Programmieren in Java:
Visualisierung von Sortieralgorithmen

von
© *Kai Richard König (12.03)*

07 2011

FOS/BOS Technik München

Zusammenfassung

Dieses Dokument gibt Aufschluss über die Überlegungen hinter der Entwicklung und Programmierung einer Java-Swing Desktop-Application zur Visualisierung von Sortieralgorithmen geben.

Inhaltsverzeichnis

Kapitel 1

Grobkonzept

1.1 Was macht das Programm ?

Das Java-Programm soll anhand von Visualisierungen die Funktionsweise von fünf[1] unterschiedlichen Sortieralgorithmen darstellen. Zur Verdeutlichung der Vorgänge wird jedem Algorithmus ein Balken-Diagramme zugewiesen, bei dem jeder Balken einem Zahlenwert entspricht, vergleicht der Algorithmus nun zwei Zahlen werden die korrespondierende Balken farblich hervorgehoben. Entscheidet der Algorithmus das einer der Balken vorschoben werden muss um alle Zahlenwerte in eine korrekte, aufsteigende Reihenfolge zu bringen so wird auch diese Operation im Balken-Diagramm durch einen farbliche Hervorhebung visualisiert.

Um die dargestellten Algorithmen noch besser verstehen zu können, laufen alle Operation synchronisiert und parallel ab, das heißt jeder Algorithmus kann pro Zeit nur eine Änderung an den Zahlenwerten durchführen. Diese Restriktion stellt sicher das die Ergebnisse reproduzierbar und vergleichbar bleiben. Dazu hat der Benutzer noch die Möglichkeit spezielle Zahlenwert/Balken Kombinationen auszuwählen um schwächen und stärken einzelner Algorithmen besser sichtbar zu machen. Denkbar wären zum Beispiel Kombinationen die in umgekehrter Reihenfolge vorliegen oder welche bei denen nur wenige Werte an der „falschen“ stelle sind oder auch solche bei denen es viele gleiche Werte gibt.

Wichtig ist vorallem das dem Anwender bei der Betrachtung klar wird welcher Algorithmus sich für welche Problemstellung am besten eignet. Erreicht wird das durch ein Ranking welches am Ende der Laufzeit jedes Algorithmus über das zugewiesen

[1] Bubble sort(Type: Exchange sort) — Quicksort(Type: Exchange sort) — Heapsort(Type: Selection sort) — Insertion sort(Type: Insertion sort) — Merge sort(Type: Merge sort) —

Balken-Diagramme gelegt wird. Auf der Fläche werden der Rang, sprich wie lange der Algorithmus im vergleich zu den anderen gebraucht hat, eine Diagramm bei dem der Grad der „Sortiertheit“ über die Zeit dargestellt wird, sowie die Anzahl durchgeführter Operationen zu finden sein.

Kapitel 2

Fachkonzept

2.1 Begriffs- und Notationsklärung

2.1.1 Big-O-Notation

$$O(n)$$

Diese Notation dient der Angabe von Effizienz von Algorithmen man könnte auch sagen das damit die maximale Anzahl Schritte im schlechtesten,durschnitts,besten Fall oder auch Worst/Average/Best-Case-Scenario angeben wird.
Also z.b.: $O(n^2)$ bedeute das der Algorithmus bei einem Array von 4 Integern maximal 16 Schritte benötigt bis das Array sortiert ist - oder eben weniger da $4^2 = 16$. Wir werden von dieser Notation im folgenenden ein paar mal gebrauch machen um die Algorithmen zu beschreiben. Mathematisch korrekter wäre aber $O(f(n^2))$ da dem n noch ein konstanter Faktor, den Algorithmus betreffend, vorsteht.

2.1.2 Die Applikation oder App

Mit der „Applikation“ oder auch „App“ ist im folgenden die von mir, Kai Richard König, entwickelte Java-Desktop-Application zur visualisierung von Sortieralgorithmen gemeint.

2.1.3 Algorithmus

> *" Ein Algorithmus ist eine aus endlich vielen Schritten bestehende eindeutige Handlungsvorschrift zur Lösung eines Problems oder einer Klasse von Problemen. "()*

2.2 Visualisierungen von Algorithmen

Ohne jetzt eine Beweis zu führen, lässt sich behaupten das der Mensch und sein Gehirn in der Lage sind Informationen in visueller Form wesentlich schneller zu absorbieren als in geschriebener Form. Darum macht es auch durchaus Sinn die Abläufe eines Sortieralgorithmus zu visualisieren anstatt den Vorgang in textueller Form oder gar in Programmcode zu beschreiben. Außerdem sind Algorithmen ein wichtiger Bestandteil von Informatik-Studiengängen und auch deshalb es ist nicht zu verachten wenn Information durch soviele Kanäle transportiert werden wie möglich - einer davon könnte eben auch die Vorgangsvisualisierung sein. Um nun wieder auf den eigentlich Inhalt dieser Seminararbeit zurück zuschwenken, möchte ich in den folgenden Abschnitte auf die einzelen in der Applikation vorkommenden Algorithmen eingehen, sie vorstellen und vergleichen.

2.3 Verwendete Algorithmen

2.3.1 Bubblesort

Der Bubblesort Algorithmus ist der einfachste unter den Sortieralgorithmen und ist dementsprechend langsam. Ganz einfach weil der Algorithmus einen ziemlich navien Ansatz wählt. Jedes Element eines Arrays wird mit dem nachfolgenden verglichen, ist es größer so tauschen die beiden Element den Platz. Dieser Vorgang wird dann solange wiederholt bis keines der Element mehr den Platz wechselt. Diese Herrangehensweise impliziert quasi das die Prüfung, ob das Array sortiert ist, schon eingebaut ist. Ebenso ergibt diese Handlungsvorschrift eine „avarage case performance“ von $O(n^2)$. Würde Bubblesort nun versuchen ein schon sortiertes Array zu sortieren so bräuchte der Algorithmus genau die Anzahl Schritte wie das Array groß wäre und daraus würde sich dann eine „best case performance“ von $O(n)$ ergeben. (vgl. H. W. Lang, *Sortierverfahren Bubblesort*)

2.3.2 Quicksort

Quicksort ist, wie der Name schon sagt, ein sehr schneller Algorithmus mit einer einer Average-Case-Performance von $O(n \cdot log(n))$. Die Funktionsweise und der Aufbau ist anders als gedacht eher simpel. Die gute Performance wird durch eine Verfahren erreicht das sich „Teile und herrsche“ (lat. Divide et impera!, engl. Divide and conquer) nennt, dabei wird das Problem in kleinere Probleme unterteilt, diese werden dann gelöst und am Ende wieder zusammengeführt. Konkret wählt der Algorithmus zu erst ein Orientierungselement aus, in den meisten fällen bietet sich dazu das Element in der Mitte an. Nun werden alle Element die kleiner sind als das Orientierungselement nach links und jene die größer sind nach rechts verschoben. Anschließend wird der Vorgang mit den Unterprobleme, also der linken und der rechten Seite, solange wiederholt, sprich der Algorithmus wird rekursiv aufgerufen, bis die länge der auftretenden Unterprobleme zwei erreicht hat. In der Praxis erweist sich ein gut implentierter Quicksort-Algorithmus als das schnelltste Sortierverfahren. (vgl. H. W. Lang, *Sortierverfahren Quicksort*)

2.3.3 Heapsort

Der Heapsort hießt deswegen so weil er ein Problem welches zuerst in einen Heap überführt wird sortieren kann. Heap bedeutet im Englischen „der haufen“ oder auch „die Menge“ aber eigenlich ist damit ein Binärbaum (engl. Binary Tree, Binary Heap) gemeint. Ein Binärbaum ist ein Datenstruktur bei der es sogenannte Konten (engl. leafs, nodes) gibt, die maximal zwei Unterknoten haben können. Diese Struktur bringt Eigenschaften mit sich die dieser Algorithmus für sich nutzt. Der Binary Heap wird von links nach recht und von oben nach unten gefüllt, das heißt, erst wird einen sogenannte Wurzel (engl. root node) angelegt die Wurzel bekommt dann zwei Nodes und so weiter bis das Problem komplett im Baum präsent ist. Zugleich wird ein Regelwerk umgesetzt das besagt das ein Knoten keinen Unterknoten haben darf dessen Wert größer ist als der eigene. Folglich steht an der Wurzel des Baumes immer der größte Werte. Entnimmt mann diesen und schreibt ihn ans Ende einer neue Datenstruktur oder Arrays, so wird der Baum nach dem beschrieben Regelwerk neugeordnent und demgemäß steht wieder das größte Element an der Wurzel welches wieder entnommen werden kann. Diesen Vorgang wird wiederholt bis der Binary Heap abgebaut ist. Im ganzen ergibt sich daraus eine „avarage case performance“ von $O(n \cdot log(n))$. (vgl. H. W. Lang, *Sortierverfahren Heapsort*)

2.3.4 Insertionsort

Beim Insertionsort (englisch insertion ‚Einfügen‘ und englisch sort ‚sortieren‘) handelt es sich um einen relativ simplen Algorithmus. Das Prozedere in welchem das Problem

sortiert wird ist vergleichbar mit dem des Bubblesort. Ein wird Element betrachtet, ist es größer als sein nächster Nachbar so werden die Element getauscht. Das Element was nun nach hinten verschoben wurde, wird solange weiter nach hinten verschoben bis es nicht mehr größer ist als das nächst hintere. Dieser Vorgang wird für alle Elemente beim ersten anfangend bis zum letzten genau einmal durch geführt. Daraus lässt sich erkennen das dass Insertionsort-Methodik erstens für eher kleinere Problem geeignet ist und zweitens dennoch schneller als der Bubblesort-Algorithmus ist, da, wenn bei dem Letzten Elemente angekommen, das Problem bereits gelöst ist. Insertionsort ist mit einer „avarage case performance“ von $O(n^2)$ angegeben.(vgl. H. W. Lang, *Sortierverfahren Insertionsort*)

2.3.5 Mergesort

Ähnlich wie der Quicksort-Algorithmus bedient sich auch der Mergesort-Algorithmus an dem übergeordnetem „Teile und herrsche“-Prinzip. Dabei fällt, anders als beim Quicksort, die meiste Arbeit auf wieder Zusammenführen aus. Was sich auch deutlích im Source-Code wiederspiegelt. Denn ist das Problem ersteinmal rekursiv in die kleinst mögliche Teilmenge oder auch elementar Menge zerlegt, wie beim Mergesort üblich, bricht die rekursion ab und die Teillösungen werden wieder zusammen geführt (B. Voecking, *Taschenbuch der Algorithmen*, S.24), indes werden die Elemente der Teilmengen mit einander verglichen und an die entsprechende Stelle geschrieben.(vgl. H. W. Lang, *Sortierverfahren Insertionsort*)

2.4 Entwurf der Oberfläche

Die Oberfläche, im folgenden GUI genannt, sollte zunächst so simple wie möglich sein und den Betrachter auf das wesentliche lenken, die Balkendiagramme in der mitte der Applikation. Oberhalb sowie unterhalb der Balkendiagramme, jeweils abgetrennt durch einen feinen Strich finden sich die wesentlichen Steuerelement wieder.

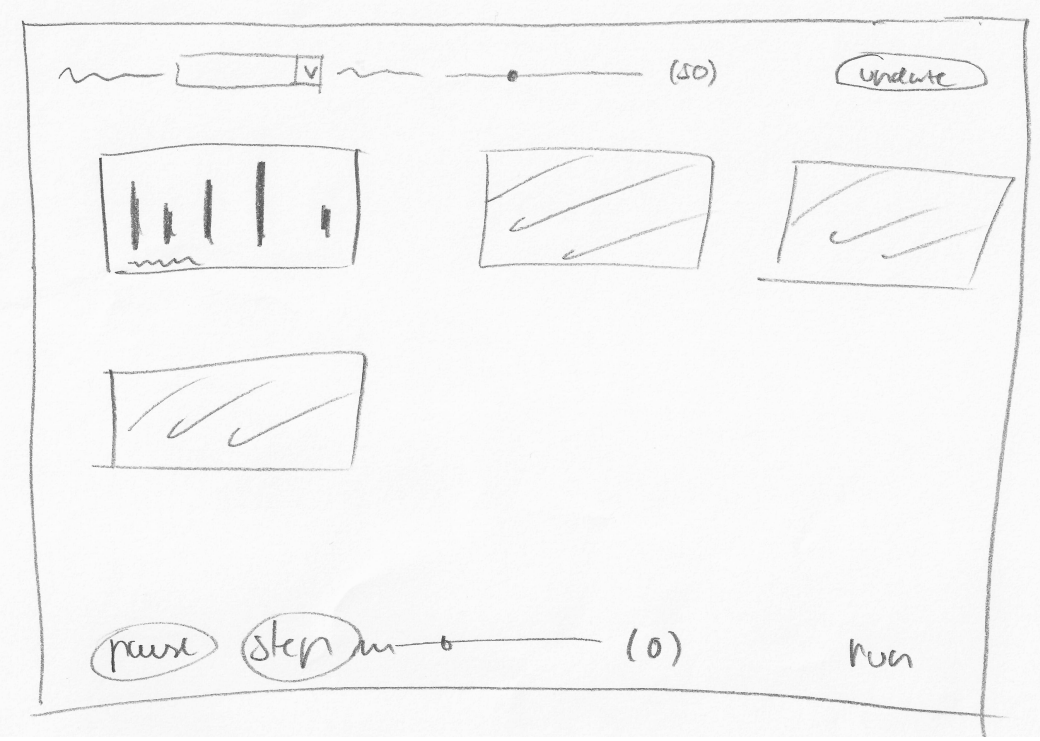

Abbildung 2.1: Eintwurf auf einem Papier, im Jargon auch Scribble genannt

Oberhalb definiert man, zum Einstieg, per Drop-Down-Menue die Art des Problems, mögliche Auswahlmöglichkeiten sind: „Random“ - für eine zufällige Zahlenreihe, „Reverse“ - für eine absteigend sortierte Zahlenreihe, „Stairs“ - für eine stufenartige Zahlenreihe mit wenig einzigartigen Element und zuletzt „predefined“ für eine immer gleichbleibende Zahlenreihe. Mit der Auswahl von „predefined“ ist sichergestellt das die Auswertungen vergleichbar bleiben.

Rechts daneben hält sich eine Slider auf mit welchem man die größe des Problems definiert, die einstellbaren Werte reichen von 10 bis 50 und lassen sich in Zehnerschritte verändern. Noch weiter rechts ist ein Button platziert mit dem die Einstellungen die mann getätigt hat auf die Balkendiagramme überträgt.

Darunter befinden sich die die Balkendiagramme, die je nach eingestellter Problemgröße unterschiedlich breite Balken ausweisen. Unterhalb der Balkendiagramme steht die Legende die die verschiedene Operationstypen mit den korrespondierenden Farben gegenüberstellt. In dem letzten sechtel des Fensters befinden sich weiter Steuerelemente, auf der linken Seite ein Button für das Pausieren der laufenden Sortierungen, wenn pausiert kann mann den Button daneben benutzen um manuell jeden noch nicht sortierten Algorithmus eine einzelene Operation durchführen zulassen. Weiter rechte ist wieder ein Slider anzutreffen mit dem mann das Interval einstellen kann mit die Algorithmen angewiesen werden einen Schritt weiter zu sortieren. Auf der rechten Seite befinden sich der Button mit der Aufschrift „sort“ welcher nach den getroffenen „Voreinstellungen“ betätigt werden kann, damit alle Algorithmen anfangen ihr Problem zulösen.

Der Untererand der App beinhaltet desweitern noch eine Statusleiste die aufschluss über die Statusse der Application selbst geben soll. Wie in Abb.2.1 angedeutet und im obrigen beschrieben kann mann in der folgenden Abb. 2.2 sehen wie der Entwurf umgestetz wurde.

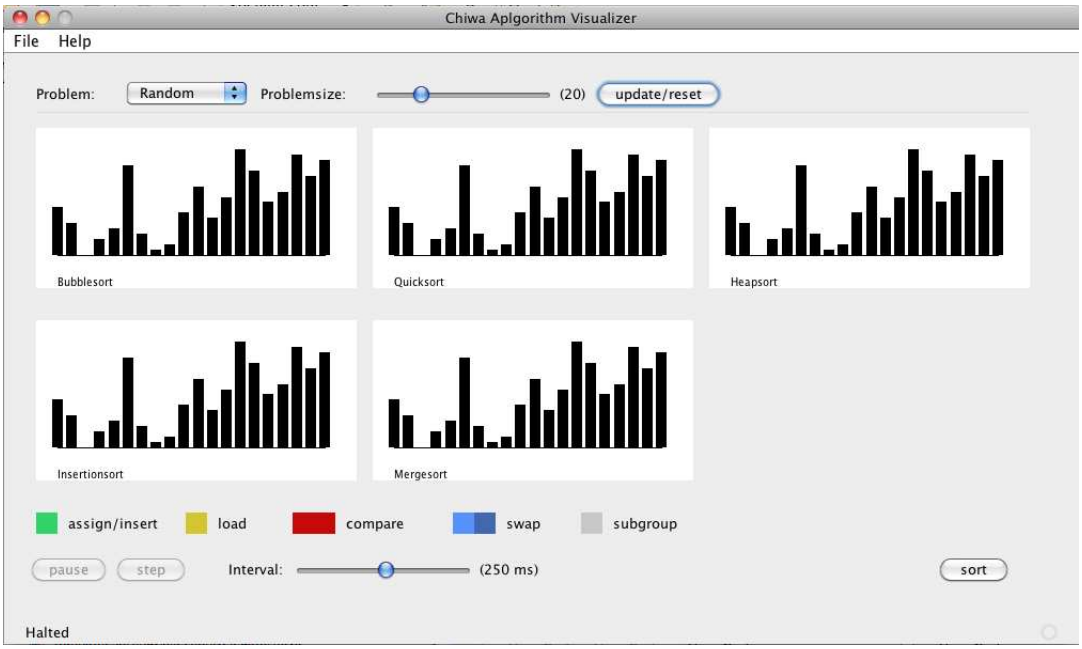

Abbildung 2.2: Java-Application nach dem Start

2.5 Programmablauf

Startet mann die Applikation werden als erstes die Standardeinstellungen geladen, Problem „Random“, Problemsize 10, Interval 250 Millisekunden. Anschließende kann mann entsprechende Einstellungen nach eigenem ermessen verändern. Der Button „pause“ und „step“ sind inaktiv gestellt, sprich sie lassen sich nicht betätigen. Nur der Button „sort“ ist aktiv. Klickt mann nun auf „sort“ so kann mann beobachten wie die Balken mit verschieden Farben hervorgehoben werden und sich das Balkendiagramm langsam, oder auch schnell je nach Interval sortiert. Nun hat man die möglichkeit „pause“ zudrücken und das sortieren anzuhalten um mit dem Button „step“ schrittweise weiter zusortiern. Währendessen steht im Pause-Button „resume“ was, wenn gedrückt den Sortiervorgang wieder automatischen weiterlaufen lässt. Ist das geschehn steht im Pause-Button auch wieder „pause“. Außerdem ist während des ganz Sortiervorgangs der Button „sort“ deaktiviert.

Ist ein Balkendiagramm fertig sotiert, also der Algorithmus durchgelaufen, so wird ein Linendiagramm über das Balkendiagramm gelegt welches die Position des Algorithmus im vergleich zu den anderen beinhaltet sowie die Anzahl durchgeführter Operation sowie die Sortiertheit auf der Y-Achse und die Anzahl Operation auf der X-Achse anzeigt.

An dem Punkt angekommen kann der Anwender nun entwerder noch einmal auf „sort“ drücken was dazu führt das die bereits sortierten Balken nocheinmal sortiert werden oder er kann den oben beschrieben Vorgang wiederholen und andere Einstellungen wählen. Falsche Eingaben kann der Nutzer dabei nicht treffen, alle Einstellungskombinationen sind denkbar und durchführbar. Einzig und allein kann ein zu kurze gewähltes Interval auf langsamen Computer dazu führen das der Sortiervorgang nicht mehr flüßig aussieht.

Kapitel 3

IT-Konzept

3.1 Logische Einheiten

In den nächsten Abschnitten möchte ich auf die verschieden Logischen Einheiten eingehen. Logische Einheiten sind Komponenten oder auch Klassen die ein klar diffenrzierbare Aufgabe innerhalb der Applikation haben.

3.1.1 App, main() und View

Um es gleich vorweg zuschicken, die Klasse App.java ist nicht von mir sondern bei der erstellung eine neuen Projekts von Netbeans[1] erstellt worden. Die Klasse enthält die main()-Methode welche zum starten einer jeden Java-Applikation benötigt wird. In der Klasse App findet mann eine Methode welche die Klasse View.java instanziert.

```
/**
 * At startup create and show the main frame of the application.
 */
@Override protected void startup() {
    show(new View(this));
}
```

Listing 1: Methode startup (App.java Z.18-20)

[1] NetBeans IDE 7.0 (Build 201104080000)

3.1.1.1 View.java

In dieser Klasse findet man alle Methoden wieder die zur generierung der einzelen Steuerelemente dienen. Ferner werden alle Balkendiagramme und deren zugehörige Algorithmen instanziert, hinzukommen alle Eventhandler mit denen die Anwendereingabe an die richtige Stelle delegiert wird.

3.1.2 RunnableSortingCollection

RunnableSortingCollection[2] - wie der name schon andeutet beinhaltet eine Instanz dieser Klasse alle Sortieralgorithmen, sofern programmatisch hinzugefügt, und delegiert eventuelle Steuerbefehle, wie zum Beispiel das Starten des Sortiervorgangs, an all jene Sortieralgorithmen. Mann könnte auch sagen das die Klasse der Kommunikation zwischen Nutzeroberfläche oder auch Nutzer und den Algorithmen dient.

3.1.3 SynchronizedSorter

Wie zu Anfang beschrieben braucht es Logik um zu gewährleisten das der Ablauf des Sortierens paralell und synchron über den ganzen „existenz Zeitraum“ der Applikation bleibt. Erreicht wird dieses Verhalten durch die Nutzung von Nativ-Threads und der tatsache das es seit Java 5[3] möglich ist das ein Thread eine anderen sogenannte Messages senden kann.

3.1.4 VisualFeedbackSorter

Eine weiter wichtige, wenn nicht sogar die wichtigeste, Komponenten der Applikation ist der „VisualFeedbackSorter“ welche für die visuelle repränstation der internen Sortiervogänge zuständigt ist.

[2] „There are only two hard things in Computer Science: cache invalidation and naming things“ - Phil Karlton

[3] siehe http://download.oracle.com/javase/tutorial/essential/concurrency/

3.1.5 Die Klasse AbstractSortingMechanics und das Interface Sorter

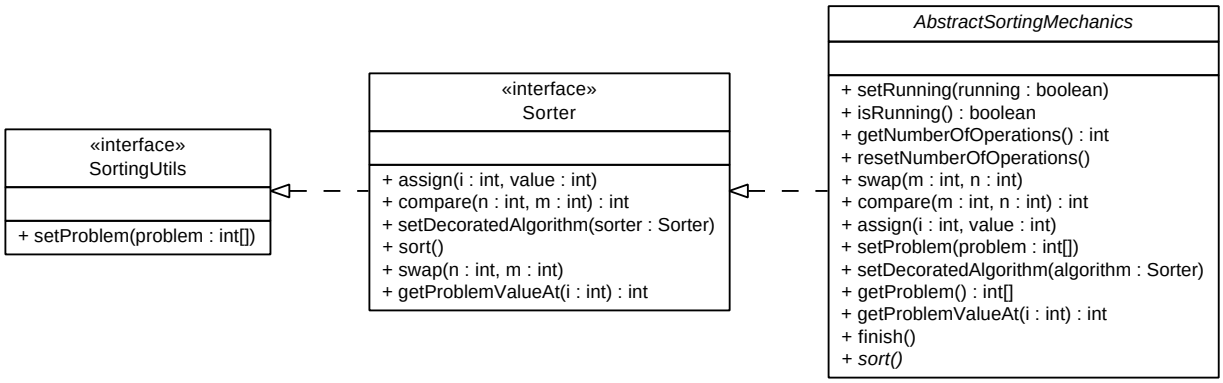

Abbildung 3.1: Klassenkarte von AbstractSortingMechanics.java und Sorter.java

Das Interface Sorter und die Klasse AbstractSortingMechanics bilden die Basis für den 3.2.1 beschrieben Decorator-Pattern aber prinzipell bilden diese Struckturen die Basis für alle Klassen die sich wie ein Algorithmus verhalten sollen.

(vgl. Abb. 3.3)

3.2 Erläuterung wichtiger Entwurfsmuster

In meiner Ausbildung zum Mediengestalter und in der Zeit als frei Programmiere, habe ich mich immer wieder verschiedener Entwurfsmuster bedient, denn sie benennen, abstrahieren und indentifiziern die Kernaspekte einer wiederkehrenden herrangehensweise und schaffen somit wiederverwendbare Objekte-Orientierte Lösungen.(Gamma u. a., *Design patterns: elements of reusable object-oriented software*, S.3)

3.2.1 Decorator-Pattern

> *“ Mit dem Decorator Pattern, in der deutschen Übersetzung naheliegenderweise Dekorierer genannt, lässt sich ein Objekt dynamisch um Fähigkeiten, auch Zuständigkeiten genannt, erweitern. Anstatt Unterklassen zu bilden und eine Klasse damit um Fähigkeiten bzw. Verhalten zu erweitern, lässt sich mit dem Einsatz des Decorator Patterns die Erzeugung von Unterklassen vermeiden ”(Sherzad,* Decorator Pattern in Java*)*

(vgl. S.175-184 Gamma u. a., *Design patterns: elements of reusable object-oriented software*)

3.2.1.1 AbstractSortingDecorator

Diese Klasse bildet die Grundlage für den Decorator-Pattern und wird an VisualFeedbackSorter und SynchronizedSorter weitervererbt. Instanziert mann einen dieser Dekoratoren und übergibts ihm das zu dekorierende Objekt so verhält sich dieses nach außen hin exakt genauso wie das übergeben Objekt, genau darin liegt die Stärke diese Entwurfsmuster welches sich im Konstrucktur von View.app zeigt.

```
Sorter bubblesort    = new SynchronizedSorter(new VisualFeedbackSorter(new Bubblesort()));
Sorter quicksort     = new SynchronizedSorter(new VisualFeedbackSorter(new Quicksort()));
Sorter heapsort      = new SynchronizedSorter(new VisualFeedbackSorter(new Heapsort()));
Sorter insertionsort = new SynchronizedSorter(new VisualFeedbackSorter(new Insertionsort()));
Sorter mergesort     = new SynchronizedSorter(new VisualFeedbackSorter(new Mergesort()));
```

Listing 2: Instanzierung und dekorierung aller Algorithmen(View.java Z.69-73)

Nachwievor verhalten sich alle Algorithmen nach außen wie ein Sorter, nur sind jetzt alle Methoden dekoriert, das heißt würde mann auf `bubblesort` nun eine Method aufrufen so würde sie von `SynchronizedSorter` über `VisualFeedbackSorter` bis `Bubblesort` durch gereicht werden. Zugleich kann jeder dieser Dekorierer auch eigenen Code ausführen. Das entscheidende Merkmal ist das `SynchronizedSorter` und auch `VisualFeedbackSorter` in den Konstrucktoren den übergebenem Parameter in einem Feld speichen.

```
public abstract class AbstractSortingDecorator
  extends AbstractSortingMechanics {
    protected final Sorter algorithm;

    public AbstractSortingDecorator(Sorter algorithm) {
        this.algorithm = algorithm;
    }
}
```

Listing 3: Konstrucktor (AbstractSortingDecorator.java Z.11-29.)

Und zweitens das die erbenden Klassen diese Feld zur „weitereichung“ benutzen. (siehe Listing 4 Z.18)

```java
public class SynchronizedSorter extends AbstractSortingDecorator {

    public SynchronizedSorter(Sorter algorithm) {
        super(algorithm);
    }

    @Override
    public void swap(int m, int n) {
        synchronized (super.algorithm) {
            try {
                super.algorithm.wait();
            } catch (InterruptedException ex) {
                Logger.getLogger(SynchronizedSorter.class.getName())
            .log(Level.SEVERE, null, ex);
            }
        }
        algorithm.swap(m, n);
    }
```

Listing 4: Beispiel einer Dekoration (SynchronizedSorter.java Z.20-36.)

3.3 Erläuterung besondere Implementationen

3.3.1 Die Klasse „SynchronizedSorter“

Wie in Listing 4 bereits gezeigt erbt die Klasse von `AbstractSortingDecorator` und dieser wiederum von `AbstractSortingMechanics`. Die Methoden `swap`,`compare`,`assign`,`getProblemValueAt` werden herbei wie schon in 4 gezeigt mit einem `Try-and-Catch-Block` in `SynchronizedSorter` implementiert. Das hat zurfolge, immer wenn eine dieser Methoden aufgerufen werden der laufende Thread ersteinmal wartet (4 Z.11) bis irgendwo auf dem selben Object ein `notify()` aufgerufen wird. Und schon hat man die Ausführung synchronisiert.

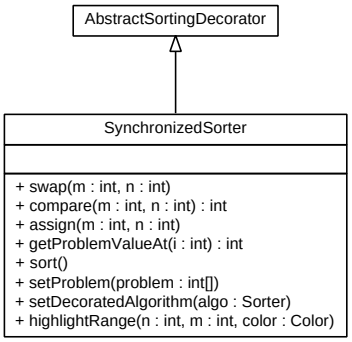

Abbildung 3.2: Klassenkarte

3.3.1.1 Synchronisierte Threads

Bei der beschrieben Applikation gibts es insgesamt für jeden Algorithmus einen Thread sowie eine Main-Thread, in dem der MainFrame läuft, einen Timer-Thread und noch eine Reihe Threads aus dem Swing-Threadpool auf die ich hier aber nicht eingehen werde.

Stellt mann sich nun der einfachheit habler vor alle Algorithmen wollen eine `swap` Operation durchführen so wird diese Methode aufgerufen und landet schluss endlich irgendwann in dem gezeigt Sourecode Beispiel aus 4 und muss dort nun verharren, bis und jetzt kommt der springende Punkt, in dem Timer-Thread der Java Timer über alle Algorithmen in `RunnableSortingCollection` enthalten sind iteriert und ein `notify()` über das Algorithmus-Object senden auf welches die Algorithmen in `swap` warten.

Das Interval mit dem die Timer Klasse das tut kann der Anwender einstellen, von 500ms bis 20ms. Oder auch kann der Anwender den Timer anhalten und die Applikation manuell anweisen ein `notify()` an alle Algorithmen zusenden.

3.3.2 Die Klasse „VisualFeedbackSorter“

Diese Klasse baut zum Start der Applikation, mit hilf der Charting-Bibliothek jChart2D, das Balkendiagramm für den übergebenen Algorithmus auf und speicher die Instanz in einem Feld. Tritt nun der Fall wie in 3.3.1.1 auf so wird das Chart angewießen die entsprechenden Balken mit einer Farbe zuhinterlegen.

3.3.3 Die Klasse `Surveyor`

Am Rande zuerwähnen aber dennoch interessant ist der Surveyor welcher, wenn ein Integer Array übergeben bekommt, die Sortiertheit im Rahmen von 0.0 bis 1.0 wiedergeben kann. Dabei wird die distanz eine Elements zu seiner eigentlichen Position bestimmt, diese Entfernungen werden aufsummiert und durch die maximale möglich gesamt Entfernung geteilt. Bei einer länge von n ergibt das eine maximal Entfernung von n^2. Die Methodik wird für die Auswertung am Ende eines jeden Sortiervorgangs benutzt.

3.4 Klassendiagramm

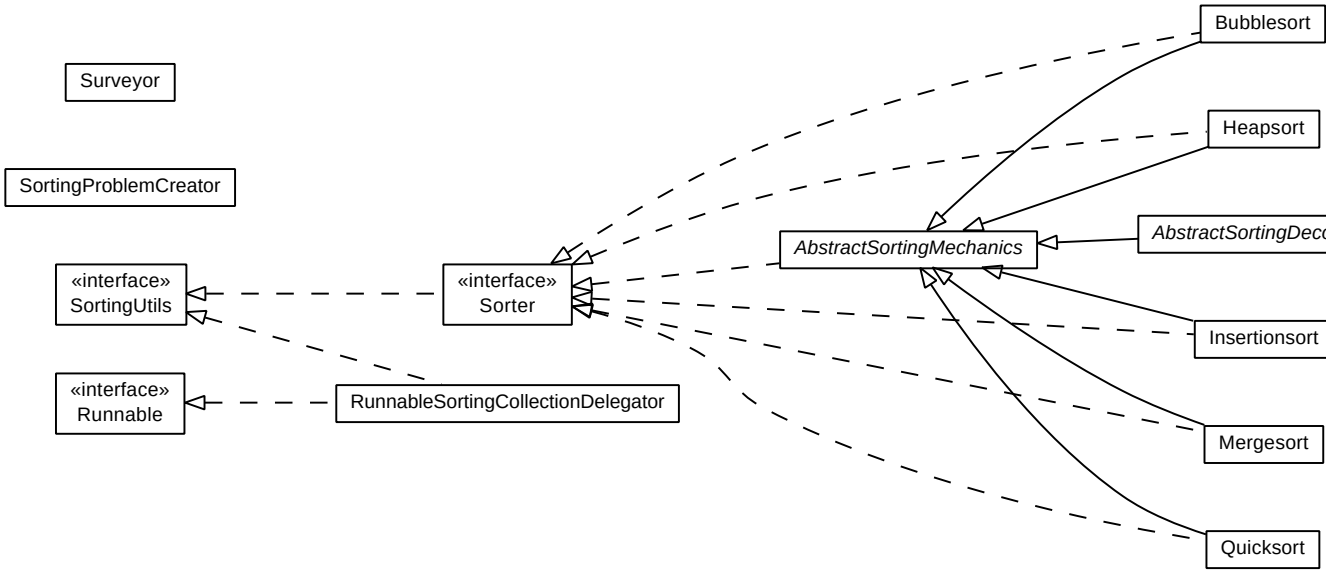

Abbildung 3.3: Klassendiagramm aller wichtigen Komponenten

Literatur

Bubblesort None H. W. Lang. *Sortierverfahren Bubblesort*. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/networks/bubble.htm.

decoratorpattern Rias A. Sherzad. *Decorator Pattern in Java*. URL: http://www.theserverside.de/decorator-pattern-in-java/.

designpatterns Erich Gamma u. a. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

heapsort None H. W. Lang. *Sortierverfahren Heapsort*. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/heap/heap.htm.

insertionsort None H. W. Lang. *Sortierverfahren Insertionsort*. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/merge/merge.htm.

mergesort None H. W. Lang. *Sortierverfahren Insertionsort*. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/insert/insertion.htm.

quicksort None H. W. Lang. *Sortierverfahren Quicksort*. URL: http://www.inf.fh-flensburg.de/lang/algorithmen/sortieren/quick/quick.htm.

taschenbuch H. Alt B. Voecking. *Taschenbuch der Algorithmen*. Springer, 2008.

wiki¨algo URL: http://de.wikipedia.org/wiki/Algorithmus.

wiki¨quicksort URL: http://de.wikipedia.org/wiki/Quicksort.