ETH Zürich

Semesterarbeit

Wintersemester 2001/2002

Animierte Datenstrukturen in Java

Schlussbericht

5. Juli 2002

Peter Häfliger

Student der
Rechnergestützten Wissenschaften

peterhaefliger@yahoo.com

Institut für Theoretische Informatik
Gruppe Prof. Peter Widmayer

Betreuer: Mark Cieliebak

cielieba@inf.ethz.ch

Einleitende Zusammenfassung

Um Algorithmen oder Datenstrukturen zu erklären, werden häufig Animationen als Hilfsmittel verwendet. Es gibt heute bereits eine Unzahl von Animationen für verschiedene Datenstrukturen und Algorithmen. Diese sind jedoch grösstenteils "handgestrickt", d.h. zu jedem Algorithmus wurde eine individuelle Animation erzeugt. ([Cieliebak01])

Mit dem Durchbruch von **objektorientierten Programmiersprachen** verlagerte sich das Schwergewicht des Software-Entwurfs von den Algorithmen zu den Datenstrukturen ([Meyer97], Kapitel 5.1: 'The Ingredients of Computation'). In der im vorliegenden Dokument beschriebenen Semesterarbeit wurde die Möglichkeit untersucht, eine Bibliothek von **Datenstrukturen** zu schaffen, **die bereits Animationen für Grundoperationen zur Verfügung stellen** und die **halbautomatische Animation bestehender Algorithmen** erlauben. Es wurde ein **Prototyp eines Frameworks** für animierte Datenstrukturen in **Java** erstellt. Innerhalb dieses Frameworks wurden animierte Versionen von zwei Datenstrukturen implementiert: von **ArrayList**, einem Standardcontainer aus der Java API, und von einem selbst definierten **Heap**. Anhand dieses Prototyps können die Möglichkeiten und Grenzen dieses alternativen Animationsansatzes aufgezeigt werden.

Zweck des Dokuments

Das vorliegende Dokument beschreibt die im Wintersemester 2001/2002 geleistete Semesterarbeit. Es ergänzt und dokumentiert das entstandene Softwareframework und gliedert sich in vier Hauptteile, von denen jeder auf dem vorhergehenden aufbaut:

- **Benutzerdokumentation**

  Die Benutzerdokumentation beschreibt die Funktionalität der entwickelten Software.
  Sie richtet sich an die Benutzer des Animationstools (Endbenutzer).

- **Entwicklerhandbuch**

  Das Entwicklerhandbuch beschreibt den Aufbau und die Erweiterbarkeit der Software.
  Es richtet sich an die Benutzer des Frameworks (Entwickler).

- **Rechenschaftsbericht**

  Der Rechenschaftsbericht beschreibt den Projektablauf der Semesterarbeit.
  Er richtet sich an die zuständigen Stellen der ETH (betreuendes Institut, RW-Ausschuss).

- **Anhang**

  Der Anhang listet alternative Arbeiten, referenzierte Literatur, verwendete Softwaretools und Internet-Links zu angesprochenen Themen auf.

Inhaltsverzeichnis

TEIL I: BENUTZERDOKUMENTATION

1. Installation

VORAUSSETZUNGEN

Das Animationstool sollte auf jeder Maschine lauffähig sein, auf der ein **Java 2 SDK** (Software Development Kit) und das zugehörige JRE (Java **Runtime Environment**) installiert sind. Beide können kostenlos vom Internet heruntergeladen werden: Siehe Anhang C für die Adresse und für die Version, die zur Entwicklung des Tools verwendet wurde. In der Installationsanleitung für das Java SDK steht auch beschrieben, wie der **Klassenpfad** auf den verschiedenen Plattformen gesetzt wird. Der Klassenpfad besteht aus einem oder mehreren Pfaden von Verzeichnissen, in denen der Classloader der JVM (Java Virtual Machine, Teil des JRE) nach aufgerufenen Klassen sucht.

Es ist zu beachten, dass die in diesem Dokument enthaltenen Screenshots mit **Windows 2000** gemacht wurden und dass die nachfolgenden Kommandozeilen -Befehle in **DOS-Syntax** angegeben sind. Für andere Plattformen müssen sie entsprechend den dort geltenden Konventionen angepasst werden (z.B. Vorwärts- statt Rückwärts-Schrägstriche als Trennzeichen zwischen Verzeichnisnamen in UNIX: your_directory/peter/sa2 statt your_directory\peter\sa2).

INSTALLATION DES TOOLS

Wenn diese Voraussetzungen erfüllt sind, kann das File **animation.jar** heruntergeladen (siehe Anhang C für die Adresse) und in einem **Verzeichnis, welches im Klassenpfad enthalten ist** , abgespeichert werden. Wenn das Java SDK richtig installiert ist, kann **animation.jar** nun von der Kommandozeile aus dekomprimiert werden:

```
> cd your_directory
> jar xf animation.jar
```

Die dekomprimierte Verzeichnisstruktur ist in Bild 1 gezeigt. **your_directory** ist dabei das im Klassenpfad enthaltene Verzeichnis. Die Unterverzeichnisse – ausser 'META-INF', 'doc' und 'images' – entsprechen den Paketen der Software.

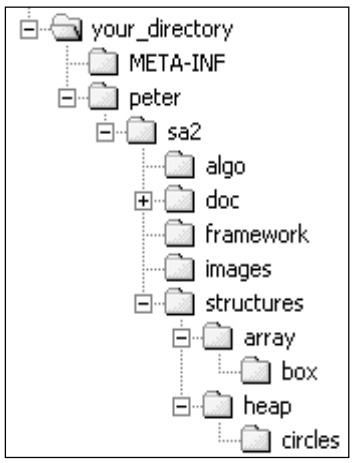

Bild 1: Verzeichnisstruktur

DOKUMENTATION

Das Unterverzeichnis your_directory \peter\sa2\doc enthält die mit **javadoc** generierte Dokumentation aller Klassen. Weil diese Dokumentation v.a. vom Entwickler gebraucht wird, der neue animierte Datenstrukturen implementiert, und nicht vom Anwender, der seine Algorithmen mit bestehenden Datenstrukturen animieren will, enthält sie die Dokumentation aller Klassen, Attribute und Methoden, auch der privaten. Der Code einer abgeleiteten Klasse ist so eng mit dem Code der Basisklasse verwoben, dass der Entwickler einer abgeleiteten Klasse die Implementation der Basisklasse genau studieren sollte, um auch ihr internes Funktionieren vollständig zu verstehen: Zwischen einer abgeleiteten Klasse und ihrer Basisklasse gibt es

keine Geheimhaltung ([Meyer97], Kapitel 16.8: 'Inheritance and In - formation Hiding'). Die Anwendung bestehender Datenstrukturen ist hingegen so einfach, dass man dazu nur ganz wenig Dokumentation konsultieren muss, falls die Hinweise in den nächsten beiden Ka - piteln nicht sogar bereits genügen. Wer sich die Dokumentation nur der öffentlich und paketweit zugänglichen Klassen, Attribute und Methoden trotzdem generieren will, kann das jederzeit selbst tun:

Bild 2: javadoc

```
> cd your_directory
> javadoc -d your_output_directory peter.sa2.algo
        peter.sa2.framework
        peter.sa2.structures peter.sa2.structures.array
        peter.sa2.structures.array.box
        peter.sa2.structures.heap peter.sa2.structures.heap.circles
```

Die mitgelieferte Dokume ntation wurde demgegenüber mit der Option –private generiert, um auch private Klassen, Attribute und Methoden einzubeziehen:

```
> cd your_directory
> javadoc -private -d your_output_directory peter.sa2.algo peter.sa2.framework
        peter.sa2.structures peter.sa2.structures.array peter.sa2.structures.array.box
        peter.sa2.structures.heap peter.sa2.structures.heap.circles
```

Zum Durchstöbern der Dokumentation bietet your_directory \peter\sa2\doc\help-doc.html einen guten Einstiegspunkt (siehe Bild 2).

2. Demoprogramme

Im Paket **peter.sa2.algo** befindet sich eine Reihe von Beispielalgorithmen:

- SelectionSort
- PointedSelectionSort
- InsertionSort
- BubbleSort
- HeapSort
- HeapDemo
- HeapSortRevisited

Der Sourcecode kann in einem beliebigen Text-Editor angeschaut werden (.java-Files mit den gleichen Namen wie die obigen Algorithmen im Verzeichnis your_directory\peter\sa2\algo).

Die Beispielprogramme können von der Kommandozeile aus gestartet werden, z.B.:

```
> cd beliebiges_verzeichnis
> java peter.sa2.algo.SelectionSort
```

Nach dem Programmstart erscheint das in Bild 3 gezeigte Fenster. Sein wichtigster Bestand - teil ist das mit 'Animation' bezeichnete Graphikpanel, in welchem die Animation der Daten - struktur stattfindet, auf der der Algorithmus arbeitet. Darunter befindet sich das mit 'Console'

bezeichnete Textpanel, in welchem ein Protokoll der Operationen ausgegeben wird, welche auf der Datenstruktur ausgeführt werden. Die Leiste, welche die beiden Panels trennt, kann vertikal verschoben werden, um je nach Wunsch dem einen oder andere n Panel mehr Platz einzuräumen. In beiden Panels kann nur im Pausenmodus gescrollt werden. Während der Al - gorithmus abläuft, ist der Focus in der 'Console' immer auf der untersten (neusten) Zeile, im 'Animation'-Panel immer so, dass (sofern das Fenster nic ht allzu stark verkleinert wurde) die - jenigen Elemente einer Datenstruktur sichtbar sind, die momentan gerade modifiziert werden.

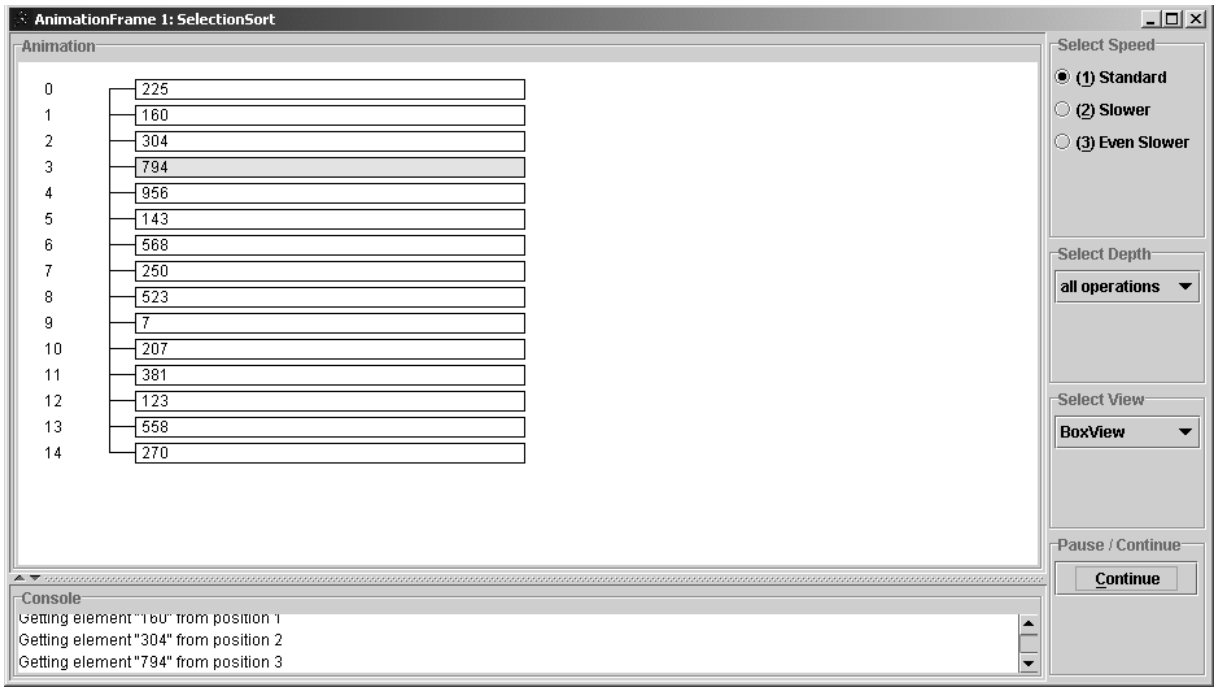

Bild 3: Graphische Benutzerschnittstelle des Animationstools

Auf der rechten Seite befindet sich ein minimales Set von Steuerelementen für die Interaktion mit dem Benutzer: Zuoberst befinden sich Radioknöpfe für die Wahl der Animationsge - schwindigkeit. Darunter befinden sich die Selektoren für Animationstiefe und Ansicht (sog. ComboBoxes, welche nach dem Anklicken eine Auswahl präsentieren). **AnimatedArrayList**, die Datenstruktur, auf welcher die Beispiel-Sortieralgorithmen ablaufen, besitzt zwei Tiefen:

- **all operations**

  In dieser Tiefe werden alle Operationen animiert, welche **AnimatedArrayList** zur Verfügung stellt: Elemente, die gerade gelesen werden, in gelb (w ie in Bild 3 gezeigt); Elemente, die hinzugefügt werden, in grün; Elemente, die entfernt werden, in rot.

- **assignments**

  In dieser Tiefe werden nur Operationen animiert, welche die Datenstruktur verändern (Hinzufügen und Entfernen). Lesezugriffe werden ausge führt und protokolliert, aber nicht animiert.

AnimatedArrayList besitzt im Moment nur eine Ansicht: **BoxView** (Darstellung der Elemente als Kästchen, wie in Bild 3 gezeigt). Für Datenstrukturen, welche mehr als eine Ansicht anbieten (**AnimatedHeap** bietet z.B . zwei verschiedene Elementgrössen an, siehe

HeapDemo-Beispielprogramm), kann die Ansicht mit dem zweituntersten Knopf zur Laufzeit gewechselt werden.

Werden in einem Programm mehrere Datenstrukturen gleichzeitig animiert, so geschieht dies für jede Datenstruktur in einem separaten Fenster (siehe z.B. **HeapDemo**-Beispielprogramm). Animationstiefe, Ansicht und Geschwindigkeit können dann für jede Datenstruktur individuell gewählt werden, nicht aber der Wechsel in den Pausenmodus (unterster Knopf: Pause / Continue). Das hat folgenden Grund: Wenn die Animation einer Datenstruktur in die Pause geschickt wird (z.B. um das Textprotokoll zu studieren oder einer grösseren Struktur entlangzuscrollen), dann hält auch der Algorithmus an, weil er darauf wartet, dass die Datenstruktur die gewünschte Operation ausführt. Deshalb befinden sich auch alle anderen Datenstrukturen in einer Zwangspause, während sie darauf warten, dass der Algorithmus weitere Operationen auf ihnen ausführt. Dies wird in den entsprechenden Fenstern auch in der Aufschrift auf dem Pausenknopf ('Continue' statt 'Pause') reflektiert. Somit muss die Pause auch nicht im gleichen Frame abgebrochen werden, in dem sie begonnen wurde.

Am besten macht man sich mit diesen einfachen Funktionen vertraut, indem man die Beispielprogramme ausführt und mit den Knöpfen ein bisschen herumspielt.

3. Schrittweises Vorgehen zur Animation eines bestehenden Algorithmus

An einem bestehenden Algorithmus, der eine Datenstruktur verwendet, von welcher im Framework ein animierter Untertyp implementiert ist (z.B. **AnimatedArrayList**, abgeleitet von **ArrayList**), sind im besten Fall zur Animation nur folgende zwei Modifikationen nötig:

1. Einfügen der Zeile

```
import peter.sa2.structures.array.AnimatedArrayList;
```

2. und ersetzen der Zeile

```
ArrayList your_array_name = new ArrayList();
```

durch

```
ArrayList your_array_name = new AnimatedArrayList();
```

für diejenige(n) Instanz(en) von **ArrayList**, die man animieren will.

Es kann gut sein, dass im Programm auch Instanzen von **ArrayList** vorhanden sind, die man nicht animieren will. Meist will man ja z.B. den Array vor der Animation irgendwie initialisieren. Dazu hat man nur die Methoden zur Verfügung, die die Datenstruktur anbietet, z.B. add(int index, **Object** element). Also muss man einen nicht animierten Array elementweise aufbauen und dem animierten Array im entsprechenden Konstruktor übergeben. In diesem Fall ist die Modifikation ein ganz klein wenig grösser, z.B.:

1. Einfügen der Zeile

```
import peter.sa2.structures.array.AnimatedArrayList;
```

2. und ersetzen der Zeile

```
ArrayList your_array_name = new ArrayList();
```

durch

```
// initialize array
int arrayLength, maxInt;
Integer randomInt;
ArrayList nonanim = new ArrayList();

// …

for (int i = 0; i < arrayLength; ++i)       {
   randomInt = new Integer((int)(Math.random()*maxInt));
   nonanim.add(i, randomInt);
}

ArrayList your_array_name = new AnimatedArrayList(nonanim);
```

für eine Instanz von **ArrayList**, die man animieren und mit natürlichen Zufallszahlen zwischen 0 und (maxInt – 1) initialisieren will.

AnimatedArrayList bietet weitere Konstruktoren an, in denen der Titel des Frames sowie seine Position und Ausdehnung spezifiziert werden können. Die Konstruktoren sind in der **javadoc**-Dokumentation beschrieben, und im Sourcecode der oben beschriebenen Beispielprogramme kann ihre Verwendung studiert werden.

4. Bildschirmkonfiguration

Der im Folgenden beschriebene Mechanismus wird nur in seltenen Fällen gebraucht, nämlich dann, wenn die Fenster, welche das Animationstool produziert, schlecht mit der Bildschirmauflösung harmonieren, also entweder viel zu klein erscheinen oder im Gegenteil so gross, dass nur ein Teil des Fensters auf dem Monitor angezeigt werden kann. Im Normalfall ist dies aber nicht der Fall und dieses Kapitel kann getrost übersprungen werden.

Die Grösse der graphischen Benutzerschnittstelle des Frameworks ist in der Klasse **AnimationFrame** hart codiert (in Pixels). Vielleicht wäre es möglich, das Programm zuerst die Grösse des Bildschirms ausmessen und dann die Grösse der Benutzerschnittstelle entsprechend berechnen zu lassen, aber diese Möglichkeit wurde nicht untersucht.

Die absolute Grösse des Fensters wurde gewählt in der Hoffnung, dass sie für die gängigen Bildschirmformate geeignet sei (nicht zu gross und nicht zu klein). Sollte diese Annahme für den Bildschirm eines Anwender nicht zutreffen, kann er diese Grösse durch Modifikation von vier Konstanten in der Klasse **AnimationFrame** im Paket **peter.sa2.framework** verändern:

- MAINWIDTH / MAINHEIGHT: absolute Grösse der 'split pane' (Einheit der beiden Panels für graphischen und Textoutput, durch Schieber getrennt). Die gesamte Benutzer - schnittstelle ist grösser durch die Steuerelemente, die rechts angehängt sind, und den Gesamtrahmen (siehe Bild 3). Eine Veränderung dieser Konstanten beeinflusst die Grösse der eingebetteten Komponenten (d.h. der beiden Panels) und der Graphiken, welche alle relativ zu diesen Konstanten definiert sind. Dabei gibt es nur ein Problem: Die Grösse von Text lässt sich nur absolut angeben, deshalb wird die Grösse der Texte in den Graphiken von diesen beiden Konstanten nicht beeinflusst. Werden also die beiden Konstanten allzu klein gewählt, haben z.B. im BoxArrayView der **AnimatedArrayList** die Beschriftungen nicht mehr in den Kästchen Platz u nd überlappen. Um diesen unschönen Effekt zu verhindern, kann das zweite Set von Konstanten Abhilfe schaffen.

- XANIMSCALING / YANIMSCALING: Die Grösse der Graphiken ist wie gesagt relativ zu MAINWIDTH und MAINHEIGHT definiert, wird aber noch mit diesem zwe iten Set von Konstanten skaliert (default: XANIMSCALING = YANIMSCALING = 1). Werden diese Konstanten vergrössert, vergrössern sich die Graphiken (der Text bleibt immer gleich), wodurch sie dann einen grösseren Teil des Grahikpanels ausfüllen und bei der Ansicht häufiger gescrollt werden muss.

Beispiel: Für einen Bildschirm mit sehr niedriger Auflösung werden MAINWIDTH und MAINHEIGHT halbiert (von 800 auf 400 bzw. von 600 auf 300 Pixels). Die Graphiken werden entsprechend verkleinert, so dass die Proportion en wieder genau die selben sind wie in Bild 3, mit Ausnahme der Texte in den Kästchen, welche zu gross sind. Zur Abhilfe werden XANIMSCALING und YANIMSCALING verdoppelt (von 1 auf 2). Dadurch erhält die Graphik wieder die ursprüngliche absolute Grösse, fül lt aber nun die ganze Breite des Graphikpanels aus, und in der Höhe werden nur noch acht bis neun Elemente gleichzeitig angezeigt.

Die durch MAINWIDTH und MAINHEIGHT definierte Grösse der Benutzerschnittstelle ist die default-Grösse. Wird z.B. für **AnimatedArrayList** ein Konstruktor gewählt, bei dem die Grösse der ‚split pane' als Argument mitgegeben werden kann, so wird diese Grösse verwen - det. Die in den Konstruktoren angegebenen Grössen wirken sich allerdings nur auf Frames, Panels und Steuerelemente aus; die Graphiken werden NICHT mitskaliert.

TEIL II: ENTWICKLERHANDBUCH

5. Überblick über das Design des Frameworks

5.1 Statischer Aufbau

Dieses Kapitel soll die Grundzüge des Designs erklären. Fürs Detailverständnis wird auf die mit javadoc aus dem Code extrahierte Dokumentation und auf den Code selbst verwiesen.

Das Paket **peter.sa2.framework** (siehe Bild 4 für die Paketeinteilung) ist ohne Modifikation für sämtliche animierten Datenstrukturen wiederverwendbar: Es stellt den Rahmen der graphischen Benutzerschnittstelle zur Verfügung und diejenigen Steuerelemente, welche für alle animierten Datenstrukturen identisch sind.

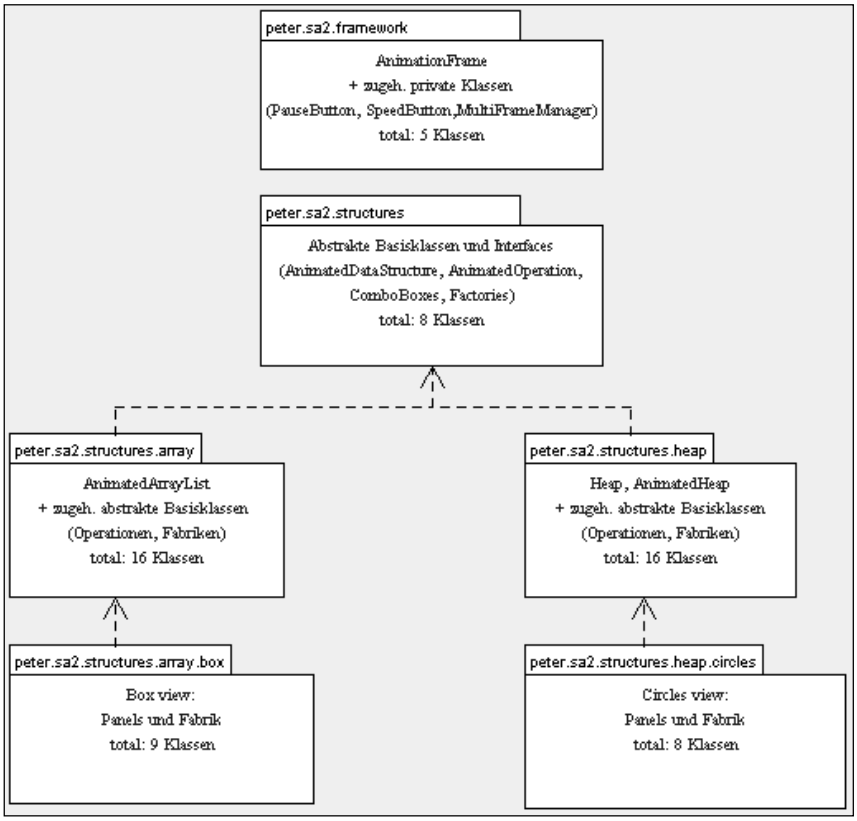

Bild 4: Paketstruktur des Framworks

Die Klasse **AnimationFrame** ist von **JFrame** abgeleitet. Sie stellt den Rahmen zur Verfügung, in den die allgemeinen und datenstrukturspezifischen Steuerelemente eingebettet sind (siehe Bild 3). Allgemeine Steuerelemente sind die RadioButtons zur Geschwindigkeitsregelung und der Pause/Continue -Button. Die entsprechenden Klassen befinden sich ebenfalls im Paket **peter.sa2.framework**. Datenstrukturspezifisch sind die beiden Panels zur Darstellung der Operationen der Datenstruktur in Bild und Text und die beiden ComboBoxes zur Wahl der Ansicht bzw. der Animationstiefe. Ihre generierenden Klassen befinden sich dementsprechend in tiefer liegenden Paketen.

Ebenfalls im Paket **peter.sa2.framework** befindet sich die Klasse **MultiFrameManager**. Von ihr gibt es zur Laufzeit jeweils nur eine einzige Instanz, welche in einem statischen Codeblock in der Klasse **AnimationFrame** erzeugt wird. Ein Algorithmus kann mit mehreren Datenstrukturen arbeiten, welche in separaten Frames animiert werden können. Der **MultiFrameManager** regelt und verwaltet alles, was alle Frames als Gesamtheit betrifft. Im Moment ist seine einzige Aufgabe die Verwaltung der Pause: Er nimmt den Pause - bzw. Continue-Wunsch von einem beliebigen Frame entgegen und schickt ihn an alle Frames weiter.

Das Paket **peter.sa2.structures** enthält Interfaces und abstrakte Basisklassen, von welchen die datenstrukturspezifischen Klassen in den tiefer liegenden, datenstrukturspezifischen Paketen abgeleitet sind.

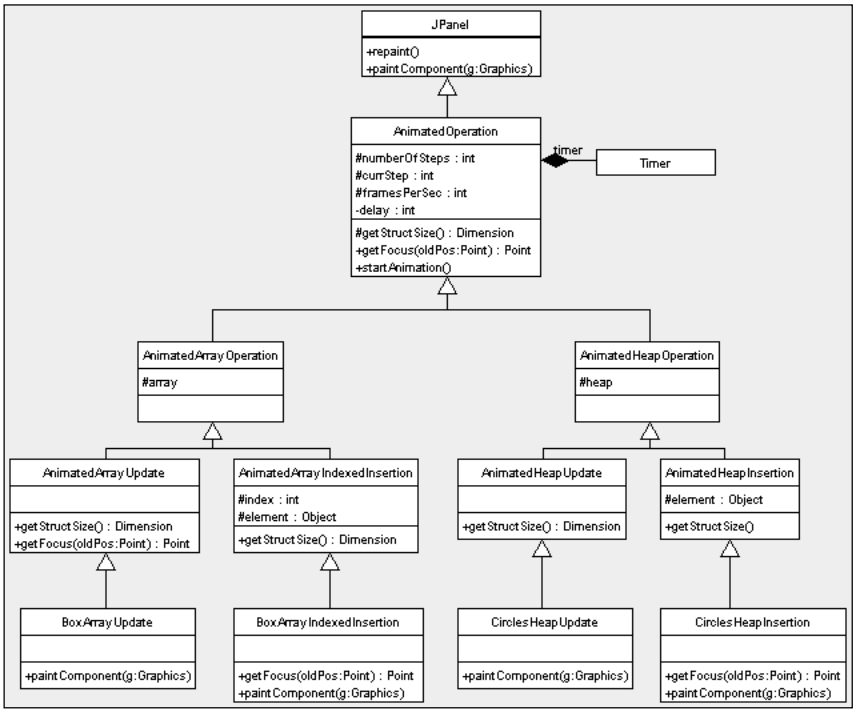

Bild 5: Vererbungshierchie der Panels für die animierten Graphiken

Die wichtigste Klasse in **peter.sa2.structures** ist **AnimatedOperation**, welche von **JPanel** abgeleitet ist. Ein Panel ist eine Zeichenfläche, auf der (statische) Vektorgraphiken gezeichnet werden können durch überschreiben der paintComponent()-Methode. **AnimatedOperation** erweitert **JPanel** durch animationsspezifische Eigenschaften. Die wichtigste dieser Eigenschaften ist ein **Timer**-Objekt, welches die Bilderfolge der Animation, welche in der Zeit parametrisiert ist (0 < currStep <= numberOfSteps), zeitlich gestaffelt und getaktet auf den Bildschirm bringt. Dazu wird via repaint() jedesmal die paintComponent()-Methode aufgerufen, die in den abgeleiteten Klassen für spezifische Ansichten von spezifischen Datenstrukturen überschrieben werden muss. Die Vererbungshierarchie dieser Panels (siehe Bild 5) verläuft parallel zur Vererbungshierarchie der Pakete (im UML-Paketmodell; in Java stehen die verschiedenen Pakete in keinerlei Beziehung zueinander).

Auf der nächst tieferen Ebene befindet sich ein Paket pro animierter Datenstruktur. Bis jetzt sind **peter.sa2.structures.array** und **peter.sa2.structures.heap** implementiert. **peter.sa2.structures.array** enthält die Klasse **AnimatedArrayList**, abgeleitet von **java.util.ArrayList**. Der Anwender, der einen bestehenden Algorithmus animieren will, welcher Operationen auf einer Instanz von **java.util.ArrayList** ausführt, braucht nur diese eine Klasse zu importieren und die Instanz von **ArrayList** durch eine Instanz von **AnimatedArrayList** zu ersetzen, und schon wird sich der Algorithmus automatisch animieren. Alles gilt sinngemäss auch für **peter.sa2.structures.heap**.

Weiter enthält Paket **peter.sa2.structures.array** die abstrakte Klasse **AnimatedArrayOperation**, abgeleitet von **AnimatedOperation**, sowie für jede Operation, welche **ArrayList** anbietet, eine weitere abstrakte Klasse, welche von **AnimatedArrayOperation** abgeleitet ist und das Panel um ansichtsunabhängige Teile der Operationsanimation ergänzt, z.B. **AnimatedArrayIndexedInsertion** für die Methode add(int index, **Object** element).

Weiter enthält das Paket je eine Abstrakte Fabrik für Ansicht und Animationstiefe (siehe [GoF95], Seite 50ff. und Seite 87ff. für Erklärungen zum Abstract Factory Pattern). Pro ange-

botener Animationstiefe ('all operations' bzw. 'assignments') enthält es eine konkrete Fabrik. Pro Freiheitsgrad (Ansicht bzw. Tiefe) enthält es weiter ein Panel mit einer ComboBox zur Auswahl der gewünschten Ansicht bzw. Tiefe.

Auf der untersten Ebene befindet sich ein Pa ket pro Ansicht einer animierten Datenstruktur, z.B. **peter.sa2.structures.array.box** für BoxArrayView (d.h. Darstellung des Arrays mit Kästchen, siehe Bild 3). Es enthält die konkrete Fabrik **BoxArrayViewFactory** sowie pro animierter Operation eine konkrete K lasse, in deren paintComponent() -Methode nun endlich der Code steckt, welcher die Operation vollständig graphisch darstellt, räumlich und zeitlich parametrisiert, z.B. **BoxArrayIndexedInsertion** für die Methode add(int index, **Object** element). Die Klasse **BoxArrayCommSubs** ist eine Sammlung von 'common subroutines', welche primitive Operationen ausführen, z.B. das Zeichnen von einzelnen Kästchen. Alles gilt sinngemäss auch für **peter.sa2.structures.heap.circles**.

5.2 Architektur und dynamischer Ablauf

Nach dem Beschrieb der (statischen) Klassenstruktur des Frameworks im Kapitel 5.1 wird nun das dynamische Zusammenspiel der Instanzen der oben beschriebenen Klassen näher beleuchtet. Die Funktion des **MultiFrameManagers** und das Pausenverhalten der **AnimationFrames** wurden bereits in Kapitel 5.1 bzw. in Kapitel 2 beschrieben. Im Rahmen dieses Überblicks soll nur auf zwei weitere Szenarios eingegangen werden: Aufruf einer animierten Operation aus einem Algorithmus und Ansichtswechsel. Für detailliertere Analysen des Lauf zeitverhaltens wird wiederum auf die Code -Dokumentation und auf die Ausführung des Codes selbst verwiesen. Da im Entwurf des Frameworks standardisierte Ent - wurfsmuster verwendet wurden, wird die entsprechende Terminologie verwendet, um die Ar - chitekturprinzipien aufzuzeigen, welche das dynamische Verhalten der Software bestimmen.

AUFRUF EINER ANIMIERTEN OPERATION AUS EINEM ALGORITHMUS

Als Beispiel wird die Methode add(int index, **Object** element) aus **ArrayList** verwendet:

Die Methode wird auf einem Objekt vom statischen Typ **ArrayList** und vom dynamischen Typ **Animated ArrayList** aufgerufen, z.B. aus einem Sortieralgorithmus:

```
import peter.sa2.structures.array.AnimatedArrayList;
public class SelectionSort {
   public static void main(String[] args) {

      // ...
      array.add(i, array.remove(minIndex));
      // ...
}
```

Es geht in dieser Betrachtung nur um den Aufruf der add -Methode nach abgeschlossenem Aufruf der remove-Methode, welche ein Objekt vom Typ **Object** zurückgibt. Die aufgerufene Methode sieht folgendermassen aus:

```
public void add(int index, Object element) {

   description = "\nAdding element \"" + limitedString(element.toString())
            + "\" at posititon " + index;

   animFrame.performAnimatedOperation(
      description, depthFactory.getAnimatedArrayIndexedInsertion(
         viewFactory, animFrame, new ArrayList(decoratedStruct), index, element));

   decoratedStruct.add(index, element);
}
```

Zuerst wird die Textdarstellung der Operation synthetisiert. Sie wird der Methode performAnimatedOperation der Klasse **AnimationFrame** als erstes Argument mitgegeben und vom **AnimationFrame** animFrame auf dem konsolenartigen Textpanel ausgegeben.

Die Beziehung zwischen **AnimatedArrayList** und **AnimationFrame** ist einseitig: Die animierte Datenstruktur instantiiert das **AnimationFrame**, in welchem sie dargestellt werden soll, in ihrem Konstruktur und behält eine Referenz auf diese Instanz, wohingegen **AnimationFrame** nie eine Referenz auf die Datenstruktur erhält, die es darstellt. Die Gesamtarchitektur des Systems ist Model/ View/Controller (MVC). Dabei ist **AnimatedArrayList** das Modell, die Text- und Graphikpanels bilden die Ansicht und die Steuerelemente am rechten Rand sind die Controllers. Das **AnimationFrame** ist nur ein Container, welcher die in ihm enthaltenen Elemente anzeigt:

- **PauseButtonPanel** und **SpeedSelectorButtonPanel**: datenstrukturunabhängig, befinden sich deshalb wie **AnimationFrame** im Paket **peter.sa2.framework**. Sie werden im Konstruktor von **AnimationFrame** instantiiert. Beide Steuerelemente enthalten auch eine Referenz auf das **AnimationFrame**, in welchem sie angezeigt werden. Wenn der Pause/ Continue-Button einen Benutzerinput erhält, meldet er ihn via **AnimationFrame** an den **MultiFrameManager** weiter, welcher ihn wiederum an alle Frames weiterleitet. Der Durchfluss dieser Nachrichten durch **AnimationFrame** hat den Vorteil, dass wirklich nur die Frames den **MultiFrameManager** kennen; der Pause/Continue-Button kennt einzig sein **AnimationFrame** und ist von den Details der Pausenverwaltung entkoppelt. Wenn der Benutzer die Animationsgeschwindigkeit ändert, wird durch das **SpeedSelectorButtonPanel** die Geschwindigkeit im **AnimationFrame** neu gesetzt, wo sie später von den Panels, welche Operationen animieren, abgefragt werden kann.

- **JTextArea** zur Textdarstellung der animierten Operationen (Konsole): ebenfalls datenstrukturunabhängig und im Konstruktor von **AnimationFrame** instantiiert. Der Text dieses Textfelds wird bei der Animation einer Operation durch **AnimationFrame** um die entsprechende Zeile ergänzt. Besitzt keine Referenz auf das **AnimationFrame**.

- Panel, auf dem die Animation einer Operation gezeichnet wird: Wird unmittelbar vor der Operationsausführung von der Datenstruktur via **DepthFactory** bei der **ViewFactory** bestellt und gleich an **AnimationFrame** weitergeschickt (siehe unten). Besitzt ebenfalls keine Referenz auf **AnimationFrame**.

- **DepthSelectorComboPanel** und **ViewSelectorComboPanel**: datenstrukturabhängig, werden im Konstruktor der animierten Datenstruktur (z.B. **AnimatedArrayList**) instantiiert und dem Konstruktor von **AnimationFrame** mitgegeben. Diese Steuerelemente besitzen auch keine Referenz auf **AnimationFrame**, dafür aber eine Referenz auf die **AnimatedArrayList**, in deren Konstruktor sie instantiiert werden. Beide senden Benutzerinputs direkt an **AnimatedArrayList** weiter. Eine animierte Datenstruktur ist normalerweise von einer nicht animierten Datenstruktur abgeleitet. So ist es möglich, einen

bereits bestehenden Algorithmus zu animieren, indem man die nicht animierte durch die animierte Datenstruktur ersetzt. Darüberhinaus implementiert die animierte abgeleitete Datenstruktur das Interface **AnimatedDataStructure**, welches die zwei Operationen enthält, welche eine animierte von einer nicht animierten Datenstruktur unterscheiden: setDepth(**DepthFactory** depthFactory) und setView(**ViewFactory** viewFactory).

Zusammenfassend ergibt sich daraus folgende Verteilung der Kontrollparameter im System: **AnimationFrame** verwaltet die Geschwindigkeit, der **MultiFrameManager** verwaltet die Pause und die animierte Datenstruktur selbst verwaltet die Referenzen auf die momentan gewählten konkreten Fabriken für Animationstiefe und Ansicht.

Vor diesem architektonischen Hintergrund ist es nun möglich, die weiteren Anweisungen der Methode add(int index, **Object** element) aus der Klasse **AnimatedArrayList** zu verstehen (Code auf der gegenüberliegenden Seite ganz oben):

Die zweite Anweisung ruft die Methode performAnimatedOperation auf dem **AnimationFrame** animFrame auf. Als erstes Argument schickt sie wie gesagt die Textbeschreibung der Operation mit. Als zweites Argument übergibt sie das Panel, welches die Animation ausführt. Der Grund, weshalb das zweite Argument seinerseits aus einem Funktionsaufruf besteht, liegt in der dynamischen Konfigurierbarkeit von Ansicht und Animationstiefe. Das der aktuellen Konfiguration entsprechende Panel wird von der Methode getAnimatedArrayIndexedInsertion der konkreten **DepthFactory** depthFactory zurückgegeben, welche ihrerseits folgende Argumente mitbekommt: die Referenz auf die momentan gewählte **ViewFactory** viewFactory, ein e Referenz auf das **AnimationFrame** animFrame (in seiner Eigenschaft als **AnimationManager** zur Abfrage der momentan gewählten Geschwindigkeit), einen Klon der animierten Datenstruktur zur Ermittlung des Zustands vor der Operation und die für die Operation relevanten Parameter (in diesem Falle das einzufügende Element und die Position der Einfügung).

Die momentan aktive konkrete **DepthFactory** bestellt nach ihren eigenen Kriterien ein Panel bei der **ViewFactory**: Falls die zu animierende Operation unter ihrer Animationstiefe liegt, wird nur ein Update bestellt, ansonsten die vollständige Animation der Operation:

```
protected AnimatedArrayOperation getAnimatedArrayIndexedInsertion(
        AbstractArrayViewFactory viewFactory,
        AnimationManager animMan, ArrayList array, int index, Object elem) {
   return viewFactory.getAnimatedArrayIndexedInsertion(animMan, array, index, elem);
}
```

Die **ViewFactory** instantiiert endlich ein Panel und gibt es die ganze Aufrufkette zurück: zurück an die **DepthFactory**, welche es gleich an die **AnimatedArrayList** zurückgibt, welche es sofort an das **AnimationFrame** schickt, welche es einbaut und anzeigt:

```
protected AnimatedArrayIndexedInsertion getAnimatedArrayIndexedInsertion(
      AnimationManager animMan, ArrayList array, int index, Object element)    {

   return new BoxArrayIndexedInsertion(animMan, array, index, element);
}
```

Zuletzt muss die Operation noch an der Datenstruktur selbst vollzogen werden. Dies geschieht durch einen Super- oder Forwardcall:

```
   decoratedStruct.add(index, element);
```

ANSICHTSWECHSEL

Nach den obigen Ausführungen über die Architektur ist dieses zweite Szenario nun leicht zu verstehen:

Der Benutzer wählt eine neue Ansicht durch die 'Select View' -ComboBox (siehe Bild 3). Die ComboBox ist Teil des **ViewSelectorComboPanels**, welches eine Referenz auf die animierte Datenstruktur enthält, wie oben beschrieben. Das **ViewSelectorComboPanel** kreiert bei seiner eigenen Instantiierung eine Instanz von jeder konkreten **ViewFactory**. Der Benutzerinput löst den Aufruf der Methode setView(**ViewFactory** viewFactory) aus, welche im Interface **AnimatedDataStructure** spezifiziert und von jeder animierten Datenstruktur implementiert wird, wie oben ebenfalls beschrieben. Als Argument wird eine Referenz auf diejenige Fabrik mitgegeben, die für die vom Benutzer gewünschte Ansicht zuständig ist. Die setView -Methode bewirkt schlicht den Wechsel der Referenz von der alten auf die neue Fabrik in der animierten Datenstruktur:

```
public void setView(ViewFactory viewFactory){
        this.viewFactory = (AbstractArrayViewFactory)viewFactory;
        update();
}
```

6. Schrittweises Vorgehen zur Implementierung einer neuen Datenstruktur

Diese Anleitung soll kurz gehalten werden. Wer neue Datenstrukturen implementieren will, muss die Funktionsweise des Frameworks genau verst ehen und wird nicht darum herum kommen, die Dokumentation und den Code selbst zu studieren. Im Folgenden wird eine Art Checklist präsentiert, was zur Implementierung einer neuen Datenstruktur getan werden muss.

PRO ANIMIERTER DATENSTRUKTUR

Für jede anim ierte Datenstruktur muss ein Paket erstellt werden im Stil von **peter.sa2.structures.array**. Darin muss die Klasse **AnimatedYourDataStructure** implementiert werden:

```
public class AnimatedYourDataStructure extends YourDataStructure
             implements AnimatedDataStructure {

}
```

In dieser Klasse müssen alle Methoden der Basisklasse durch Animationsaufrufe dekoriert werden. Zudem müssen die im Interface **AnimatedDataStructure** spezifizierten Methoden setDepth(**DepthFactory** depthFactory) und setView(**ViewFactory** vie wFactory) implementiert werden. **AnimatedArrayList** soll als Beispiel dienen.

Weiter sind folgende Klassen gemäss den Beispielen (in Klammern) in **peter.sa2.structures.array** zu implementieren:

- **AbstractYourDataStructureDepthFactory** (**AbstractArrayDepthFactory**)
- **YourDataStructureDepthSelectorComboPanel** (**ArrayDepthSelectorComboPanel**)
- 1 konkrete DepthFactory pro Animationstiefe (**AnimateAllDepthFactory**, **AssignOnlyDepthFactory**)
- **AbstractYourDataStructureViewFactory** (**AbstractArrayViewFactory**)
- **YourDataStructureViewSelectorComboPanel** (**ArrayViewSelectorComboPanel**)
- Basisklasse für alle Operationen (**AnimatedArrayOperation**)

PRO ANIMIERTER OPERATION

Für jede animierte Operation ist ein abstraktes Panel im Stil von **AnimatedArrayIndexed-Insertion** zu erstellen und darin die getStructSize-Methode zu implementieren: getStructSize soll die grösste Dimension der Datenstruktur in Elementen während der Operation zurückgeben (Höhe, Breite). Die Grösse in Elementen ist – im Gegensatz zur Grösse in Pixeln – ansichtsunabhängig und deshalb auf dieser abstrakten Ebene anzusiedeln.

PRO ANSICHT

Für jede Ansicht der Datenstruktur muss ein Paket erstellt werden im Stil von **peter.sa2.structures.array.box**. Darin ist eine konkrete ViewFactory nach dem Beispiel von **BoxArrayViewFactory** zu implementieren. Weiter ist darin eine CommSubs-Klasse nach dem Beispiel von **BoxArrayCommSubs** zu erstellen. Sie enthält einerseits die Methode calculatePanelSize, welche ansichtsspezifisch aus der abstrakten Grösse der Datenstruktur (in Elementen) die konkrete Grösse (in Pixeln) berechnet, andrerseits Grundroutinen für die Graphikprogrammierung (siehe Kapitel 8 zur Idee, diese Routinen auf einer höheren Ebene anzusiedeln).

PRO OPERATION PRO ANSICHT

Für jede Operation ist nun ein konkretes Panel zu implementieren, welches zwei öffentliche Methoden enthält:

- GetFocus(**Point** oldPos): Gibt den Focus zurück, d.h. die Panelkoordinaten, welche in der ScrollPane, in welche das Panel eingebettet ist, die obere linke Ecke einnehmen sollen. Beispiel: Beim Array wird der Focus vertikal so gewählt, dass das eingefügte oder herausgenommene Element etwa ein Viertel der Panelhöhe vom oberen Rand entfernt ist. In der horizontalen wird die alte Position beibehalten, die deshalb immer als Argument mitgegeben werden muss.
- PaintComponent(**Graphics** g): Die wesentliche Graphikroutine, in der die Animation der Operation implementiert ist:
  - Parametrisierung im Ort: `x, y`
  - Parametrisierung in der Zeit: `for(currStep=0;currStep<numberOfSteps; ++currStep)`
  - Iteration über alle Elemente der Datenstruktur

Anlässlich einer Präsentation des Frameworks vor einer Gruppe von theoretischen Informatikern war Skepsis zu spüren, ob sich der Einsatz des Frameworks wirklich lohne, wenn man jedesmal so viele Klassen ableiten müsse. Ich kann verstehen, dass die Menge der zu erstellenden Klassen am Anfang abschreckend wirkt. Trotzdem glaube ich, dass sich der Einsatz lohnt: Die meisten der oben aufgeführten Klassen sind entweder ganz leere abstrakte Marker-Klassen, oder sie erweitern eine bestehende Klasse nur um ein oder zwei Methoden von wenigen Zeilen. Zudem können sie durch 'copy and paste' einfach von den bestehenden Beispielen übernommen und modifiziert werden. Diese Form von 'copy and paste' wird durch die Verwendung von Patterns leider nicht verhindert, sondern im Gegenteil noch gefördert durch die Einführung von Vererbungshierarchien ausschliesslich zur Schaffung polymorpher Datenstrukturen. Der Gewinn besteht in erhöhter Flexibilität für den Benutzer (Ansichtswechsel zur Laufzeit, etc.). Spätestens bei der Implementierung der zweiten Datenstruktur wird die Erstellung all dieser kleinen Klassen keine Probleme mehr bereiten. Es ist natürlich, dass die Planung für Wiederverwendbarkeit am Anfang einen gewissen Overhead erfordert, der sich erst mit der Zeit (mit dem Wachstum der Bibliothek von animierten Datenstrukturen) auszahlt. Der Hauptaufwand bei der Implementation einer neuen Datenstruktur liegt sicher nicht in den kleinen Klassen zur Eingliederung in die bestehende Architektur, sondern in der einen graphischen Methode paintComponent. Hier könnte die Schaffung einer Graphics-Toolbox Abhilfe schaffen (siehe Kapitel 8).

7. Erläuterung der wichtigsten Design-Entscheide

ANIMATION MANAGER

Die Verwaltung der Geschwindigkeit ist die einzige Verwaltungsaufgabe (control task) des **AnimationFrames**. Man mag es für inkonsistent halten, dem **AnimationFrame**, welches sonst nur ein dummer Container ohne höhere Verwaltungsaufgaben ist, eine einzige solche Aufgabe zu übertragen. Es wäre vielleicht konsequenter, **AnimationFrame** wirklich nur als Container ohne eigene Intelligenz zu implementieren und einen separaten **Animation-Manager** zu schaffen. Andrerseits kann man argumentieren, dass die Schaffung eines separaten Managers für die Verwaltung eines einzigen Attributs (float speed) übertrieben ist. Deshalb wurde entschieden, das Interface **AnimationManager**, welches nur eine einzige Methode getSpeed() besitzt, durch die Klasse **AnimationFrame** zu implementieren.

ABSTRACT FACTORY PATTERN

Die Wahl der Abstrakten Fabrik als zentrales Element der Architektur war keine sehr gut motivierte Wahl, sondern mehr ein Zufallsentscheid bzw. ein Hack: In der Phase der Entwicklung, als ich noch mit den Grundtechniken (Java, Swing-Bibliothek) kämpfte, wusste ich nicht, wie man ausserhalb der paintComponent-Methode einer von **JPanel** abgeleiteten Klasse Zeichnungsroutinen implementiert. Heute ist mir klar, dass man einfach das **Graphics**-Objekt der paintComponent-Methode übergeben müsste. Damals sah ich die einzige Möglichkeit darin, für jede animierte Operation ein neues Panel zu kreieren. Heute weiss ich, dass man eigentlich immer wieder das gleiche Panel übermalen könnte. Aus der Abstrakten Fabrik würde dann das Strategy Pattern, welches nichts zurückgibt, sondern nach den gleichen Kriterien einfach die richtige Zeichnungsroutine zur Laufzeit auswählt. Allerdings wären weitere Anpassungen nötig und weitere Fragen neu zu klären, z.B. wo dann das **Timer-**

Objekt am besten angesiedelt würde. Der Wechsel hätte also Auswirkunge n auf die Architektur und würde ein bisschen Nachdenken erfordern. Da ich diese Zeit nicht mehr habe, habe ich entschieden, lieber einstweilen die Abstrakte Fabrik zu belassen als mit einem Schnellschuss neue Probleme einzuführen.

Ein Nachteil der Abstr akten Fabrik ist sicher der grosse Aufwand für die Kreation all dieser vielen Panel-Objekte. In den Tests war allerdings Performanz nie ein Problem, also ist der Effekt wohl noch nicht kritisch. Weitere Vorteile der Umstellung auf das Strategy Pattern wäre die Erweiterbarkeit auf Einzelbild-Animation mit der Möglichkeit, die Animation bei jedem Bild anzuhalten. Im Moment wird die laufende Operation nach drücken der Pausentaste noch zu Ende geführt, denn nur das **AnimationFrame** wird darüber informiert, dass Pause gewünscht wurde und einstweilen keine weiteren Animationen zu starten sind; das momentan aktive Panel erfährt davon nichts mehr. Sobald Einzelbild-Animation zur Verfügung steht, wird der Benutzer nicht nur Vorlauf, sondern auch Rücklauf wünschen. Rücklauf über den Beginn der aktuellen Operation hinaus wäre allerdings nicht trivial, da man Konsistenz mit dem Algorithmus und mit weiteren Datenstrukturen, die durch den gleichen Algorithmus manipuliert werden, sicherstellen müsste. Diese Erweiterung wäre ein Projekt für sich und war ebenfalls nicht durch einen Schnellschuss zu bewerkstelligen.

FUNKTIONSWEISE DES Timer-OBJEKTS

Im Moment wird das **Timer**-Objekt zum Beginn der Animation einer Operation gestartet. Darauf löst es jeweils nach jeder Zehntelss ekunde einen **ActionEvent** aus, der von einem zugehörigen **ActionListener** aufgefangen wird, welcher daraufhin den Zeitparameter inkrementiert und via repaint() die paintComponent-Methode aufruft, welche das nächste Bild zeichnet. Der **Timer** arbeitet dabei in einem eigenen Thread und der Hauptthread der Applikation muss angehalten werden. Im Moment wird er einfach für die ganze Zeit der Animation eingeschläfert und wacht nachher selbständig wieder auf:

```
Thread.sleep((int)(opExecTime*mainThreadPauseFactor));
```

Dabei ist opExecTime die Zeit, welche die Animation der Operation benötigt, und mainThreadPauseFactor ein Sicherheitsfaktor für den verlängernden Effekt von rückwärtigen Prozessen z.B. des Betriebssystems. Für mainThreadPauseFactor hat sich 1.3 bewährt. Dieses Vorgehen wurde bei einer Präsentation kritisiert, weil es die Gefahr birgt, dass im Hintergrund einmal so viele Prozesse ablaufen, dass die Animation der nächsten Operation bereits gestartet wird, bevor die Animation der aktuellen Operation beende t ist, was zu einem unschönen Ruck führen würde. Es wurde deshalb vorgeschlagen, den Thread explizit einzuschläfern und wieder aufzuwecken. Ich habe daraufhin ein paar Stunden in diese Angelegenheit investiert, ohne zu einem befriedigenden Resultat zu k ommen. Multithreading scheint eine diffizile Angelegenheit zu sein. Verschiedene Methoden der **Thread**-Klasse sind inzwischen deprecated, weil sie zu fehleranfällig waren oder zu oft missbraucht wurden. Die momentane Lösung funktioniert, und eine allfällige Änderung würde am besten durch jemanden vorgenommen, der eine gewisse Erfahrung mit Multithreading hat.

DECORATOR PATTERN

Die Grundidee unseres Ansatzes besteht darin, die animierten Datenstrukturen von ihren nicht animierten Versionen abzuleiten, um dank Polymorphie bestehende Algorithmen, welche für die nicht animierten Datenstrukturen geschrieben wurden, auch mit den animierten Versionen verwenden und so fast automatisch animieren zu können. Dafür müssen wir die Schnittstelle erben (Subtyping), was keine Probleme verursacht. Fraglich ist hingegen, ob wir auch die Implementation erben können (Subclassing), was eine Reihe von viel diskutierten Problemen mit sich bringt (Semantic Fragile Base Class Problem, siehe z.B. [Szy97], Kapitel 7).

Die Ergebnisse dieser Diskussion führen zu einer einfachen Antwort:

- Wenn der Entwickler der abgeleiteten Klasse die Basisklasse selbst unter Kontrolle hat (z.B. **Heap**), kann er problemlos die gesamte Implementation erben. Nach einer allfälligen Änderung der Basisklasse muss er die Gültigkeit der abgeleiteten Klasse überprüfen und falls nötig wieder herstellen.

- Falls er die Basisklasse nicht selbst kontrolliert (z.B. **ArrayList** als Teil der Java API), dann kann durch Implementationsvererbung Folgendes passieren: Zwei Methoden der Basisklasse, welche zum Zeitpunkt der Entstehung der abgeleiteten Klasse unabhängig voneinander implementiert sind (z.B. add(int index, **Object** Element) und addAll(int index, **Collection** c) in **ArrayList**) könnten in einer zukünftigen Version der Basisklasse so implementiert werden, dass die letztere die erstere intern aufruft. An der veröffentlichten Schnittstelle würde sich nichts ändern und der Entwickler der abgeleiteten Klasse würde über diese Veränderung nicht informiert. Die Veränderung würde sich aber auch auf die abgeleitete Klasse auswirken und hätte unschöne Effekte zur Folge: Die Animation der addAll-Methode würde durch die Animation der add-Methode unterbrochen bzw. von ihr überlagert, was sicher nicht im Sinne des Erfinders liegt.

Für den zweiten Fall sind v.a. zwei Abhilfen bekannt, die in ihrer Wirkung äquivalent sind:

- State Test Functions (siehe z.B. [Szy97], Kapitel 5.4 / 5.5): Die abgeleitete Klasse enthält einen integer 'state', der im Grundzustand null ist. Beim Eintritt in eine Methode wird er inkrementiert, beim Austritt dekrementiert. Wenn er beim Eintritt vor der Inkrementation 0 ist, wird die ganze Methode ausgeführt, ansonsten nur der Supercall. Dieser Mechanismus stellt sicher, dass immer nur die von aussen aufgerufene Operation animiert wird, intern zu Hilfe gezogene Methoden hingegen nicht.

- Decorator Pattern (siehe [GoF95], Seite 175): Jede Methode der abgeleiteten Klasse muss überschrieben werden, und es darf nie ein Supercall ausgeführt werden. Stattdessen enthält die abgeleitete Klasse zusätzlich eine Referenz auf ein Objekt der Superklasse, an welches es alle Operationen delegiert, die nur den Zustand des Objekts der Superklasse (Teilmenge des Zustands der abgeleiteten Klasse) betreffen. In diesem Entwurfsmuster wird die Animation als Dekoration der ursprünglichen Datenstruktur betrachtet und von den Operationen auf der Datenstruktur selbst entkoppelt.

Für **AnimatedArrayList** wurde das Decorator Pattern gewählt.

NAMENSKONVENTION: AnimatedArrayList STATT NUR ArrayList

Im Moment sind zur Wiederverwendung eines bestehenden Algorithmus im Wesentlichen zwei Modifikationen nötig (Details in Kapitel 3):

1. Einfügen der Zeile

```
import peter.sa2.structures.array.AnimatedArrayList;
```

2. und ersetzen der Zeile

```
ArrayList your_array_name = new ArrayList();
```

durch

```
ArrayList your_array_name = new AnimatedArrayList();
```

für diejenige(n) Instanz(en) von **ArrayList**, die man animieren will.

Es wäre sogar möglich, die zweite Modifikation einzusparen, wenn man die abgeleitete Klasse ebenfalls bloss **ArrayList** nennen würde. Durch das Importstatement wäre klar definiert, welche Klasse zu verwenden ist. Dieser Vorteil würde allerdings nur solange bestehen, als dass man in einem Algorithmus ausschliesslich die animierte Version verwendet. Oft verwendet man aber beide Versionen nebeneinander (siehe Kapitel 3). Wenn nun beide Klassen gleich hiessen, müsste man deshalb immer die voll qualifizierten Namen verwenden, wodurch man nichts gespart, sondern sich im Gegenteil noch mehr Schreibarbeit aufgebürdet hätte. Aus diesem Grund wurde die bestehende Lösung gewählt.

DARSTELLUNG DES ARRAYS

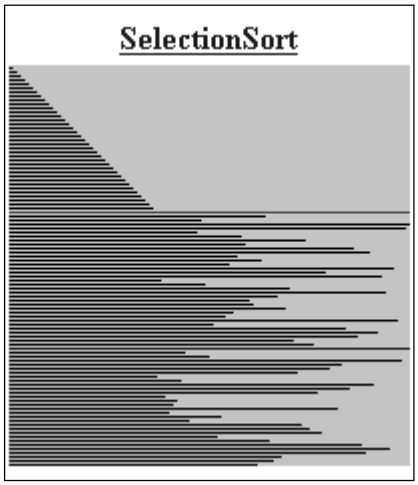

Bild 6: SortAnimation-Darstellung

Die grösste Herausforderung bei der Visualisierung von abstrakten Dingen wie Datenstrukturen besteht sicher darin, eine geeignete Darstellung zu finden. Wie immer gilt es auch hier, im Spektrum zwischen totaler Spezialisierung und totaler Verallgemeinerung eine sinnvolle Position zu finden.

Die **SortAnimation**-Darstellung (siehe Anhang A: **SortAnimation**) ist natürlich für das spezifische Problem der Darstellung von Sortieralgorithmen auf einem Array der ersten n Integerzahlen besser geeignet als unsere BoxArray-Darstellung (vergleiche Bild 6 mit Bild 3). Der Vorteil unserer Darstellung liegt in ihrer Allgemeinheit: Die Elemente in **ArrayList** sind vom Typ **Object**, also bei weitem nicht auf Zahlen beschränkt. Bei nicht-numerischen Typen macht aber die unterschiedliche Stäbchen- oder Kästchenlänge zur Charakterisierung keinen Sinn mehr. Der Text in den Kästchen wird durch die toString-Methode in der generierenden Klasse des entsprechenden Objekts definiert: So kann z.B. auch die alphabetische Sortierung von Videokassetten visualisiert werden, sofern die compareTo-Methode und die toString-Methode dementsprechend überschrieben sind.

8. Offene Punkte

GRAPHICS TOOLBOX

Anlässlich einer Präsentation des Frameworks vor einer Gruppe von theoretischen Informatikern war Skepsis zu spüren bezüglich der Einsparung an Entwicklungszeit bei der Verwendung des Frameworks für die Implementation neuer animierter Datenstrukturen. Der Hauptteil des Entwicklungsaufwands liegt in den Graphikroutinen, und gerade da bietet das Framework wenig Unterstützung. Die Swing -Bibliothek bietet Punkte, Geraden, Kreise und Vierecke als einzige Primitiven an. Das Animationsframework sollte deshalb selbst graphische Elemente auf höherem Niveau zur Verfügung stellen. Ein Ansatz dazu besteht in den **CommSubs**-Klassen, in welchen Grundelemente für eine bestimmte Darstellung einer einzelnen Datenstruktur ausgelagert sind. Der Ansatz ist allerdings wiederum "handgestrickt", da für jede neue Datenstruktur wieder neue solche Grundelemente definiert werden müssen.

Die Lösung für dieses Problem liegt auf der Hand: Die **CommSubs**-Klassen stehen bereits jetzt in einer Vererbunghierarchie, welche parallel zu jener der Pakete verläuft; man könnte also einfach mehr konkrete Graphikroutinen von der untersten Ebene (Pakete für einzelne Ansichten einzelner Datenstrukturen) auf die höchste Ebene (Paket **peter.sa2.structures**) verschieben, ohne die Architektur des Frameworks zu verändern. Allerdings sollten solche Bibliotheken umsichtig entworfen werden: Die zur Verfügung gestellten Graphikroutinen müssen allgemein genug sein, um ihre Ansiedlung auf dieser hohen Ebene zu rechtfertigen und häufige Wiederverwendung zu ermöglichen. Die Entwicklung einer brauchbaren Bibliothek graphischer Grundelemente hat deshalb die Dimension eines separaten Projekts.

Anlässlich desselben Vortrags wurde auch der Wunsch nach einer graphischen Programmieroberfläche geäussert: Graphische Grundelemente sollten durch 'drag and drop' manipuliert und so ganze Darstellungen abstrakter Datentypen 'zusammengeklickt' werden können. Diese Strategie wird zwar für die Komposition statischer Einzelbilder erfolgreich angewendet, aber es stellt sich die Frage, wie man die zeitliche Entwicklung (Δ zwischen zwei Einzelbildern) graphisch spezifizieren könnte, denn man will ja nicht Bild für Bild einzeln komponieren. Die Entwicklung einer solchen Oberfläche oder der Schnittstelle zu einem bestehenden Grahiktool wäre ein separates Projekt, wobei an den Erfolgschancen gezweifelt werden darf.

ZOOM: 3. FREIHEITSGRAD NEBEN ANSICHT UND ANIMATIONSTIEFE

Im Moment kann man nicht in die animierten Datenstrukturen hineinzoomen; eine Art Zoom wird im Falle des Heaps durch verschieden grosse Ansichten imitiert. Zoom ist wünschenswert, v.a. bei zweidimensionalen Strukturen wie Bäumen.

INTERAKTION VON MEHREREN DATENSTRUKTUREN

Viele Algorithmen arbeiten mit mehreren Datenstrukturen und schieben Elemente zwischen den einzelnen Strukturen hin und her. Es wäre schön, interagierende Datenstrukturen gemeinsam im gleichen Frame darzustellen zu können. Die Entwickler des JDSL Visualizers (siehe Anhang A: The JDSL Visualizer) behaupten, dies implementiert zu haben.

TEIL III: RECHENSCHAFTSBERICHT

Nachdem in den ersten beiden Teilen dieses Dokuments das Softwareframework beschrieben wurde, welches als Resultat dieser Semesterarbeit entstanden ist (das *Produkt*), soll nun auf die Arbeit an sich (das *Projekt*) eingegangen werden: Fragestellung, Vorgehen und Evaluation der Resultate.

9. Aufgabenstellung

Die grobe Aufgabenstellung lautete gemäss Projektausschreibung ([Cieliebak01]):

Um Algorithmen oder Datenstrukturen zu erklären, werden häufig Animationen als Hilfsmittel verwendet. Es gibt heute bereits eine Unzahl von Animationen für verschiedene Datenstrukturen und Algorithmen. Diese sind jedoch grösstenteils "handgestrickt", d.h. zu jedem Algorithmus wird eine individuelle Animation erzeugt.

Ein alternativer Ansatz besteht in der Verwendung von Datenstrukturen, die bereits Animationen für Grundoperationen zur Verfügung stellen. Für ein Array könnte z.B. eine Methode "swap(i,j)" existieren, die das i-te und j-te Element vertauscht und diesen Vorgang ausserdem visuell darstellt. Solche Datenstrukturen können verwendet werden, um Algorithmen halbautomatisch zu animieren.

Das Ziel dieser Arbeit ist, ein Framework für animierte Datenstrukturen in Java zu erstellen und einige einfache Datenstrukturen und Algorithmen im Rahmen dieses Frameworks zu implementieren.

10. Zielsetzungen

Eines der Hauptziele objektorientierter Software-Entwicklung ist **Wiederverwendbarkeit** bestehender Arbeit auf allen Stufen des Entwicklungszyklus. Das zu entwickelnde Framework hatte deshalb zwei wesentlichen Anforderungen zu genügen:

1. Einfache Bedienbarkeit für den Endbenutzer

Als Endbenutzer sollen theoretische Informatiker angesprochen werden, die Algorithmen entwickelt und in Java implementiert haben und diese nachträglich animieren wollen, z.B. zwecks Fehlersuche oder für didaktische Zwecke. Die nachträgliche Animation soll mit minimalem Aufwand möglich sein, d.h., es soll möglichst grosse Wiederverwendbarkeit von bestehenden, nicht animierten Algorithmen ermöglicht werden.

2. Wiederverwendbarkeit von Basisfunktionalität wie Timing und Benutzerschnittstelle bei der Implementierung einer neuen animierten Datenstruktur

Der Entwurf des Animationstools als erweiterbares Framework soll möglichst grosse Wiederverwendbarkeit der hier entwickelten Architektur bei der Implementation von neuen animierten Datenstrukturen ermöglichen.

" A **framework** is a **set of cooperating classes that make up a reusable design** for a specific class of software. ... The framework dictates the architecture of your application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. **A framework predefines these design parameters so that you, the application designer/implementor, can concentrate on the specifics of your application.** ... Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately. ... When you use a toolkit (or a conventional subroutine library for that matter), you write the main body of the application and call the code you want to reuse. When you use a framework, **you reuse the main body and write the code it calls**." ([GoF95], Seite 26f.)

Der Application Designer/Implementor ist im Fall unseres Frameworks ein Programmierer mit Grundkenntnissen in Java, in Algorithmen/Datenstrukturen und in Graphik/Kinematik. Er ist Anwender des hier entwickelten Frameworks (*intermediate user*) und Applikationsentwickler für den Endbenutzer (*intermediate developer*). Seine grösste Herausforderung wird darin bestehen, eine geeignete Repräsentation des abstrakten Datentypen zu finden und die Operationen, die der Datentyp anbietet, räumlich und zeitlich parametrisiert zu beschreiben. Auf diese Aufgaben soll er sich konzentrieren können, indem er an klar definierten Stellen des Frameworks abgeleitete Klassen hinzufügt, in welchen er seine Animationsroutinen ausprogrammiert. Das Zusammenspiel der Klassen und die Reihenfolge, in welcher sie ihre Methoden aufrufen, ist hingegen bereits vorgegeben, und die Benutzerschnittstelle mit all ihren Panels, Buttons und Scrollbars und der entsprechenden Funktionalität ist bereits vorhanden und kann vollständig wiederverwendet werden.

11. Persönliche Ziele und Motivation

Seit meinem ersten Pascal-Programmierkurs im Chemie-Grundstudium an der EPF Lausanne bin ich begeistert von allem, was mit Software zu tun hat. Leider ging meine Ausbildung im Grundstudium nicht über den einsemestrigen Pascal-Kurs und einen einsemestrigen MATLAB-Kurs hinaus.

Im Fachstudium Rechnergestützte Wissenschaften wurden auf dieses schwache Fundament hochstehende Vorlesungen wie 'Graphische Datenverarbeitung I' und 'Softwarekonstruktion' aufgesetzt. Bis zur Neugestaltung der Vorlesung 'Softwarekonstruktion' im Wintersemester 2001/2002 wurde der "Mittelbau" vollständig ausgeklammert: Algorithmen und Datenstrukturen, strukturierte Programmierung, objektorientierte Programmierung, Entwurfsmuster.

In der Vorlesung 'Softwarekonstruktion', wie sie von Prof. Thomas Gross im Wintersemester 2000/2001 für Studenten der Informatik und der Rechnergestützten Wissenschaften gemeinsam gelesen wurde, lag das Schwergewicht nicht auf der Implementationsphase. Es wurde ein Überblick über den gesamten Entwicklungszyklus vermittelt: Analyse, Entwurf, Implementation, Wartung. Für das Begleitprojekt konnte ich mir von Java nur gerade das Allernötigste aneignen. Mein Programmierstil war im besten Fall objekt-basiert, aber sicher nicht objektorientiert.

Die Grundlagen über Algorithmen und Datenstrukturen habe ich mir im Sommersemester 2001 im Selbststudium mit [Wirth86] angeeignet.

Ich hatte also zwei Gründe für die Wahl gerade dieser Semesterarbeit:

1. Ich wollte meine Kenntnisse in zwei Gebieten, in denen ich zuvor nur eine theoretische Grundahnung hatte, durch eine praktische Arbeit vertiefen und festigen: Algorithmen/Datenstrukturen und Java/OOP.

2. Die Arbeit versprach Befriedigung, weil dabei nicht einfach ein Artikel entstehen sollte, der dann in irgendeiner Schublade verstauben würde, sondern ein Produkt, welches tatsächlich Verwendung finden kann, weil es das Erlernen von Algorithmen und Datenstrukturen vereinfacht (Animation in der Didaktik) oder den Entwicklungsprozess von Algorithmen und Datenstrukturen beschleunigt (Animation in der Fehlersuche).

12. Projektablauf

Im Projektplan vom 25. Oktober 2001 ([Häfliger01]) wurde vorgesehen, die vorgeschriebenen 180 Arbeitsstunden auf die ersten neun Wochen des Wintersemesters vor Weihnachten zu verteilen, und zwar gleichmässig mit etwa zwanzig Stunden pro Woche. Die ersten drei Wochen, also ein relativ grosser, in Anbetracht der Komplexität der Techniken aber immer noch eher knapper Zeitanteil, wurde für das gründliche Erlernen von Java/OOP vorgesehen.

Dieser ursprüngliche Projektplan wurde bereits in diesen ersten drei Wochen komplett über den Haufen geworfen. Die Vorlesung 'Softwarekonstruktion' wird seit dem Wintersemester 2001/2002 von Dr. Dominik Gruntz für die Studenten der Rechnergestützten Wissenschaften separat gelesen. Es wird nun nicht mehr der gesamte Entwicklungszyklus behandelt, sondern vor allem die Implementationsphase: Sechs Lektionen Einführung in Java/OOP, eine Lektion UML, pro weitere Lektion je die Erarbeitung eines einzelnen Entwurfsmusters und die Besprechung seiner Einsatzmöglichkeiten, Vorteile und Kosten.

Es hat sich als viel effizienter herausgestellt, diese völlig umgebaute Vorlesung nochmals zu besuchen, statt die Techniken im Selbststudium zu erlernen. Der Rhythmus wurde aber dadurch gegenüber dem ursprünglichen Plan stark verlangsamt.

Anfang Dezember beherrschte ich die Grundzüge von Java und der API soweit, dass ich mir gezielt die Techniken aneignen konnte, die ich nachher zur Konstruktion des Frameworks brauchte: Entwurf von graphischen Benutzerschnittstellen mit der Swing-Bibiliothek, Vektorgraphik, Anwendung von Entwurfsmustern wie Observer und Abstract Factory, welche die Ansicht (view) vom Modell (model) entkoppeln und es so dem Benutzer ermöglichen, Eigenschaften wie Ansicht oder Animationstiefe zur Laufzeit zu verändern. Daneben habe ich in dieser Zeit mein theoretisches Hintergrundwissen ausgebaut: [Eckel00], [StePoo01], [GoF95].

Nach diesen Vorversuchen habe ich die Grundzüge des Frameworks in der Ferienwoche zwischen Weihnachten und Neujahr am Stück implementiert. Als erste Datenstruktur wählte ich **AnimatedArrayList**, abgeleitet von **ArrayList** aus der Java API, welche sowohl als Array als auch als verbundene Liste eingesetzt werden kann und sich unter anderem gut zur Demonstration von Sortieralgorithmen eignet. Da ich untersuchen wollte, wie aufwändig es ist, sämtliche Operationen einer solchen gegebenen Datenstruktur zu animieren, habe ich sehr viel Zeit auf ArrayList verwendet, was mich den grössten Teil des Januars beschäftigt hat.

Die Resultate waren insofern positiv, als dass mit dem entwickelten Framework die Arbeitsweisen von einfachen Sortieralgorithmen wie SelectionSort, InsertionSort oder BubbleSort auch für Laien sofort nachvollziehbar waren. Die Grenzen traten erst bei HeapSort zutage: Allein aufgrund der Animation des Arrays ist die Sortierstrategie nicht ersichtlich: Elemente werden scheinbar willkürlich vertauscht. Die Erklärung findet sich in jedem Buch über Algorithmen und Datenstrukturen: Der Algorithmus baut in situ einen Heap auf und wieder ab. Um die Funktionsweise zu verstehen, genügt die Animation des Arrays nicht: Der Heap muss animiert werden. So hatte ich also guten Grund, den Heap als zweite Datenstruktur zu wählen, um die Verallgemeinerungsfähigkeit meines Frameworks zu demonstrieren. Aufgrund anderer Projekte konnte ich mich dieser Herausforderung allerdings erst im März widmen. Die Klasse **AnimatedHeap** ist von einer selbst implementierten Klasse **Heap** abgeleitet, welche sich ebenfalls im Paket **peter.sa2.structures.heap** befindet und ihrerseits von **AbstractDispenser** in **peter.sa2.structures** abgeleitet ist.

13. Evaluation

Die Aufgabenstellung wurde erfüllt:

Das Ziel dieser Arbeit ist, ein Framework für animierte Datenstrukturen in Java zu erstellen und einige einfache Datenstrukturen und Algorithmen im Rahmen dieses Frameworks zu implementieren. (aus Kapitel 9: Aufgabenstellung)

Es wurde ein Framework erstellt, und darin wurden zwei Datenstrukturen implementiert: **AnimatedArrayList** und **AnimatedHeap**. Es wurde eine Reihe von Sortieralgorithmen implementiert, welche die Funktionalität des Frameworks demonstrieren.

Das Framework wurde an drei Vorträgen verschiedenem Publikum vorgestellt:

1. den Studenten der Rechnergestützten Wissenschaften im Rahmen der Veranstaltungsreihe 'Fallstudien'

2. der Gruppe Widmayer im Rahmen ihres Gruppenseminars

3. einem Teil einer Abschlussklasse der Hochschule für Gestaltung und Kunst Zürich (HGKZ) im Rahmen eines fünfundzwanzigstündigen Kulturmarathons in einer abbruchreifen Sozialwohnung in Zürich-Altstetten

Feedback des ersten und des dritten Publikums: Lernende und sogar Laien können die Sortierstrategie von HeapSort verstehen, nachdem sie die Animation gesehen haben, ohne Animation hingegen nur schwerlich. Dies ist ein Hinweis darauf, dass sich eine meiner persönlichen Hoffnungen erfüllt hat:

... Produkt, welches tatsächlich Verwendung finden kann, weil es das Erlernen von Algorithmen und Datenstrukturen vereinfacht ... (aus Kapitel 11: Persönliche Ziele und Motivation)

Allerdings war der Beweis des Nutzens von Algorithmenanimationen nicht Teil der Arbeit; dieser Nutzen scheint allgemein anerkannt und war eine Prämisse. Das Ziel des Projekts war der Beweis des Nutzens eines alternativen Ansatzes zur Animation von Datenstrukturen. Das erste der beiden Teilziele wurde klar erreicht (siehe Kapitel 3: Schrittweises Vorgehen zur Animation eines bestehenden Algorithmus):
Einfache Bedienbarkeit für den Endbenutzer (aus Kapitel 10: Zielsetzungen)

Beim zweiten Teilziel besteht im Moment allerdings noch Skepsis (siehe Kapitel 6: Schritt - weises Vorgehen zur Implementierung einer neuen Datenstruktur):

Wiederverwendbarkeit von Basisfunktionalität wie Timing und Benutzerschnittstelle bei der Implementierung einer neuen animierten Datenstruktur (aus Kapitel 10: Zielsetzungen)

Dass Timing und Benutzerschnittstelle wiederverwendet werden können, ist zwar eine Tat - sache, aber es fehlt im Moment noch die Erfahrung, um eine Aussage darüber machen zu können, wieviel die Einsparung an Entwicklungszeit im Vergleich zum totalen Implementa - tionsaufwand beträgt (prozentuale Einsparung). Im Bereich dieses zweiten Teilziels besteht noch Verbesserungspotential. Kapitel 8 versucht eine mögliche Verbesserung aufzuzeigen.

Die persönlichen Ziele wurden erreicht:

... meine Kenntnisse in zwei Gebieten, in denen ich zuvor nur eine theoretische Grundahnung hatte, durch eine praktische Arbeit vertiefen und festigen: Algorithmen/Datenstrukturen und Java/OOP. ... (aus Kapitel 11: Persönliche Ziele und Motivation)

Ich konnte mein Wissen über Algorithmen und Datenstrukturen vertiefen und festigen. Für die Entwicklung des Frameworks war ich gezwungen, mir ein fundiertes Wissen aufzubauen über die Programmiersprache Java und die Konzepte der objektorientierten Programmierung. Bei der praktischen Arbeit konnte ich Erfahrung sammeln in der Entwicklung von interaktiver Software durch Anwendung verschiedener Design Patterns und unter Verwendung der Swing-Bibliothek.

TEIL IV: ANHANG

A. Alternative Simulationstools für Algorithmen und Datenstrukturen

Sämtliche Internet-Adressen sind in Anhang C aufgeführt.

xtango

xtango ist Teil einer ganzen Gruppe von Animationstools (polka, samba, jsamba, etc.). xtango verspricht Sprachunabhängigk eit bezüglich der zu animierenden Algorithmen (wohingegen unser Framework nur mit Algorithmen in Java arbeiten kann), scheint aber expliziten Einbau der Animation in die Algorithmen zu bedingen (was bei uns nicht nötig ist, da wir die Operationen verwenden, welche die animierten Datenstrukturen selbst anbieten).

SortAnimation

Prof. Dr. Robert Stärk von der ETHZ hat zur Demonstration von Multithreading in Java Sor - tieralgorithmen implementiert und auf seiner Homepage veröffentlicht, welche auch ins ent - sprechende Kapitel des Java Tutorials aufgenommen wurden. Sie können als Applets angeschaut werden. Die Grundlage bildet eine abstrakte Klasse **SortAnimation**, welche eine Methode swap(int i, int j) sowie verschiedene Methoden zur Steuerung der Animation an - bietet. Die Klassen, welche die Sortieralgorithmen implementieren, sind von **SortAnimation** abgeleitet und verwenden ausschliesslich die geerbten Operationen. Der Ansatz ist unserem Ansatz ähnlich, indem er auf Wiederverwendbarkeit setzt, unterscheidet sich abe r, indem der Algorithmus von der Datenstruktur erbt, wo wir eine polymorphe Client -Beziehung verwen- den. Für genau dieses spezifische Problem (Sortieralgorithmen) ist die graphische Darstellung besser als in unserem Framework. Unsere Darstellung ist aber viel allgemeiner, wie in Kapitel 7 über Design-Entscheide dargelegt.

The JDSL Visualizer

Der JDSL Visualizer ist ein Visualisierungstool für die JDSL (The Java Data Structures Library), welches nach dem gleichen Prinzip aufgebaut ist wie unser Framework: Animation der Datenstrukturen (JDSL) und damit indirekte Animation der Algorithmen. JDSL kann als Binärdatei gratis bezogen werden; das Visualisierungstool kann als Binärdatei oder als Quellcode gratis heruntergeladen werden. Leider funktionieren aber beid e Installationen des Visualizers nicht, und zwar aufgrund fehlender Klassen, also aufgrund von Fehlern, die man nicht selbst beheben kann. Die Mitglieder der Supportgruppe antworten leider nicht auf e - mails; es macht den Anschein, als würde das Projekt nic ht mehr weiterverfolgt und das Tool nicht mehr gepflegt. Das ist schade, denn gemäss der Information auf der Homepage wurden in diesem Projekt alle Probleme bereits gelöst, die wir nun nochmals gelöst haben, und darüberhinaus vieles, was bei uns noch reine s Wunschdenken ist: mehr Interaktion mit dem Benutzer und Interaktion zwischen verschiedenen Datenstrukturen. Leider war es aufgrund der beschriebenen Umstände nicht möglich, von den Erfahrungen dieser Gruppe zu profitieren.

B. Literaturverzeichnis

Zu dieser Semesterarbeit

[Cieliebak01] Projektausschreibung Animierte Datenstrukturen in Java als Diplomarbeit oder Semesterarbeit, Ende Sommersemester 2001 im Internet veröffentlicht und im Informatikgebäude der ETHZ (IFW) ausgehängt. Erhältlich bei Mark Cieliebak oder Peter Häfliger.

[Häfliger01] Projektplan Animierte Datenstrukturen in Java, 2. Version vom 25. Oktober 2001. Erhältlich bei Peter Häfliger.

Zu Algorithmen und Datenstrukturen

[NieHin99] Jürg Nievergelt, Klaus H. Hinrichs: Algorithms & Data Structures with Applications to Graphics and Geometry, vdf Hochschulverlag, Zürich, 1999

[OttWid02] Thomas Ottmann, Peter Widmayer: Algorithmen und Datenstrukturen, 4. Auflage, Spektrum Verlag, Heidelberg, Berlin, Oxford, 2002

[Wirth86] Niklaus Wirth: Algorithmen und Datenstrukturen mit Modula-2, 4., neubearbeitete und erweiterte Auflage, Teubner Verlag, Stuttgart, 1986

Zu Java

[Eckel00] Bruce Eckel: Thinking in Java, 2nd edition, 2000, http://www.planetpdf.com

Zur Software-Entwicklung

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995

[Meyer97] Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997

[StePoo01] Perdita Stevens, Rob Pooley: UML – Softwareentwicklung mit Objekten und Komponenenten, Pearson Studium, 2001, deutsche Übersetzung von UML – Softwareengineering with Objects and Components, 2nd edition, Pearson Education Limited, 2000

[Szy97] Clemens Szyperski: Component Software – Beyond Object-Oriented Programming, Addison-Wesley, 1997

C. Softwaretools und Internet-Links

Das in diesem Dokument beschriebene Animationsframework

Auf der Homepage des Betreuers Mark Cieliebak: http://www.inf.ethz.ch/personal/cielieba

Programmiersprache und API

Java 2 SDK Standard Edition, Version 1.3.1_01: http://www.java.sun.com

Entwicklungsumgebung

eclipse: open source IDE framework, entwickelt von OTI: http://www.eclipse.org

CASE tool

Poseidon for UML (vormals ArgoUML), Community Edition: volles CASE tool, in dieser Arbeit aber nur zur Diagramm-Generierung verwendet: http://www.gentleware.com

Andere Simulationstools für Algorithmen und Datenstrukturen

xtango:	http://www.cc.gatech.edu/gvu/softviz/algoanim/xtango.html
SortAnimation:	http://www.inf.ethz.ch/~staerk/ algorithms/SortAnimation.html
JDSL Visualizer:	http://www.cs.brown.edu/cgc/jdsl/explorations/jdslviz/jdslviz.html

Datenstrukturen in Java

JDSL:	http://www.cs.brown.edu/cgc/jdsl/