mitverfolgen.

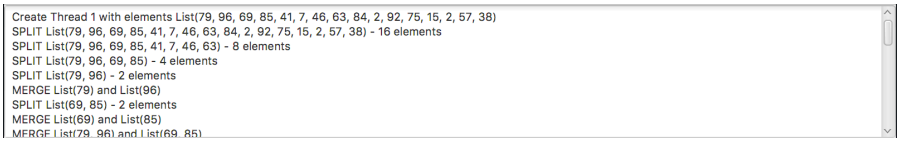

Abbildung 4.3: Textarea als Konsole

4.2 Das Objekt VisualMergesort

Das Objekt VisualMergesort ist ein Singleton-Objekt der anonymen Klasse VisualMergesort, welches garantiert einzigartig ist. Das bedeutet, dass keine weiteren Objekte gleicher Art erzeugt werden können. „VisualMergesort“ erbt von JFXApp und definiert somit den Startpunkt unserer Applikation.

```
1  import java.io.IOException
2
3  import scalafx.application.JFXApp
4  import scalafx.application.JFXApp.PrimaryStage
5  import scalafx.scene.Scene
6  import scalafx.Includes._
7  import scalafxml.core.{FXMLView, NoDependencyResolver}
8  object VisualMergesort extends JFXApp {
9
10   private val layoutFile: String = "/VisualMergesort.fxml"
11   val resource = getClass.getResource(layoutFile)
12
13   if (resource == null) {
14     throw new IOException(s"Cannot␣load␣resource:␣$layoutFile")
15   }
16
17   val root = FXMLView(resource, NoDependencyResolver)
18
19   stage = new PrimaryStage() {
20     title = "Visual␣Mergesort"
21     scene = new Scene(root)
22   }
23
24 }
```

In diesem Objekt wird das in FXML vordefinierte Layout unserer GUI in die Stage der Applikation geladen.

4.3 Die Klasse SortElement

Die Klasse SortElement ist für die Objekte verantwortlich, die durch den Algorithmus auf der Zeichenfläche sortiert werden sollen. Die zu sortierenden Objekte sind zusammengesetzte Gruppen und bestehen aus einem Rechteck sowie einem darunter liegenden Text, welcher den Wert des Objektes angibt. Die Höhe des Rechtecks verhält sich proportional zu diesem Wert. Der Text benötigt, um mittig unter dem Rechteck zu stehen, noch einen offset, falls der Wert einstellig ist.

Über das `require` werden Voraussetzungen für die Klasse angegeben, die zu Exceptions führen, wenn diese nicht korrekt sind.

Listing 4.1: Kopf der SortElement Klasse

```
1  class SortElement(val number: Int, var _xPos: Double, var _yPos:
       Double) extends Group with Ordered[SortElement]  {
2    require(number >= 1 && number <= 99 , "the␣number␣must␣be␣
       between␣1␣and␣99␣(inclusive)")
3
4    var text = new Text(number.toString)
5    text.style = "-fx-font-size:␣10px;␣-fx-background:␣#f00"
6
7    val offset = if (number < 10) { SortElement.smallNumberOffset }
       else { 0 }
8    text.translateX() = xPos + offset
9    text.translateY() = _yPos + number + SortElement.width
10
11   var rectangle = new Rectangle(new javafx.scene.shape.Rectangle(
       _xPos, _yPos, SortElement.width, number))
12   rectangle.setFill(Color.DARKBLUE)
13   this.getChildren.addAll(rectangle,text)
```

Um die von Scala vordefinierten Getter und Setter der Variablen `xPos` und `yPos` zu überschreiben, werden diese in `_xPos` und `_yPos` umbenannt. Danach werden zunächst die Getter definiert:

Listing 4.2: Definiert die Setter Methoden

```
1  def xPos = _xPos
2  def yPos = _yPos
```

Dieser Code definiert zwei simple Methoden `xPos` und `yPos`, welche die Variablen `_xPos` und `_yPos` zurückgeben. Da bei Scala der letzte Ausdruck einer Methode auch gleichzeitig der Rückgabewert ist und geschweifte Klammern nicht benötigt werden, falls die Methode nur aus einem Ausdruck besteht, sind diese und das Return-Statement nicht vorhanden.

Als nächstes werden die Setter neu definiert:

Listing 4.3: Setter Methoden

```
1  def xPos_= (x: Double) {
2    _xPos = x
3    text.translateX = _xPos + offset
4    rectangle.x = x
5  }
6
7  def yPos_= (y: Double) {
8    _yPos = y
9   text.translateY = _yPos + number + SortElement.width
10   rectangle.y = y
11 }
```

Der Name dieser Methoden ist `xPos_` und `yPos_`. Der Unterstrich dient in Scala als spezielles Zeichen und ist in diesem Fall als Platzhalter zu verstehen, der beim Aufrufen der Methoden durch ein Leerzeichen ersetzt werden kann. Somit können diese Methoden im folgenden Code sowohl mit `xPos =` bzw. `yPos =` als auch mit `xPos_=` bzw. `yPos_=` aufgerufen werden. Siehe 2.4.5

Um die Objekte beim Sortieren durch den Mergesort miteinander vergleichen zu können, werden die Methoden des Trait `Ordered` implementiert. Dieser Trait entspricht dem Interface `Comparable` in Java. Zusätzlich zu der Methode `compare` werden verschiedene Vergleichsoperatoren für das Objekt SortElement neu definiert:

Listing 4.4: Überschriebene Methoden

```
1  override def compare(that: SortElement): Int = {
2    this.number - that.number
3  }
4
5  override def <(that: SortElement): Boolean = {
6    this.number < that.number
7  }
8
9  override def <=(that: SortElement): Boolean = {
10   this.number <= that.number
11 }
12
13 override def >(that: SortElement): Boolean = {
14   this.number > that.number
15 }
16
17 override def >=(that: SortElement): Boolean = {
18   this.number > that.number
19 }
20
21 override def toString(): String = {
```

```
22    this.number.toString
23  }
```

Zusätzlich zu der Klasse SortElement wird noch ein `Singleton-Object SortElement` angelegt, welches alle vordefinierten Werte beinhaltet, die von der Klasse SortElement benötigt werden. Die Klasse SortElement kann auf alle Felder des Objektes zugreifen und diese für Erstellung neuer Objekte benutzen.

Listing 4.5: Das Singleton–Object SortElement

```
1  object SortElement{
2    // To place small numbers (numbers < 10) centered under the
       rectangle, this offset is needed.
3    val smallNumberOffset = 3
4    // This is the rectangles width
5    val width = 12
6    // This is the space between two rectangles
7    val offsetToNextElement = 8
8
9    val wholeElementWidth = width + offsetToNextElement
10
11   val maxHeight = 99
12   val minHeight = 1
13 }
```

4.4 Die Klasse MainController

Die Klasse MainController ist, wie der Name schon hergibt, der Controller unserer Applikation. Hier werden alle Interaktionen des Benutzers mit der GUI entgegengenommen und verarbeitet. Um dies zu implementieren, werden alle im FXML-Layout vordefinierten Buttons, Slider und andere Eingabemöglichkeiten in diese Klasse importiert und deren Methoden definiert. Hierzu dient die über der Klasse stehende Annotation `@sfxml` . Die Objekte lassen sich über eine im FXML-Code festgelegte ID vom Controller ansprechen. Diese ID's müssen für die Klasse übernommen und am Anfang festgelegt werden:

Listing 4.6: Erzeugen der Elemente und Initialisierung der Anwendung durch den Benutzer

```
1  @sfxml
2  class MainController(
3                        private val generateButton: Button,
4                        private val runButton: Button,
5                        private val amountOfElementsSlider: Slider,
6                        private val amountOfThreadsSlider: Slider,
7                        private val amountOfElementsLabel: Text,
8                        private val amountOfThreadsLabel: Text,
```

```
 9                          private val playPauseMenu: MenuItem,
10                          private val playPauseButton: Button,
11                          private val playbackSpeed: Slider,
12                          private val playbackSpeedLabel: Label,
13                          private val consoleLog: TextArea,
14                          private val borderPane: BorderPane,
15                          private val scrollPane: ScrollPane,
16                          private val pane: Pane,
17                          private val actionBar: AnchorPane) {
```

Um den Visual-Mergesort betriebsbereit zu machen, hat der Benutzer die Möglichkeit, aus drei verschiedenen Möglichkeiten auszuwählen, um die zu sortierenden Elemente auf der Zeichenfläche zu platzieren. Im Menü lässt sich über die vorhandenen Schaltflächen auswählen, ob die Elemente vom System zufällig verteilt, vorsortiert oder invertiert erzeugt werden sollen. Hierzu dient ein Enum, das je nach betätigtem Button mit dem entsprechenden Wert an die Methode `generateNumbers` übergeben wird.

```
1  def generateRandomNumbers(): Unit = {
2    generateNumbers(ElementOrder.Random)
3    changeButtonActivationToRun()
4  }
5
6  def generateOrderedNumbers(): Unit = {
7    generateNumbers(ElementOrder.Ordered)
8    changeButtonActivationToRun()
9  }
10
11 def generateInverseNumbers(): Unit = {
12   generateNumbers(ElementOrder.Inverse)
13   changeButtonActivationToRun()
14 }
```

Nachdem die Elemente erzeugt wurden, ist die Anwendung betriebsbereit und der Run-Button wird aktiviert.

In der Methode `generateNumbers` wird eine Liste mit Zufallszahlen zwischen dem Minimalwert `defaultMinimumNumber` und dem Maximalwert `defaultMaximumNumber` generiert, die so viele Elemente enthält, wie über den Slider in der Applikation eingestellt wurde. Der Unterstrich dient in diesem Fall als Platzhalter für jedes Element in der Liste. Je nachdem, welcher Button zuvor betätigt wurde, wird die Liste anschließend optional noch sortiert oder invertiert.

Listing 4.7: Erstellung der Zahlen-Elemente

```
1  def generateNumbers(elementOrder: EnumVal): Unit = {
2
3    // Get the selected amount of elements
```

```
4     val amountOfElements: Integer = amountOfElementsLabel.text().
          toInt
5
6     val randomNumberList = List.tabulate(amountOfElements)(_ =>
          ThreadLocalRandom.current.nextInt(defaultMinimumNumber,
          defaultMaximumNumber + 1))
7     val elements = elementOrder match {
8       case ElementOrder.Random => randomNumberList
9       case ElementOrder.Ordered => randomNumberList.sorted
10      case ElementOrder.Inverse => randomNumberList.sorted.reverse
11      case _ => throw new IllegalArgumentException(s"$elementOrder␣
            is␣not␣supported")
12    }
13
14    placeElementsOnPane(elements)
15  }
```

Zum Schluss wird die fertige Liste and die Methode `placeElementsOnPane` übergeben.

Die Methode `placeElementsOnPane` erzeugt für jeden Wert in der übergebenen Liste ein `SortElement` und fasst alle generierten Objekte in einer Gruppe zusammen. Ist die Applikation bei Aufruf der Methode am Laufen, wird sie gestoppt.

```
1     def placeElementsOnPane(elements: List[Int]): Unit = {
2
3       cleanEverythingUp
4       // Stop the running transition, and then place the new
            elements on the pane
5       if (transition != null) transition.stop()
6       // Setting up the canvas
7       val elementGroup = new Group()
8       elementGroup.id() = "level-1"
9       // Place the elements on the pane
10      for ((value, position) <- elements.zipWithIndex) {
11        val xPos: Double = (position * SortElement.
              wholeElementWidth).toDouble
12        val yPos: Double = (SortElement.maxHeight - value).toDouble
13        val sortElement = new SortElement(value, xPos, yPos)
14        sortElement.id() = s"sortElement-$position"
15        elementGroup.children.add(sortElement)
16      }
17
18      elementGroup.translateX <== scrollPane.getScene.getWindow.
            width/2 - elementGroup.getBoundsInParent.getWidth/2
```

Anschließend wird die Zeichenfläche von allen darin befindlichen Objekten gesäubert und die neu generierte Gruppe wird hinzugefügt:

Basierend auf der Anzahl der neu erzeugten Elemente, wird die Größe der Zeichenfläche angepasst. Dazu wird vorerst die maximale Tiefe der Gruppen definiert. Bei jedem Split und bei jedem Merge, wandern die gesplitteten oder zusammengesetzten Gruppen eine Ebene nach unten. Da sich eine Gruppe bei jedem Split so lange in zwei Teile teilt, bis sich nur noch ein einziges Element in der Gruppe befindet und genauso viele Merges wie Splits durchgeführt werden, lautet die Formel für die Anzahl an Splits und Merges:

$$log_2 n \times 2$$

Allerdings befindet sich von Anfang an schon die vom System generierte Gruppe auf der Zeichenfläche. Diese muss noch hinzugerechnet werden. Somit ergibt sich für die maximale Tiefe

$$log_2 n \times 2 + 1$$

```
1 pane.children.clear()
2 pane.children.add(elementGroup)
3 pane.setPrefWidth(elementGroup.getBoundsInParent.getWidth)
4 scrollPane.vvalue = 0.0
5
6 val depth = Math.ceil(Math.log(elementGroup.children.size) / Math
     .log(2)) * 2 + 1
7 pane.setPrefHeight(depth * 130)
```

Wurde die zu sortierende Menge auf der Zeichenfläche platziert, kann der Benutzer auf „Run“ klicken, um den Algorithmus zu starten. Dadurch wird die Methode `runSorting` ausgeführt.

Listing 4.8: runSorting startet die Sortierung mit anschließender Animationen

```
1  def runSorting():Unit = {
2    runButton.disable = true
3    isPlaying() = true
4    playPauseMenu.disable = false
5    playPauseButton.disable = false
6    val elementGroup: javafx.scene.Group = pane.children.get(0).
        asInstanceOf[javafx.scene.Group]
7
8    val sorter = new SortElementsController(pane, consoleLog)
9    sorter.maxDepth =  Math.ceil(Math.log(elementGroup.children.
        size) / Math.log(2)) * 2 + 1
10   sorter.sort(elementGroup, 0)
11   transition = sorter.getSequence
12   transition.rate <== MathBindings.pow(2.0, playbackSpeed.value)
13
14   transition.play()
```

```
15    transition.onFinished = {
16      event: ActionEvent =>
17        cleanEverythingUp
18    }
19  }
```

Durch die Methode `runSorting` wird der Run-Button deaktiviert und der Algorithmus zum Sortieren der Elemente durchgeführt, welcher die vorher generierte Gruppe als Eingangsmenge übernimmt. Anschließend wird die Animation, die den Algorithmus visualisiert, gestartet.

Die Implementierung der Logik für den Algorithmus befindet sich in der Klasse `SortElementsControlle`

Wie schon erwähnt, wird bei unserer Implementierung des Mergesort-Algorithmus nicht mit einer einzigen Gruppe gearbeitet, sondern mit vielen Listen, die nicht überschrieben werden. Das dient dazu, die schon gesplitteten und gemergten Elemente auf der Zeichenfläche unangetastet zu lassen. Um dies zu realisieren, müssen Duplikate der Eingangslisten erstellt werden, mit denen weitergearbeitet werden kann, ohne die schon gezeichneten Objekte zu beeinflussen. Man braucht also eine zweite Liste, welche Objekte mit äquivalenten Werten enthält. Für diese Aufgabe existiert die Methode `createGroup`, welche in unserer `sort` - Methode aufgerufen wird.

Listing 4.9: Erstellung der Gruppe

```
1  def createGroup(parentGroup: Group, splitList: (List[SortElement
       ], List[SortElement]), part: EnumVal, depth: Int): Group = {
2
3    val duplicateList = (part match {
4      case Part.Left => splitList._1
5      case Part.Right => splitList._2
6      case _ => throw new IllegalArgumentException(s"$part␣is␣not␣a
         ␣valid␣argument")
7    }).map(_.duplicate())
8
9    val group = new Group() {
10       opacity = 0.0
11       children.addAll(duplicateList.asJava)
12       translateY = parentGroup.translateY() + (if (depth > 0)
           moveDownByPixel else 0)
13       id = s"level-${depth␣+␣1}"
14     }
15     part match {
16       case Part.Left =>
17         group.translateX <== parentGroup.translateX - group.
             getBoundsInParent.getWidth / 2 + SortElement.width / 2
18       case Part.Right =>
19         group.translateX <== parentGroup.translateX + parentGroup
             .getBoundsInParent.getWidth / 2 - group.
```

```
                getBoundsInParent.getWidth/2 - SortElement.width/2
20
21      }
22    group
23  }
```

Die Methode createGroup erzeugt gleich zu Beginn ein Duplikat der eingegangenen splitList und zwar, abhängig vom mitgegebenem Enum, entweder von der linken oder der rechten Hälfte und speichert dieses in der Value `duplicateList`. Die `map`–Methode, die hier verwendet wird, erzeugt für jedes SortElement-Objekt in einer SplitList ein Objekt mit gleichen Werten und gibt dieses an die neue Liste weiter. Der Unterstrich steht in diesem Fall also für jedes Element. Nachdem alle Elemente der „duplicateList“ zu einer Gruppe hinzugefügt wurden, wird diese Gruppe, relativ zu der Gruppe, aus der sie entstanden ist, ausgerichtet. Durch das verwendete Binding `<==` wird die Positionierung stets angepasst, falls sich die Position der Elterngruppe ändern sollte.

4.5 Implementierung des Autoscrolls

Um dem Benutzer den Fokus auf den Algorithmus zu erleichtern, haben wir es für sinnvoll erachtet, einen automatischen Scrollmechanismus zu implementieren, welcher stets das Geschehen zum Mittelpunkt der Szene macht. Hierbei machen wir uns noch einmal die zuvor für die Größe der Zeichenfläche genutzte Tiefe der Gruppenstruktur zu Nutze.

Listing 4.10: Das Autoscrolling auf der Pane

```
1  def scroll(group: Group, depth: Int) = {
2    val factor = 1.0/(maxDepth)
3    val timeline = new Timeline {
4      autoReverse = false
5      keyFrames = Seq(
6        at (0.5.s) {
7          group.getScene.lookup("#scrollPaneID").asInstanceOf[
               ScrollPane].vvalue -> (factor * (if(depth == 0){depth}
                else {depth + 1}))
8        }
9      )
10   }
11
12   sequence.children.add(timeline)
13 }
```

4.6 Implementierung der Animationen

Die Methode `relocateElementGroup` erzeugt eine Animation, welche die übergebene Gruppe in 0.2 Sekunden sichtbar macht und sie binnen einer Sekunde von der Elterngruppe um einen festgelegten Wert nach unten bewegt. Hierbei ist zu beachten, dass die `threadNumber` mit an die Methode übergeben wird, da bei mehreren Threads jeder seine eigene Animationssequenz besitzt, die später parallel zu den Animationssequenzen der anderen Threads abgespielt wird.

Listing 4.11: Umpositionierung der Elemente

```
1  def relocateElementGroup(group: Group, depth: Int, threadNumber:
       Int): Timeline = {
2    val timeline = new Timeline {
3      autoReverse = false
4      keyFrames = Seq(
5        at(0.2.s) {
6          group.opacity -> 1.0
7        },
8        at(1.s) {
9          group.translateY -> (group.translateY() +
               SortElementsController.moveDownByPixel)
10       }
11     )
12   }
13   addToSequence(threadNumber, timeline)
14
15   timeline
16 }
```

Um den Threads ihre Animationen zuzuteilen, wird die Methode `addToSequence` benutzt. Hier werden zwei verschiedene Transitionen für die beiden möglichen Threads gepflegt. Zusätzlich gibt es noch eine weitere Transition, die den final merge beim Sortieren mit zwei Threads übernimmt.

Listing 4.12: Einteilung der Animationen in die entsprechende SequentialTransition

```
1    def addToSequence(threadNumber: Int, timeline: Timeline):
         Boolean = {
2
3      if (threadNumber == 0) seq1.children.add(timeline)
4      else if (threadNumber == 1) seq2.children.add(timeline)
5      else {println("Adds␣transition␣to␣last␣group"); groupSeq.
           children.add(timeline)}
6    }
```

5 Bedienungsanleitung

5.1 Bedienen

Im Normalfall liegt der Visual-Megesort als Datei mit der Endung `.jar` vor. Die Dateiendung JAR kennzeichnet Archive, die mehrere Java-Dateien und deren Metainformationen enthalten. Um die Datei ausführen zu können, muss die `Java Runtime Environment` installiert sein. Die Laufzeitumgebung kann gegebenenfalls kostenlos im Internet heruntergeladen werden.

Die Anwendung kann man durch einen Doppelklick, oder aus der Konsole durch den folgenden Aufruf starten.

```
java -jar Visual-Mergesort.jar
```

Startet man die Applikation, so öffnet sich folgendes Fenster (Abbildung 5.1):

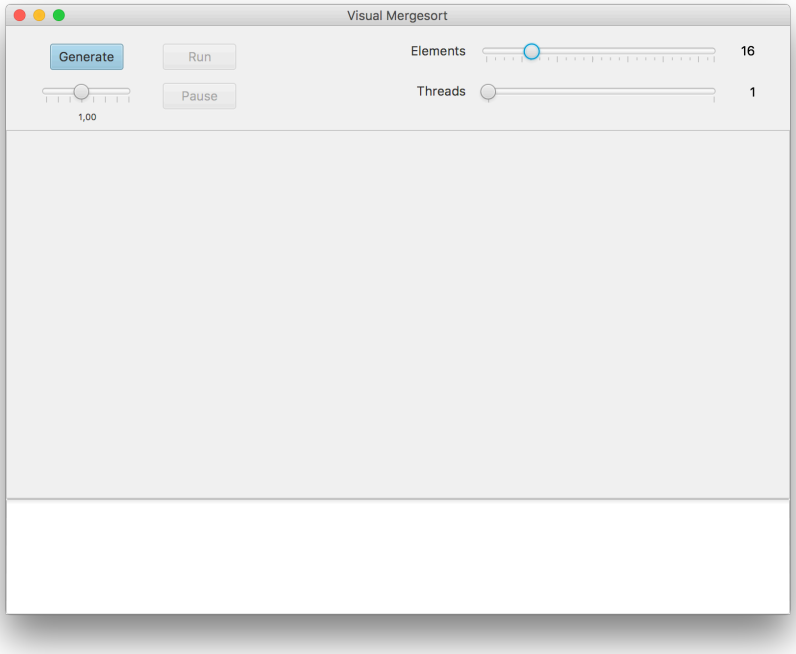

Abbildung 5.1: Applikation direkt nach dem Starten

Über `ENTER` oder das Klicken auf `Generate` können direkt die über den Slider voreingestellten 16 Zufallselemente in willkürlicher Reihenfolge generiert werden. Alternativ kann über die Menüleiste `File` → `Generate Random Data` eine andere Reihenfolge ausgewählt werden. Um einen schnellen Start mit der gewünschten Menge zu ermöglichen, existieren folgende Shortcuts, welche optional verwendet werden können:

Je nach benutztem Shortcut wird eine andere Menge von Elementen auf der Zeichenfläche platziert. Der Slider für die Anzahl der zu zeichnenden Elemente gilt bei allen Optionen, außer bei der generierung von benutzerspezifischen Elementen (siehe unten).

STRG + B Generiert eine Menge von Zufallszahlen in willkürlicher Reihenfolge (default).

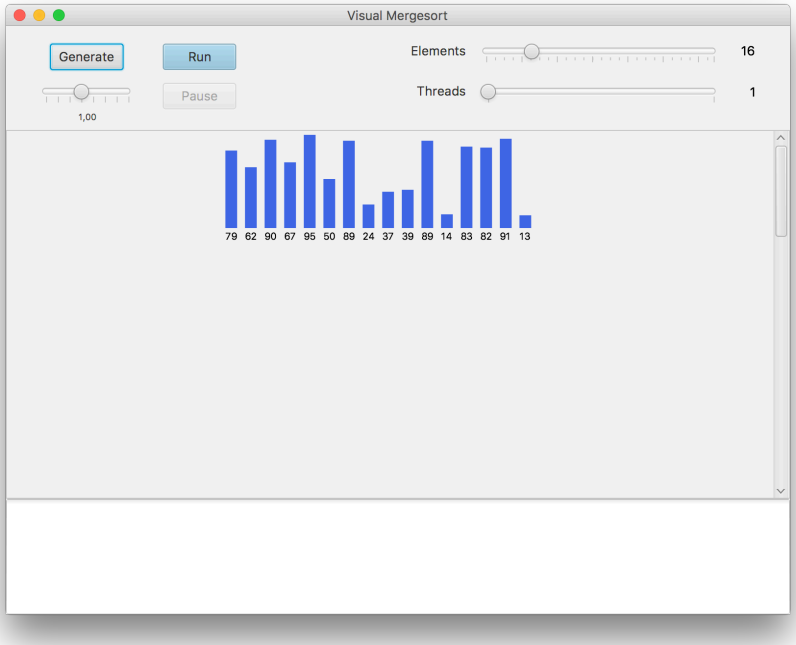

Abbildung 5.2: Elemente in zufälliger Reihenfolge

STRG + O Generiert eine Menge von Zufallszahlen in vorsortierter, aufsteigender Reihenfolge.

STRG + I Generiert eine Menge von Zufallszahlen, welche absteigend sortiert ist.

STRG + U Sowohl die Menge der Zahlen als auch die Reihenfolge kann über den Benutzer manuell eingegeben werden. Hierzu öffnet sich ein Fenster, bei dem die gewünschten Werte eingegeben werden. Dabei ist darauf zu achten, dass jeder Wert durch ein Komma vom nächsten Wert getrennt wird.

Bestätigt man anschließend durch das Drücken auf den Button `OK`, erscheinen die eingegebenen Werte als Elemente auf der Zeichenfläche

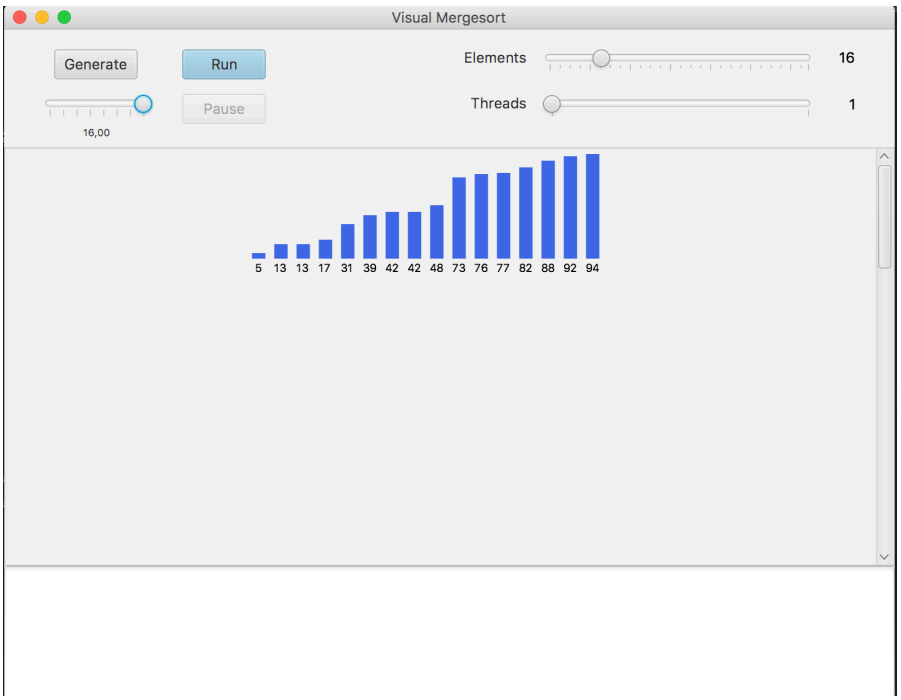

Abbildung 5.3: Elemente vorsortiert in aufsteigender Reihenfolge

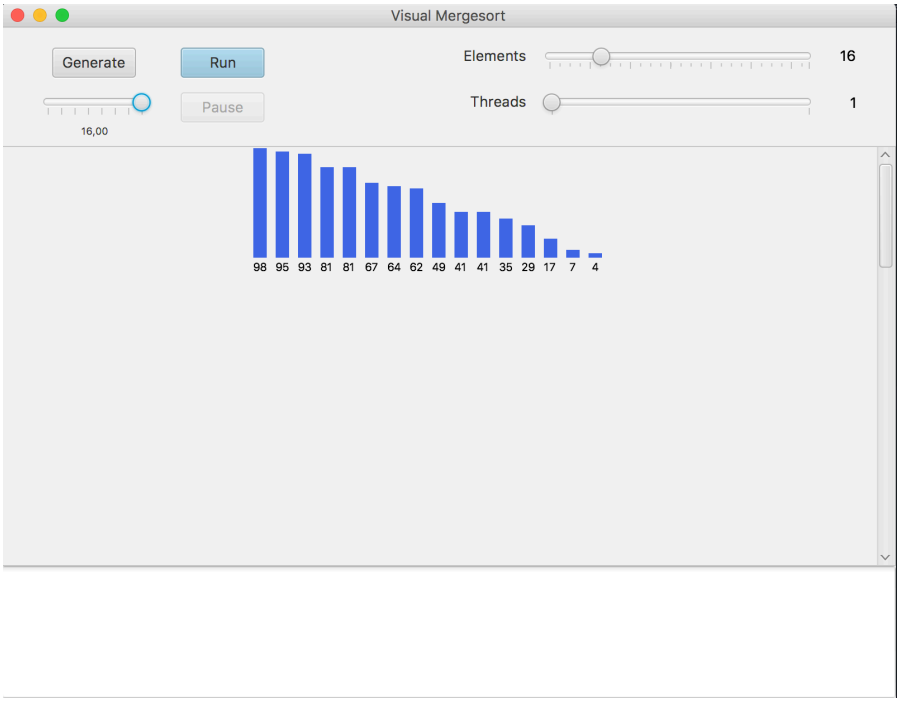

Abbildung 5.4: Elemente vorsortiert in absteigender Reihenfolge

Mit `ENTER`, einem Klick auf **Run** oder dem optionalen Shortcut `STRG + r` wird der

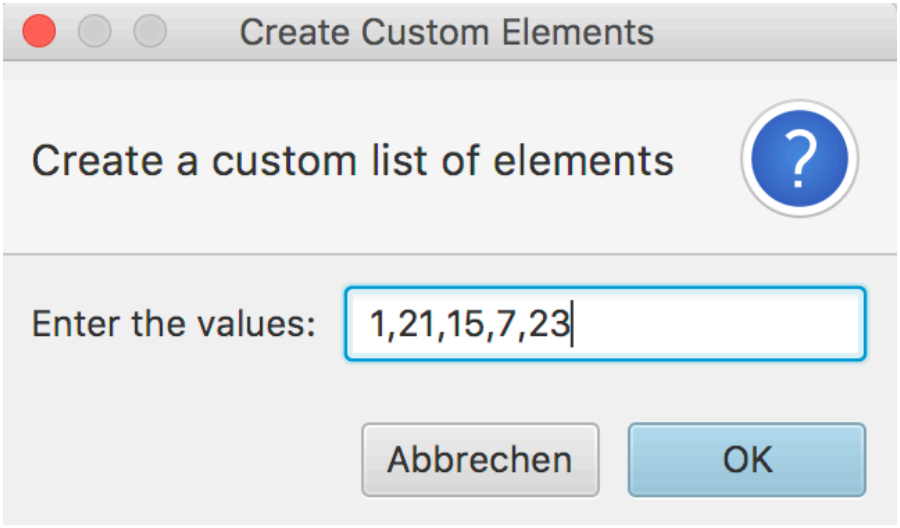

Abbildung 5.5: Generieren von benutzerspezifischen Elementen

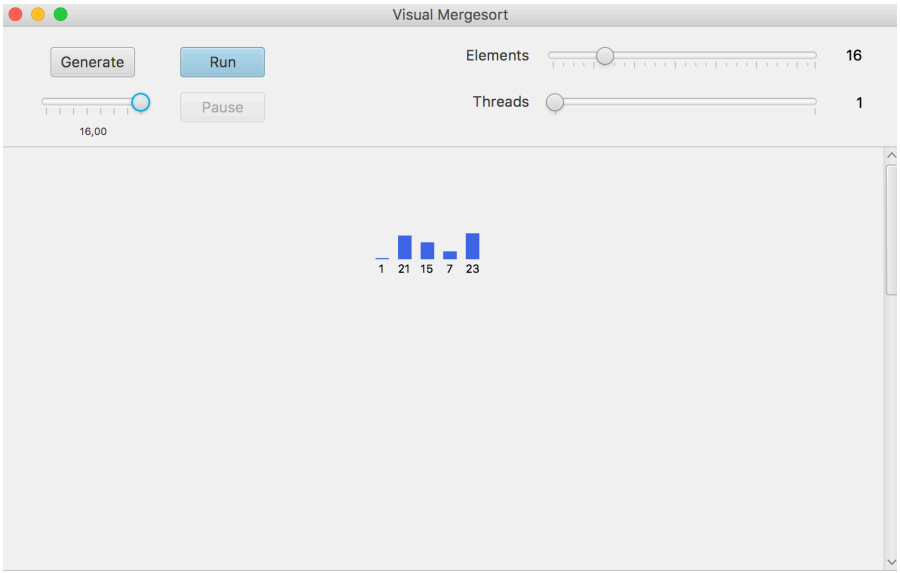

Abbildung 5.6: Benutzerspezifische Elemente wurden generiert

Sortieralgorithmus gestartet. Über den Slider mit der Signatur *Threads* kann die Anzahl der für den Algorithmus verwendeten Threads eingestellt werden. Hierbei kann man zwischen einem und zwei Threads wählen.
Wurde die Animation gestartet, kann diese jederzeit in ihrer Geschwindigkeit über den Regler unter dem Generate-Button variiert oder über den Button *Pause* bzw. *Play* komplett pausiert bzw. fortgesetzt werden. Beim Starten der Applikation fällt mit Sicherheit auf, dass sich das System automatisch zum Geschehen mitbewegt, um das Vorgehen besser zu visualisieren. Möchte man sich bestimmte Teile genauer ansehen, kann die Anwendung pausiert werden - Anschließend kann man sich frei auf der Zeichenfläche

bewegen.

Darüber hinaus kann über den Menüpunkt View die Ansicht der Applikation zu jeder Zeit angepasst werden. Hier existieren die Optionen `Toggle Log Console` und `Toggle Action Bar`. `Toggle Log Console` sorgt für das Ein- und Ausblenden der Konsole im unteren Teil der Anwendung. Dadurch kann man bei Bedarf die zusätlichen Informationen zu `Split` und `Merge` ausblenden, um mehr Platz für die eigentliche Animation zu schaffen. `Toggle Action Bar` ist hierbei aus den gleichen Gründen für das Ein- und Ausblenden der Schalt- und Kontrollflächen im oberen Teil der Applikation zuständig. Zusätzlich können diese Funktionen optional auch über Tastenkombinationen ausgeführt werden:

STRG + K Blendet die Aktionsbar ein und aus.

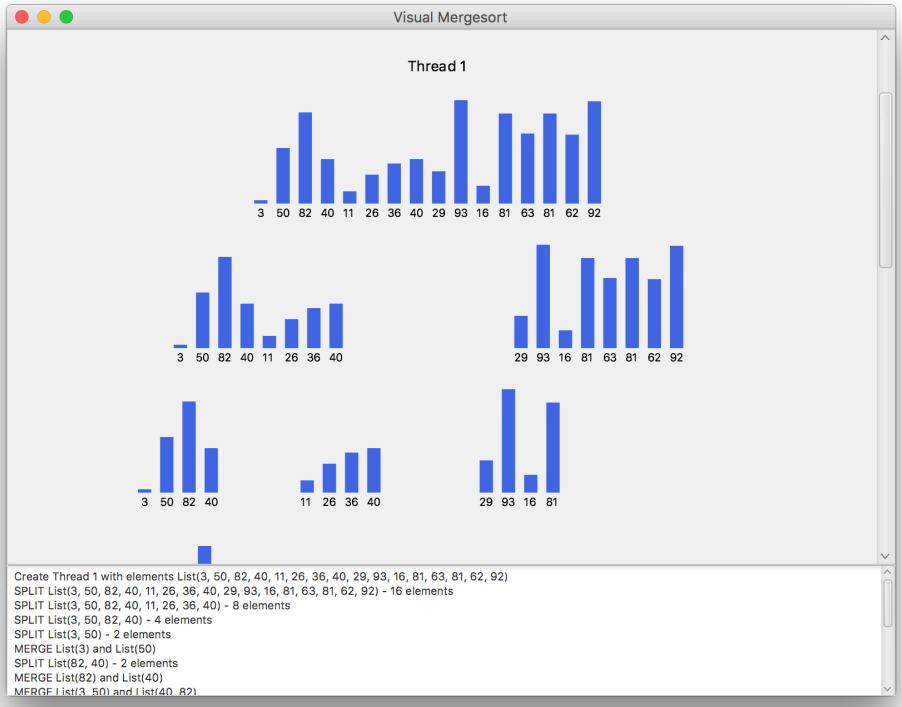

Abbildung 5.7: Visual Mergesort mit ausgeblendeter Aktionsbar

STRG + L Blendet die Konsole ein und aus.

Um den Algorithmus effizienter zu machen, ist es möglich, beim Visual Mergesort die Anzahl der verwendeten Threads, die den Sortieralgorithmus ausführen, zu verändern. Dadurch kann sich der Benutzer vor Augen halten, wie der Mergesort parallelisiert werden kann, um dessen Leistung zu steigern. Wurde über den Slider `Threads` die Zahl 2 ausgewählt, wird die zu sortierende Liste beim Starten der Anwendung über `Run` direkt

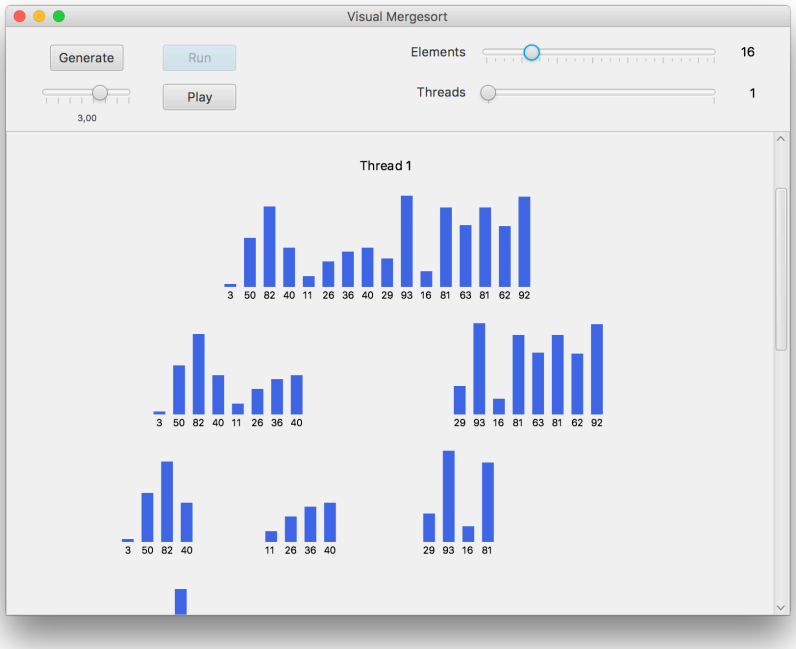

Abbildung 5.8: Visual Mergesort mit ausgeblendeter Konsole

in zwei Listen geteilt, welche jeweils einem `Thread 1` und einem `Thread 2` zugewiesen werden:

Ab hier führt jeder Thread den Mergesort-Algorithmus für die Liste durch, die ihm zugeteilt wurde, bis diese vollständig sortiert ist. Zum Schluss kommt es zu einem finalen Merge, bei dem die beiden durch die Threads generierten Teillisten zu einer Ganzen zusammengefügt werden. Hierbei kann es, beispielsweise bei einer ungeraden Anzahl an Ausgangselementen dazu kommen, dass ein Thread mehr Zeit benötigt, als der andere, da dieser ein Element weniger sortieren muss. Für diesen Sonderfall wird die Autoscrollfunktion automatisch deaktiviert, da die Threads an verschiedenen Stellen auf der Zeichenfläche arbeiten. Vor dem final Merge wartet der eine Thread auf den anderen, bis dessen Liste auch sortiert ist.

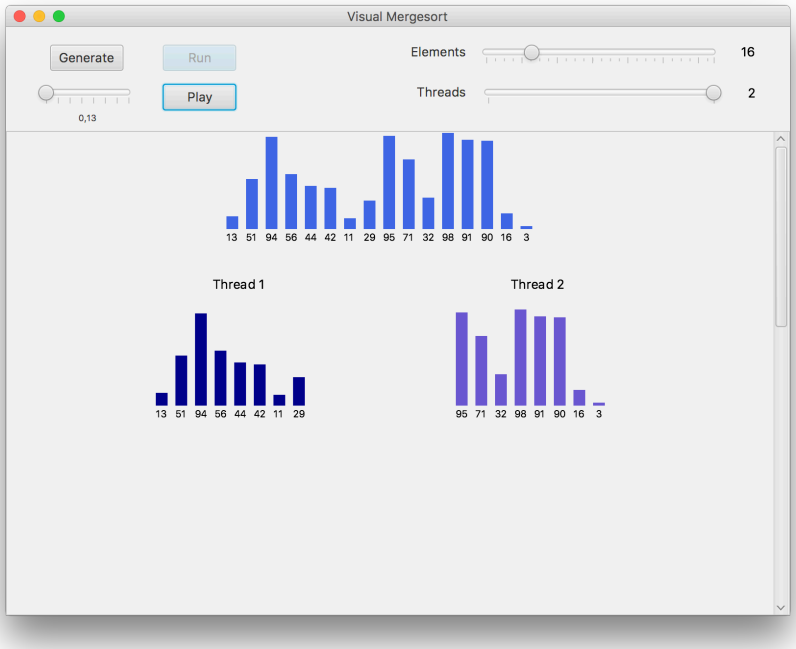

Abbildung 5.9: Visual Mergesort mit zwei Threads

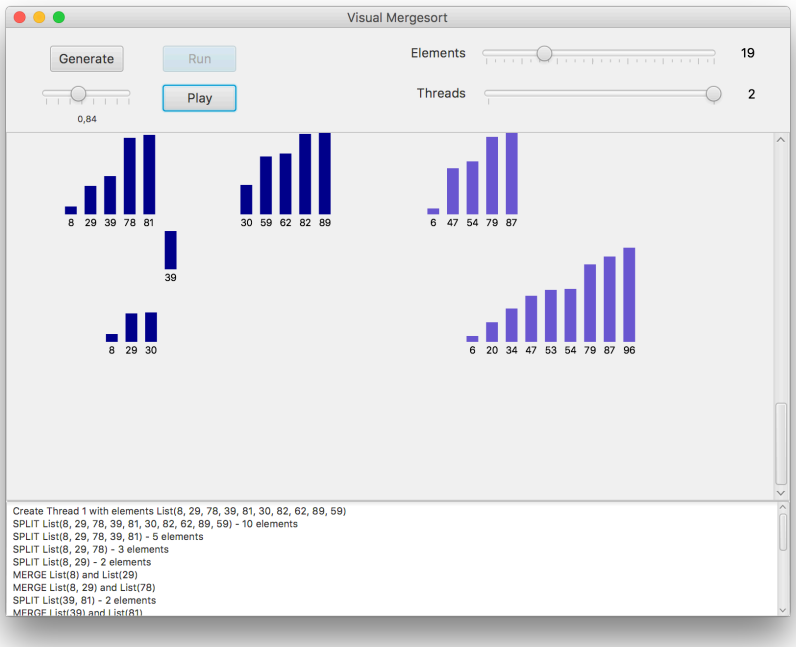

Abbildung 5.10: Thread 2 wartet auf Thread 1

5.2 Weiterentwickeln

Falls man das Programm manuell kompilieren möchte, oder man die Funktionen weiterentwickeln möchte, haben wir hier eine kleine Anleitung: Welche Programme man benötigt, wo welche Dateien liegen, wie man das Programm manuell kompiliert und wie man schlussendlich eine ausführbare Datei bekommt.

5.2.1 Voraussetzungen

Wenn man das Programm selber kompilieren und weiter-entwickeln möchte, benötigt man die folgenden Dinge:

1. SBT (Scala Build Tool)
2. git
3. JVM und JDK

Wir gehen davon aus, dass sowohl JVM als auch JDK installiert sind. Alternativ kann man auf der Oracle–Seite[1] nachlesen, wie man diese für sein Betriebssystem installiert.

Das Scala Built Tool kann man auf der Projektseite[2] herunter laden, und im Anschluss kann man dieses installieren. Über dieses Build Tool wird dann später Scala in der entsprechenden Version geladen, sodass man Scala selber nicht installieren muss. Möchte man Scala dennoch installieren, um beispielsweise die `REPL` [Ale] zu nutzen, so kann man Scala über die Webseite[3] herunter laden.

Zuletzt benötigt man noch git[4], um das Projekt über Github beziehen zu können.

5.2.2 Programm starten

Im Nachfolgenden werden die Befehle aus im Terminal ausgeführt. Diese Schritt-für-Schritt–Anleitung wurde auf Linux und OSX getestet, sollte jedoch auf Windows ähnlich funktionieren.

Zuerst muss das Projekt von Github geladen werden. Über git kann man den folgenden Befehl ausführen, um das Projekt von Github zu `clonen`.

```
git clone https://github.com/TobsCore/Visual-Mergesort.git
```

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html
[2] http://www.scala-sbt.org/index.html
[3] http://scala-lang.org/
[4] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Es ist ein neues Verzeichnis mit den Projektdaten erstellt worden. Wenn man in dieses Verzeichnis navigiert, befinden sich dort zwei Ordner und eine `README.md` Datei.

```
cd Visual-Mergesort/
ls
```

In den Ordnern befinden sich die folgenden Daten:

Code/ In diesem Ordner ist das eigentliche Programm enthalten, sowie die sbt–Konfiguration und die Intellij Dateien.

Documentation/ Hier ist die Dokumentation über unsere Projektarbeit enthalten, die wir geschrieben habe. Die Dokumentation ist in LaTeX geschrieben und kann über `xelatex` kompiliert werden. Mehr dazu in 5.2.3.

Wenn man nun in den `Code/` Ordner wechselt, befinden sich darin wiederrum 4 Ordner (3 sichtbar, 1 nicht sichtbar):

project/ Hier sind die sbt Projektdateien enthalten.

src/ Dies ist der wichtigste Unterordner. Hier ist der Programmcode enthalten, da es sich um den `Source`–Ordner handelt.

target/ Hier landen kompilierte Dateien und Caches. Die ausführbare `Visual-Mergesort.jar`–Datei wird beispielsweise auch in diesen Ordner geschrieben.

.idea/ Dies ist der Intellij Projektordner.

Zuletzt möchten wir noch die Ordnerstruktur des `src/` Ordners erklären. In dem Unterordner `main/` befinden sich noch die Ordner `resources/` und `scala/`. In `resources/`, sind Bilder, die CSS Dateien und die `FXML` Dateien enthalten. Der ganze Programmcode ist in dem Ordner `scala/`.

Um das Programm nun kompilieren zu können, gehen wir zurück in den Ordner `Visual-Mergesort/Code/`. In diesem Ordner sollte sich eine Datei mit dem Namen `build.sbt` befinden. Wenn man nun den Befehl

```
sbt compile
```

ausführt, so wird der Programmcode kompiliert und kann im Anschluss über

```
sbt run
```

ausgeführt werden. Nun wird die Anwendung gestartet.

Ausführbare Datei erstellen

Da man nicht immer den Programmcode beziehen und kompilieren möchte, ist es sinnvoll, wenn man sich eine ausführbare Anwendung erstellen lässt. Dies ist, wie bereits

erwähnt, eine Datei mit der Endung `.jar`. Wir benutzen ein sbt–Plugin, das sich um die Erstellung dieser Datei kümmert. Man kann die Erstellung über

```
sbt assembly
```

anstoßen. Die generierte Datei liegt dann in dem Ordner `target/scala-2.11/` und heißt `Visual-Mergesort.jar`. Mit

```
java -jar Visual-Mergesort.jar
```

kann man die Anwendung starten.

5.2.3 Dokumentation

Die Dokumentation ist in LaTeX geschrieben und kann über `xelatex` erstellt werden. Das Programm kann auf Linux, Windows und OSX installiert werden. Wir haben zusätzlich ein `Makefile` angelegt, das das Kompilieren und anschließende Aufräumen für uns übernimmt. Zusätzlich kann man hierüber auch die `Bibtex` Datei in das Dokument einbauen. `BibTex` wird genutzt, um die Literaturquellen zu verwalten. Die Datei `Documentation/README.md`[5] beschreibt, welche Befehle man über Make[6] ausführen kann und was diese Befehle bewirken.

[5] Auch hier zu finden: https://github.com/TobsCore/Visual-Mergesort/blob/master/Documentation/README.md

[6] Es kann sein, dass man das Make–Progamm installieren muss, um das Makefile nutzen zu können. Dies war bei uns zumindest auf Windows–Systemen der Fall

6 Zusammenfassung & Fazit

Nachdem wir nun die letzten Wochen und Monate mit der Entwicklung der Anwendung verbracht haben, schließen wir das Projekt nun ab. Wir haben viele Dinge gelernt, uns mit neuen Konzepten befasst und eine neue Programmiersprache gelernt. Aber wir haben uns auch mit einigen teils hartnäckigen Problemen auseinander setzen müssen, die uns Zeit und Nerven gekostet haben und dazu geführt haben, dass wir nicht alles wie geplant umsetzen konnten.

6.1 Reflexion des Vorgehens

6.1.1 Aufgabenteilung

Wir haben uns direkt zu Beginn des Semesters bei unserem Dozenten um das Thema beworben, ohne, dass dieses ausgeschrieben war. Der Hintergrund war, dass wir die Thematik, die Visualisierung eines Algorithmus' interessant fanden, aber auch die Programmiersprache Scala hatte ihren Reiz.

Nachdem wir die Zusage bekamen, richtete Herr Kerst ein git-Repository[1] auf github ein und erstellte dort eine Vorlage für diese Dokumentation, die in LaTeXgeschrieben sein sollte. Außerdem wurde von ihm das eigentliche Scala Projekt eingerichtet, also die Ordner-Struktur und die Entwicklungsumgebung `IntelliJ IDEA`. Da wir uns entschieden, `sbt` als Build Tool und für das Verwalten von *Dependencies* zu verwenden, wurde schon früh die `build.sbt` Datei erstellt und mit den nötigsten Informationen befüllt, sodass man ein Programm mit Scala kompilieren konnte.

Nachdem diese Vorbereitungen getroffen waren, lasen wir uns intensiv in die Sprache ein. Hierbei ging es nicht nur darum, die Syntax zu verstehen und anwenden zu können, sondern auch die in der Sprache üblichen `Code Conventions` kennen zu lernen und uns mit der funktionalen Programmierung etwas näher auseinander zu setzen. Das spiegelt sich zum Beispiel darin wieder, dass wir probiert haben, nur `val`–Values zu benutzen, anstelle von möglicherweise `var`iablen Werten. Auch die Schleifen sind häufig ohne direkte Zählvariable, sondern mit `zipWithIndex` geschrieben worden und wir nutzten auch sonst viele Scala Features. Ganz besonders zu erwähnen seien hier `Pattern Matching` und der (quasi ternäre) `ternäre if`–Operator.

[1] https://github.com/TobsCore/Visual-Mergesort/

Die Programmiersprache Scala zu lernen hat zwar viel Zeit in Anspruch genommen, doch hat uns im Endeffekt sehr geholfen, da wir uns nicht mit elementaren Syntax-Problemen auseinander setzen mussten, sondern uns bei der Entwicklung auf die wesentlichen Probleme beschränken konnten.

Unser Test–Programm

Wir haben die Entwicklung mit einer anderen Anwendung begonnen, die nichts mit der eigentlichen Projektarbeit zu tun hatte. Wir haben uns dafür entschieden, damit wir uns gemeinsam an die Scala Syntax gewöhnen und wir wollten dadurch ein besseres Verständnis von ScalaFX erlangen. Die Test-Anwendung hat den User dazu aufgefordert eine Zahl in ein Textfeld einzugeben und hat dann der Eingabe entsprechend viele Balken auf einen `Canvas` gezeichnet. Dies war bereits eine wichtige Entscheidung, da wir hierbei merkten, dass man die Elemente auf einer `Canvas` nicht ausreichend manipulieren kann und wir uns somit für eine `Pane` als *Zeichenfläche* entschieden.

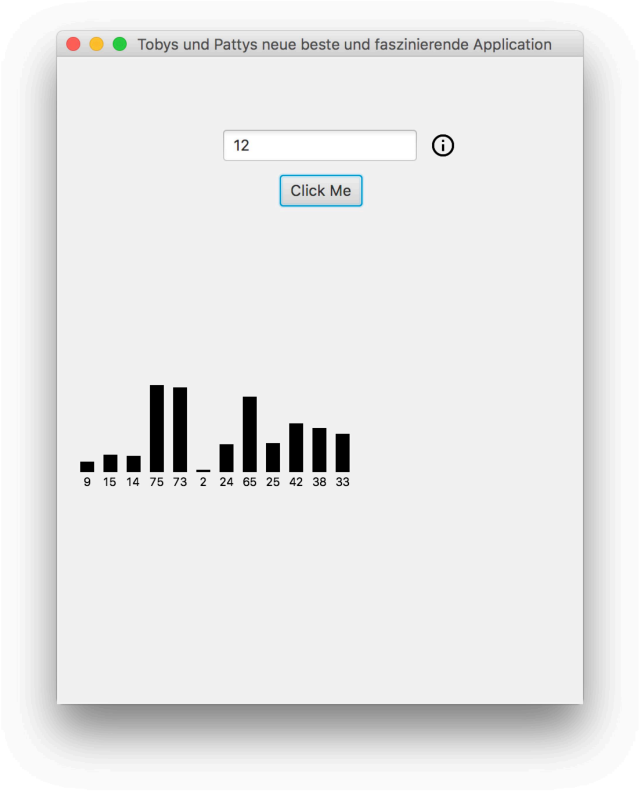

Abbildung 6.1: Test Anwendung: Generierung der Balken

Außerdem war es eine gute Entscheidung, dass wir uns erst mit einer Anwendung beschäftigten, bei der wir viele Fehler machen konnten, da wir somit keine Design-Entscheidungen im späteren Verlauf bereuen würden.

Das Resultat ist, dass wir selbstbewusst die Beispielsanwendung löschten und uns mit einem ausreichend guten Vorwissen an die Entwicklung der eigentlichen Aufgabe machten. Wir wussten nun, wie man auf Events reagiert, wie man Elemente in FXML definiert und in dem Controller anspricht. Wir konnten Elemente nun manipulieren (zwar auf einem `Canvas`) und kannten uns nun besser mit der baumartigen Struktur von JavaFX aus, bei der oben die `Stage`, darunter die `Scene` und darunter die `Node`s waren.

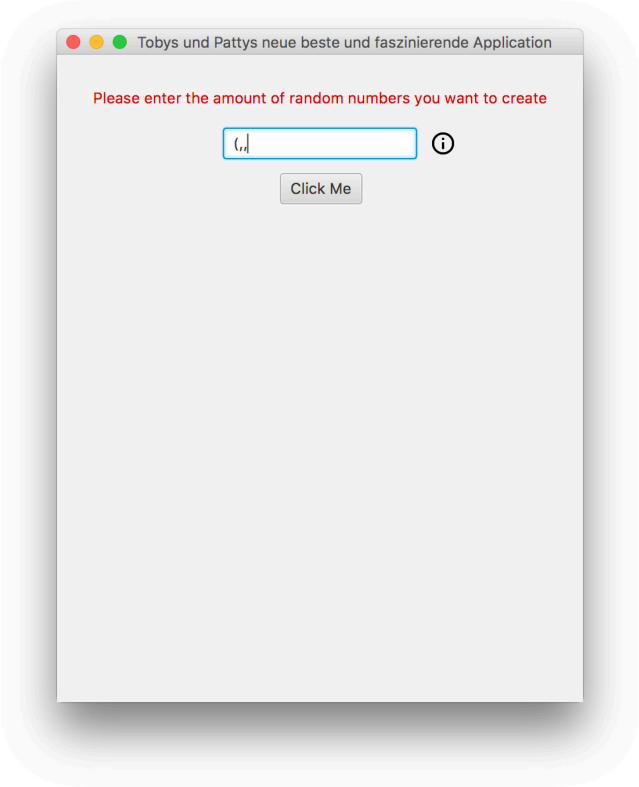

Abbildung 6.2: Test Anwendung: Fehlermeldung bei Falscheingabe

Gemeinsame Entwicklung

Danach haben wir uns täglich getroffen und gemeinsam an der Anwendung gearbeitet. Hierbei haben wir die meiste Zeit vor einem Bildschirm gesessen und gemeinsam entwickelt. Dies hat sich als sehr sinnvoll heraus gestellt, da man so schneller Fehlern vorbeugen konnte und wir kleine Probleme diskutieren konnten, ohne den anderen aus Gedanken reißen zu müssen, was eher passiert, wenn jeder an seinen Teil entwickelt. Somit nutzten wir die ganzen Vorteile von *Pair Programming*[2]

- Wir haben besseren Code geschrieben

[2] https://www.agilealliance.org/glossary/pairing/

- Wir kannten alle Code-Teile und konnten so Bugs schneller finden und fixen
- wir haben viel diskutieren können

Im späteren Verlauf wurden einzelne Teile vermehrt aufgeteilt, da es sich hierbei um *einfach zu schreibenden* Code handelte, oder weil es zeitlich sinnvoller war, die Arbeit aufzuteilen. Beispielsweise wurden die Shortcuts im Menü eingefügt, während der andere sich über ein bestehendes Problem informierte.

Wir erreichten, was man nur selten bei Abschluss eines Projekts behaupten kann: Eine gerechte Arbeitsteilung, bei der am Ende niemand von sich beansprucht, dass er mehr als der andere gemacht hat.

6.2 Kritische Betrachtung

Wir haben viele unserer gewünschten Ideen in das Programm implementiert und einige Features sind dazu gekommen, von denen wir es nicht erwartet hatten, dass sie überhaupt möglich sein werden. Es ist aber auch eine Sache weggefallen, die wir sehr gerne implementiert hätten.

6.2.1 Erwartete Features

Wir haben uns zu Beginn viele Gedanken gemacht, wie wir das Programm entwickeln werden und welche Features enthalten sein sollen und müssen. Dabei lagen uns die folgenden ganz besonders am Herzen:

Elemente als Balken

Da die Aufgabe die **Visualisierung** des Algorithmus' war, gab es keinen Zweifel, dass die Elemente nicht einfach als Zahl dargestellt werden können, sondern die Gewichtung auch sichtbar sein soll. Wir haben uns somit dafür entschieden, dass wir diese Gewichtung durch Balken darstellen. Dennoch sollte es einfach sein, die Werte zu vergleichen, damit der Algorithmus immer noch leicht verständlich bleibt.

Durch unsere Klasse `SortElement`, die von `Group` erbt, welche wiederum ein Node ist, konnten wir genau das erreichen. Wir hatten ein Objekt, das wir in Gruppen stecken, bewegen, färben und vergleichen konnten. Somit wurde der Code nicht nur lesbar, sondern die Implementierung auch sehr einfach und ein Nutzer der Anwendung bekommt die Gewichtung von Zahlen mit.

Einstellbare Anzahl an Elementen

Um das Programm für den User ansprechend und auch interaktiv zu machenn, wollten wir auf jeden Fall die Anzahl an Elementen einstellbar machen. Wir dachten zwar ursprünglich an ein Textfeld, in das der User eine Zahl eingeben kann, jedoch hat man dabei das Problem, dass man die Eingabe auf die folgenden Dinge prüfen muss:

- Ist der eingegebene String eine Zahl?
- Erlaubt man auch Double Werte, also Zahlen mit Komma?
- Wie kennzeichnet man den erlaubten Wertebereich?

Aufgrund dieser Probleme haben wir uns nach Alternativen umgesehen und mit dem `Slider` die perfekte Alternative gefunden, die unseren Ansprüchen genügte und die oben genannten Probleme auf eine sehr intuitive Weise löst.

Anzahl an Threads

Wir wollten auch, dass der User die Möglichkeit hat, die Anzahl an Threads einzustellen. Auch hier entschieden wir uns für einen `Slider`, da die oben beschriebenen Probleme auch bei der Auswahl an Threads existieren. In der jetztigen Form gibt es die Möglichkeit zwischen einem und 2 Threads zu wählen, wieso es nur diese beiden Möglichkeiten gibt, möchten wir in Section 6.2.3 erläutern.

Zufällig gewählte Elemente erzeugen

Uns war klar, dass auf irgendeine Art die Elemente erzeugt werden müssen, die dann sortiert werden. Diese Elemente sollten zufälllig generiert werden.

Sortierung durch Animationen

Um den Algorithmus zu veranschaulichen, sollten die Elemten aufgespalten und wieder gemerget, also zusammengeführt werden. Diese Schritten sollten durch *herumfliegende* Elemente klar erkennbar sein, wofür sich die JavaFX Transitions anbieten.

Text-Ausgabe über das, was gerade passiert

Um den Algorithmus nicht nur visuell zu erklären, wollten wir, dass der User *textuell* darüber informiert wird, was gerade passiert und mit welchen Elementen.

6.2.2 Hinzugekommene Features

Während der Entwicklung merkt man oft, welche essentiellen Features noch fehlen, welche teilweise leicht implementiert werden können, oder aber die Nutzbarkeit der App wesentlich verbessern.

Generierung von vorsortierten Elementen

Die Generierung von vorsortierten Elementen ist ein Beispiel für eine Feature, das schnell implementiert war, aber die Nutzung der Applikation interessanter macht. Da man durch dieses Feature ein für das Algorithmus typisches Verhalten gut beschreiben kann, haben wir uns entschieden, dass Elemten *sortiert* und *invers sortiert* generiert werden sollen.

Interaktive Eingabe von Elementen

Dieses Feature haben wir implementiert um unser Programm besser Testen zu können. Da man teilweise nicht nur zufällig generierte Zahlen testen möchte, sondern ganz bestimmte, haben wir uns dazu entschieden, dass man die Elemente durch einen Dialog eingeben kann. Diese müssen dann jedoch an Zulässigkeit geprüft werden und im Anschluss erzeugt werden. Dieses Feature hat sich nicht nur für Testzwecke als praktisch erwiesen, sondern auch für den *produktien* Einsatz und ist deshalb nun Bestandteil der Anwendung.

Autoscrolling

Dieses Feature hat uns viel Zeit gekostet, aber hat sich im Endeffekt gelohnt. Wir haben uns gedacht, dass ein interaktiv zu benutzendes Programm zwar toll ist und dass man von dem User erwarten kann, dass er auf der Pane scrollt, jedoch haben wir es als störend empfunden, dass man das Programm nicht *einfach laufen lassen* konnte. Man musste immer aktiv eingreifen um zu der Stelle zu gelangen, an der der Algorithmus gerade arbeitet. Dies war vor allem bei sehr vielen Elementen anstrengend.

Wir haben aber mit Einschränkungen leben müssen. Bei zwei Threads ist das Autoscrolling nur wirklich nutzbar, wenn man Anzahl an Elementen gerade ist, da das Autscrolling sonst dafür sorgt, dass die Pane immer zum aktiven Geschehen scrollen möchte, was jedoch an zwei verschiedenen Orten ist. Das Resultat ist, dass man von dem Algorithmus nichts mehr mitbekommt, deswegen haben wir das Feature für zwei Threads bei einer ungeraden Anzahl von Elementen deaktiviert.

Play, Pause und die Animationsgeschwindigkeit

Unsere App sticht durch das Feature hervor, dass man die Animationen pausieren und wieder starten kann. Dadurch kann man, beispielsweise bei der Präsentation auf Eigenarten des Algorithmus' hinweisen und an eine andere Stelle scrollen. Das Einstellen der Geschwindigkeit ist macht die Benutzung der Applikation erst so richtig spannend, denn teilweise möchte man an einen Punkt gelangen und kann es nicht erwarten, bis das Programm da ist. Wenn man diesen Punkt dann erreicht hat, dann möchte man das Geschehen jedoch genau verfolgen und die Geschwindigkeit runter stellen. Genau das haben wir durch den Geschwindigkeits-Slider erreicht und freuen uns auch, dass dieses Feature stabil und verlässlich funktioniert.

Ein– und Ausblenden der Leisten

Das Ein– und Ausblenden der Logging-Konsole und der Controls ist kein wichtiges Feature, ist aber sehr zweckdienlich, wenn man sich auf die Visualisierung des Algorithmus beschränken möchte. Gerade dadurch, dass die Rekursion durch Verschieben nach unten visualisiert wird und die Bildschirmhöhe begrenzt ist, kann man so mehr auf dem Bildschirm darstellen.

Shortcuts

Wir benutzen Shortcuts, um ohne umständliche Navigation die Visualisierung zu starten. Gerade für Präsentationszwecke ist dies sehr praktisch und auch während unserer Tests haben wir gemerkt, wie praktisch es ist, wenn man über die Eingabe zweier Befehle die Animation starten kann. Somit haben wir uns dafür entschieden, alle Menüpunkte mit Shortcuts zu versehen, da dadurch die Nutzung der Anwendung spielerisch wird.

6.2.3 Fehlende Features

Natürlich spielt der Faktor Zeit eine Rolle, wenn man eine solche Anwendung schreibt. Jedoch hat uns auch JavaFX einige Probleme bereitet, die dazu geführt haben, dass wir nicht alles so implementieren konnten, wie wir es gerne wollten. Wir möchten im Folgenden die Punkte nennen, die wir gerne noch implementiert hätten und auf die wir leider verzichten mussten.

Actors

Aktoren (im engl. *Actors*) ist ein Modell, das es erlaubt multithreading-fähige Anwendungen zu schreiben. Es handelt sich bei den Aktoren um eine Abstraktions-Schicht, die auf Threads aufbaut. Bei den Aktoren werden unveränderliche (*immutable*) Nachrichten

verschickt, auf die reagiert wird. Da wir den Mergesort-Algorithmus multithreading-fähig implementieren sollten, hatten wir uns fest vorgenommen, dass wir die Nebenläufigkeit mittels Aktoren implementieren. Hierbei sollte jeder Actor (also jeder Thread) eine Teilliste bekommen und sortieren. Das Mergen dieser sortierten Listen sollte dann durch eine `Synchronisation` geschehen. Die Nachricht zwischen den Aktoren wäre die Liste gewesen, die erst unsortiert und später sortiert zwischen den Aktoren ausgetauscht wird.

Diese Implementierungsstrategie ist intuitiv und stabil. Leider hat uns an dieser Stelle JavaFX im Stich gelassen. Die Manipulation der Elemente auf der Pane und das Starten der Transitionen ist leider nicht möglich, man bekommt eine Exception wenn man aus einem anderen Thread auf die Pane zugreifen möchte. Unser Programm nutzte leider von Grund auf den Ansatz, dass man Elemente auf der Pane verschiebt. Somit konnten wir keine Aktoren für die Berechnung und anschließende Animation benutzen. Da wir sehr lange probiert haben, das Aktorenmodell zu nutzen und zum Laufen zu bringen, mussten wir das parallele Ablaufen das Animation durch das mehrfache Abspielen zweier Animations-Sequenzen implementieren. Somit konnten wir uns jedoch auch nur auf zwei Threads konzentrieren.

Autoscrolling der Logging–Konsole

Das Autoscrolling in der Logging–Konsole ist zwar vorhanden, aber alles andere als stabil. Dies hat mit einem Fehler in JavaFX zu tun, das beim Ändern des Texts automatisch an die Position 0.0, also den Anfang scrollt. Da wir jedoch mit dem Text-`Property` arbeiten mussten, um die Darstellung während der Animation zu ermöglichen, mussten wir Werte finden, an denen man an das Ende Scrollen kann, nachdem an die Position 0.0 gesprungen wurde. Wir nutzen momentan eine `TextArea`, haben eine Implementierung sowohl über ein `TextFlow` als auch einen `ListView` versucht. Keine der Lösungen hat funktioniert und somit muss man sich nun mit einer etwas unrund scrollenden Logging–Konsole abfinden. Wir haben entschieden, dass diese Funktionalität keine Prioriät hat.

Merge: Einfliegen von ursprünglicher Position

Die Bestimmung der Positionen auf einer `Pane` ist nicht immer klar und somit mussten wir uns teilweise damit behelfen, dass wir Werte durch das Testen heraus bekommen. Oder das wir Features nicht zu unserer vollen Zufriedenheit implementieren. Ein Beispiel ist das Mergen. Hierbei kommen die Elemente zwar aus der richtigen Richtung, jedoch nicht von der richtigen Position. Dieses Problem hat uns ganze Tage geraubt, es ist nicht möglich die x–Position eine Elements genau zu bestimmen. Weder die absolute Position, noch die relative Position konnte bestimmt werden. Wir mussten es dann darauf beruhen lassen, dass man erkennt, ob ein Element aus der linken, oder rechten Gruppe kommt.

6.3 Fazit

Das Projekt war herausfordernd und spannend. Wir haben viele neue Dinge gelernt und besonders die Team–Arbeit hat sich als sehr wertvoll herausgestellt.

Wir sind beide begeistert von der Programmiersprache Scala, die anfangs zwar etwas kompliziert erscheint, aber bei längerer Benutzung wirklich viele praktische Features mitbringt. Viele davon haben wir in dieser Ausarbeitung bereits vorgestellt. Wir empfehlen auf jeden Fall, dass man sich genauer mit dieser Sprache befasst, da besonders die funktionale Programmierung ein sehr interessanter Ansatz ist, die Bibliotheks–Unterstützung durch die Integration in die JVM (und somit Java) sehr gut ist und man nicht unnötig viel Code schreiben muss. Java–Code wirkt im Vergleich mittlerweile umständlich und oft unnötig.

JavaFX, das wir durch ScalaFX genutzt haben, ist *die* neue Bibliothek, die man zur Entwicklung von Benutzeroberflächen nutzt. Als Bestandteil von Java 8 wird diese vermutlich noch lange weiter entwickelt werden. Das wir uns während dieser Projektarbeit intensiv mit diesem Framework auseinander gesetzt haben, wird sich spätestens dann auszahlen, wenn wir das nächste Mal eine Anwendung mit einer Oberfläche entwickeln werden.

Die Arbeit mit einem richtigen Build–Tool wie `sbt` und einem Versionskontrollsystem (`git`), hat von uns verlangt, dass wir uns nicht nur mit der Programmiersprache und Entwicklungsumgebung auseinander setzen, sondern auch die anderen Aspekte, mit denen sich ein Programmierer auseinander setzen muss, kennen lernen.

Was uns überrascht hat, ist die Tatsache, dass die Entwicklung des Algorithmus' nicht nur elementare Kenntnis des Verhaltens vorraussetzte: Man musste alle Aspekte des Algorithmus' verstehen, um sich darüber unterhalten zu können, wie man den Algorithmus nun am besten visualisiert.
Somit wurde das sehr theoretische Konzept sehr schnell etwas, das man benannt hat (`merge`,`split`,`sort`,`shuffle`) und bei dem man dann wusste wie es geschieht. Sehr schnell wurde so aus einem abstrakten Problem ein gestalterisches, bei dem man die Konzepte, die man nun benennen konnte, visuell so präsentieren musste, dass auch jemand, der den Mergesort–Algorithmus nicht kennt, versteht worum es geht.

Neben all dem technischen Know-How, das wir erworben haben, hat sich aber auch vor allem herausgestellt, wie Zusammenarbeit funktioniert und wie man sich verhält, wenn man auf einmal nicht mehr alles selber macht. Wir haben festgestellt, dass Kommunikation das A und O bei einem erfolgreichen Projekt ist. Da wir mit viel Begeisterung bei der Sache waren und die Entwicklung nicht nur zeitintensiv, sondern auch interessant, lehrreich und unterhaltsam war, blicken wir zufrieden auf das Resultat: Unser Visual-Mergesort.

Literaturverzeichnis

[Ale] Alvin Alexander. Getting started with the Scala REPL (command-line shell). http://alvinalexander.com/scala/getting-started-scala-repl-command-line-shell-options. [Online; Zugriff am 18. September 2015; Letztes Update vom 3. Juni 2016].

[Ale13] Alvin Alexander. *Scala cookbook : [recipes for object-oriented and functional programming]*. OReilly, Beijing, 1. ed. edition, 2013.

[Cze] Jörg Czeschla. Properties und Binding in JavaFX. http://javabeginners.de/Frameworks/JavaFX/Properties_und_Binding.php. [Online; Zugriff am 5. September 2016].

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, Mass. [u.a.], 2002.

[Sue12] Joshua D. Suereth. *Scala in depth*. Manning, Shelter Island, 2012.

[VGC+14] Johan Vos, Weiqi Gao, Stephen Chin, Dean Iverson, and James Weaver. *Pro JavaFX 8 : A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients*. SpringerLink : Bücher. Apress, Berkeley, CA, 2014.

[vig] vigoo. Projektseite von ScalaFX. https://github.com/vigoo/scalafxml. [Online; Zugriff am 15. September 2015; Version 0.2.2].

Abbildungsverzeichnis