# Layoutspezifikationen für komplexe graphische Objekte

Ralf Möller und Volker Haarslev†
Universität Hamburg, Fachbereich Informatik, Bodenstedtstrasse 16, D-2000 Hamburg 50
†z.Z. Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
Email: moeller@rz.informatik.uni-hamburg.dbp.de, haarslev@parc.xerox.com

**Zusammenfassung:** Dieser Beitrag stellt einen neuen Ansatz zur Anordnungsbeschreibung allgemeiner zweidimensionaler Objekte vor, der im Rahmen eines Unterstützungssystems zur Visualisierung objektorientierter Programmsysteme implementiert wurde. Wichtige Bestandteile dieses Systems sind erweiterbare Bausteine, die schon während der Programmentwicklung die Erstellung von Visualisierungen unterstützen. Die geometrische Anordnung von Objekten kann mithilfe einer allgemeinen deklarativen Beschreibung spezifiziert werden, die an das Boxmodell des TEX-Satzsystems angelehnt wurde. Aufbauend auf dieser Grundlage werden weiterführende Anordnungsmöglichkeiten vorgestellt.

## 1 Einleitung

Eine angemessene Visualisierung objektorientierter Systeme ist aufgrund ihrer Komplexität ein äußerst schwieriger Vorgang, der leistungsfähige Werkzeuge erfordert. Die Eignung entsprechender Werkzeuge kann daran gemessen werden, inwieweit sie strukturelle und konzeptionelle Visualisierungen unterstützen. Nachfolgend stellen wir einige grundlegende Konzepte eines Ansatzes vor, der beide Formen der Visualisierung unterstützt. Diese Konzepte bilden die Grundlage für ein objektorientiertes Visualisierungssystem, das im Kern aus erweiterbaren Bausteinen besteht. Es ist in Allegro Common Lisp[1] auf einem Apple Macintosh implementiert und baut auf eine im Xerox Palo Alto Research Center entwickelte Implementation (PCL) von CLOS (Common Lisp Object System [9]) auf. Die Leistungsfähigkeit unseres Ansatzes wird am Beispiel einer flexiblen Benutzungsoberfläche für einen CLOS „Browser/Inspector“ demonstriert. Weitergehende Anwendungen (z.B. Graphikeditor, Visualisierung von Einschränkungsnetzen) sowie detailliertere Ausführungen sind in [13] zu finden.

Visualisierungen, die sich an den Konzepten von Anwendungsdomänen orientieren, müssen meistens noch „von Hand“ erstellt werden, da die zur Erstellung einer konzeptionellen, d.h. problemnahen Visualisierung nötige geometrische und graphische Information im allgemeinen Fall nicht aus den zu visualisierenden Daten gewonnen werden kann. Dieses ist insbesondere der Fall, wenn die für eine konzeptionelle Visualisierung eines Algorithmus' benötigte Zusatzinformation über die Anwendungsdomäne für diesen Algorithmus nicht relevant und daher überhaupt nicht modelliert ist. Ein Beispiel hierfür ist die Programmvisualisierung eines Sortierverfahrens in Form einer Animation durch Verschiebung von Plättchen [3].

Eine strukturelle Interpretation dagegen betrachtet die Datenstruktur und versucht eine Visualisierung mithilfe geometrischer Information, die aus dieser Struktur gewonnen wurde. Der Nachteil dieser Interpretation besteht darin, daß sich die intendierte (konzeptuelle) Interpretation eines Programmierers (z.B. einer Datenstruktur) meist nur indirekt durch die in die Visualisierung übernommenen Bezeichner des Programms widerspiegelt. Die strukturelle Interpretation von Prozessen orientiert sich häufig am verwendeten Programmierstil bzw. dessen Verarbeitungsmodell [14]. So finden sich für imperativ ausgelegte Systeme zur Visualisierung des Kontrollflusses Flußdiagramme (flow-charts) oder Nassi-Shneiderman-Diagramme. Für relationale oder auch logische Programmiersysteme wurden Darstellungen wie „Transparent Prolog Machine“ [4] entwickelt. Datenflußdarstellungen u.a. für funktionale Systeme

sind z.B. in VIPEX [5, 6], Pluribus [16] und Prograph [12] zu finden. Beschränkungen (constraints) sind im ThingLab-System [2] graphisch darstellbar.

Zur Unterstützung der Programmentwicklung erscheint es sinnvoll, sowohl strukturelle als auch konzeptionelle Visualisierungen zu verwenden. Dafür werden geeignete vordefinierte Bausteine und Konstruktionsmethoden benötigt (siehe hierzu auch [8]). Unser System gestattet es dem Benutzer, vordefinierte Bausteine zu verwenden bzw. zu modifizieren sowie eigene zu integrieren. Eine Forderung dabei ist, daß vom Benutzer erstellte Bausteine leicht in anderen Kontexten wiederverwendbar sein sollen. Ein Grundproblem in der Visualisierung von zusammengesetzten Bausteinen besteht in der flexiblen Anordnung (Layout) ihrer Teilkomponenten. Das Layout spielt für eine angemessene ästhetische Darstellung eine wichtige Rolle. In formularorientierten Darstellungen muß sich die Größe und Lage der Teilkomponenten nach dem insgesamt zur Verfügung stehenden Platz richten. Die Anordnung der Teilkomponenten wird in diesem Fall durch globale Beschränkungen beeinflußt. Auch in graphischen Darstellungen wie Regelnetzen oder Darstellungen von Objekthierarchien werden die Graphknoten nach bestimmten Kriterien (z.B. als Baum oder Graph) angeordnet. Hier liegt eine lokale Beschränkung vor: die Lage der Kanten im Graph ist von der Lage der jeweils angrenzenden Knoten abhängig[2].

Dieser Beitrag beschreibt in den nächsten Abschnitten, wie Layouts für graphische Objekte spezifiziert werden können. Darauf aufbauend werden Sichtbereiche und deren Elemente vorgestellt, die den Kontext zur Visualisierung von Objekten definieren. Wir führen das Konzept von Referenzen zwischen Sichtbereichselementen ein und zeigen am Beispiel von gerichteten azyklischen Graphen eine Anwendung dieser Konzepte. Die beiden darauffolgenden Abschnitte erläutern die den Spezifikationen zugrundeliegenden Anordnungsalgorithmen und deren Ausnutzung zur Optimierungen der graphischen Ausgabe. Dieser Beitrag schließt mit der Einordnung unseres Ansatzes in die Forschungslandschaft und gibt einen Ausblick auf mögliche Erweiterungen.

## 2 Layoutangaben

Das zugrundeliegende Modell der Anordnung von Objekten ist in Anlehnung an das „Box-and-glue-Modell“ des TEX-Satzsystems konzipiert [10]. Ein (rechteckiger) Bereich kann als eine Box aufgefaßt werden, deren Ausmaße festgelegt sind, in der aber Elemente in bestimmter Weise horizontal bzw. vertikal anzuordnen sind. Als konkretes Beispiel für eine Box wäre ein Dialogfenster zu nennen, in dem Dialogelemente anzuordnen sind (s.u.). Grundsätzlich ist jedoch die Verwendung von Layoutmustern nicht auf Dialoge bzw. Fenster beschränkt. Jedes Objekt kann als Box interpretiert werden. Die Layoutalgorithmen evaluieren generische Funktionen (z.B. `box-items` für Boxen oder `box-item-position` für Boxelemente). Soll nun ein Objekt als Box oder als Boxelement interpretiert werden, so müssen geeignete Methoden für diese generischen Funktionen definiert werden. Die Interpreterfunktionen bleiben unverändert. Durch Verwendung dieses abstrakten Protokolls ist außerdem eine leichtere Portierung der Layoutfunktionen gewährleistet.

Ein Layoutmuster ist zunächst eine Liste mit besonderen Einträgen: Schlüsselwörtern, Distanzangaben, Boxelementen oder auch geschachtelten Boxen. Mögliche Angabeformate zu Distanzen zwischen Elementen sind absolute Abstände (in Pixeln), Abstände relativ zur Gesamtbreite bzw. Gesamthöhe der umschließenden Box (prozentuale Angabe aus [0.0, 1.0]) sowie auch Angaben, eine bestimmte Distanz einfach mit dem restlichen zur Verfügung stehenden Platz aufzufüllen. Letztere Angaben werden als Füller (filler) bezeichnet. Sind mehrere Füller in einer Dimension (horizontal oder vertikal) angeordnet, so wirkt jeder Füller wie eine Feder. Zwischen den Federn stellt sich ein Gleichgewicht ein, d.h. jeder Füller ist gleich lang. Um „zu kleine“ bzw. „zu große“ Füllabstände zu vermeiden, können Füllangaben mit Minimal- und Maximalwerten bzgl. ihrer Ausdehnung versehen werden.

### 2.1 Layoutmuster für Boxen

Ein Layoutmuster der Form `(:vbox (:width` *h* `:height` *v*`)` *box-specifier-1 box-specifier-2 ...*`)` erlaubt die vertikale Anordnung von Boxelementen. Distanzangaben in einem vertikalen Layoutmuster werden dabei als vertikale Strecken zwischen den Boxelementen interpretiert. Für horizontale Boxen, die

durch Angabe des Layoutmusters `(:hbox (:width` *h* `:height` *v*`)` *box-specifier-1 box-specifier-2 ...*`)` definiert werden, sind Distanzangaben entsprechend horizontale Strecken. Relative Größenangaben der Boxen beziehen sich – auch bei der Größenangabe einer Box durch `(:width` *h* `:height` *v*`)` – auf die Breite bzw. Höhe der jeweils umschließenden Box. Sowohl die Höhen als auch die Breitenangabe einer Box ist optional; bei fehlender Angabe wird für *h* bzw. *v* (s.o.) ein Füller eingesetzt.

Die in einem horizontalen oder vertikalen Boxmuster aufgeführten Boxelemente werden nicht in ihrer Größe verändert. Reicht etwa der Platz in einer Box nicht aus, so stehen die Boxelemente über den Rand der Box hinaus. In einigen Anwendungen ist es jedoch erforderlich, daß sich die Größe eines einzelnen Elementes nach der Größe der umschließenden Box richtet. Das zugehörige Layoutmuster einer solchen „Rahmenbox“ (frame-box) hat die Form `(:fbox (:width` *h* `:height` *v*`)` *item*`)`. Die Größe von *item* wird so gesetzt, daß der gesamte für diese Box zur Verfügung stehende Raum von *item* ausgenutzt wird. In einem Rahmenboxmuster tritt daher nur ein Boxelement (item) auf.

## 2.2 Layoutmuster für Füller

Angaben zu Füllabständen haben folgende vollständige Form: `(:filler :min` *m* `:max` *n*`)`, `:min` und `:max` sind optional. Für das Füllmuster `(:filler :min 0 :max` *box-size*`)` steht `:filler` als Kurzform. Wenn statt einer Ganzzahl für *m* oder *n* das Schlüsselwort `:as-needed` angegeben ist (z.B. `(:filler :min :as-needed)`), so berechnet das System eine angepaßte Mindest- bzw. Maximalgröße anhand der Größe der Boxelemente und der Abstandsangaben, die innerhalb des Boxmusters auftreten. Abbildung 1 zeigt als Beispiel eine Layoutspezifikation und den daraus erzeugten Dialog, der im Rahmen eines CLOS-Inspektors zur Visualisierung der Klassenhierarchie dient (siehe auch Abschnitt 3.2).

```
(let
 ((left-table (make-dialog-item …))
  (right-table (make-dialog-item …))
 (make-layouted-dialog
  :layout
   (:vbox (:width :filler
           :height :filler)
    (:hbox (:height 1/4
            :width :filler)
     (:fbox () left-table)
     (:fbox () right-table)))))
```

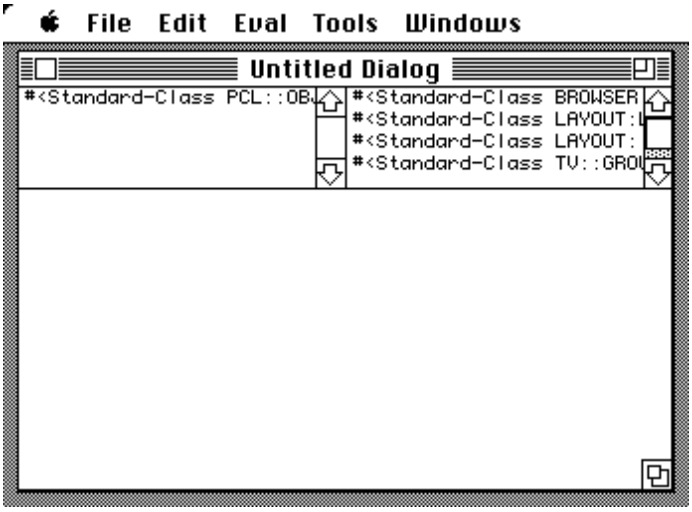


Abbildung 1: Layoutspezifikation und entsprechender Dialog (schematisch).

Für die Tabellen `left-table` und `right-table` müßten noch Methoden zur Behandlung der Mausklicks definiert werden. Dieses sei hier vernachlässigt. Der Dialog wird als `:vbox` angeordnet. Im oberen Viertel befindet sich eine `:hbox`, in der sich wiederum zwei mit einer `:fbox` angeordnete Tabellen befinden. Jede der Rahmenboxen erhält durch die `:filler` Angaben die Hälfte des Platzes. Durch die `:fbox` Muster werden wiederum die Tabellenpositionen und -größen bestimmt. In der unteren Darstellungsfläche ist Platz für einen Graphen. Dieser soll jedoch nicht speziell konstruiert werden, sondern aus vorgefertigten Bauteilen und Konstruktionsprinzipien hervorgehen. Diese werden in den nächsten Kapiteln geschildert.

# 3 Sichtbereiche und Sichtbereichselemente

Für problemnahe, konzeptionelle Visualisierungen reichen Standardinteraktionsobjekte wie Tabellen, statische und editierbare Texte oder auch Schaltflächen nicht aus. Es werden beliebige, zunächst zweidimensionale graphische Objekte benötigt. Um den Erstellungsaufwand gering zu halten, erweist es sich als vorteilhaft, auch hier vorgefertigte Bausteine (graphische Objekte) kombinieren zu können. Graphische Objekte werden (prozedural) durch eine Zeichenfunktion definiert. Wann und unter welchen Bedingungen diese Zeichenfunktion evaluiert wird, ist durch ein Verwaltungssystem geregelt. Graphische Objekte werden innerhalb eines Teilrechtecks eines Fensters gezeigt. Dieses Teilrechteck heißt Sichtbereich und definiert in einem eigenen Koordinatensystem einen Kontext, in dem graphische Objekte, sog. Sichtbereichselemente, gezeigt werden. Sichtbereichselemente können in einen Sichtbereichskontext eingefügt oder aus ihm entfernt werden. Die hierzu notwendige Verwaltung wird durch den Sichtbereich übernommen. Die Aufgaben eines Sichtbereichs umfassen: Rollen der Darstellungsfläche, Verschiebung, Markierung und Gruppierung von Sichtbereichselementen sowie eine Erzeugung von elementbezogenen Interaktionsereignissen (z.B. bei Mausklicks). Die Algorithmen brauchen also nicht für jede Visualisierung neu implementiert zu werden. Weitere über diese Standardaufgaben hinausgehende Funktionen der Sichtbereiche bzw. Sichtbereichselemente werden in den nächsten Abschnitten geschildert.

Die Eigenschaften von selbstdefinierten Sichtbereichselementen lassen sich durch Verwendung vordefinierter (Super-)Klassen geeignet zusammenstellen. Hierzu werden Klassen benötigt, die Sichtbereichselemente maussensitiv, beweglich oder markierbar machen. Das Verwaltungssystem evaluiert bei Bedarf generische Zeichen- und Löschfunktionen für aktuell sichtbare Elemente.

Graphknoten für den obigen Dialog lassen sich als spezielle Sichtbereichselemente (view-items) realisieren.

```
(defclass label-view-item (moveable-view-item-mixin view-item)
  ((label :initarg :label :accessor label)))
```

Die Basisklasse aller Sichtbereichselemente `view-item` wird mit einer Zusatzklasse `moveable-view-item-mixin` zu einer neuen Klasse `label-view-item` kombiniert. Zunächst ist für die Klasse `label-view-item` noch das „Aussehen“ der Instanzen zu bestimmen. Das System evaluiert zum Zeichnen eines Elements eine generische Funktion (`view-item-draw`). Für spezielle Klassen von Elementen (oder für ein spezielles Element) läßt sich das gewünschte Verhalten durch Definition einer `:after` Methode bestimmen.

```
(defmethod tv:view-item-draw :after ((item label-view-item) view dialog)
  (let ((position (view-item-position item))
        (size (view-item-size item)))
    (frame-round-rect dialog standard-gcontext …)
    (move-to dialog …)
    (draw-string dialog standard-gcontext (label item))))
```

Die Zeichenfunktion wird in einem Kontext mit entsprechenden Koordinatentransformationen evaluiert. Durch Position und Größenvektor eines Sichtbereichselements ist ein sog. Zeichenrechteck definiert (Abbildungen 2 und 3). Das System setzt den Klippbereich während der Evaluierung der Zeichenfunktion so, daß nur innerhalb dieses Rechtecks gezeichnet werden kann. In CLOS ist es möglich, alle erforderlichen Parameter einer Methode zu spezialisieren. Dieses kann bei der obigen Methode sehr schön ausgenutzt werden, um z.B. eine kontextabhängige Zeichenfunktion (Spezialisierung von `view`) oder eine anwendungsabhängige Zeichenfunktion (Spezialisierung von `dialog`) zu definieren.

In prototypischen Anwendungen ist die Angabe einer Zeichenmethode schon ausreichend, um ein Sichtbereichselement einzuführen. Ist es nötig, ein Element zu löschen (z.B. bei Verschiebung oder Entfernung eines Elements aus dem Sichtbereich), wird vom System das Zeichenrechteck gelöscht. Hierdurch zerstörte Elemente werden erneut gezeichnet. Dadurch wird in vielen Fällen allerdings mehr gelöscht und neu gezeichnet als nötig (Blitzeffekt). Durch die objektorientierte Architektur der Sichtbereichsverwaltung lassen sich jedoch Verwaltungsfunktionen wie die Löschfunktion eines Elements anpassen (z.B. auch das „Aussehen“ eines Elements bei einer Verschiebung) und auf spezielle Sichtbereichselemente zuschneiden. Genauere technische Angaben sind [13] zu entnehmen.

### 3.1 Referenzen zwischen Sichtbereichselementen

Nachdem im vorigen Abschnitt die allgemeine Verwaltung von Sichtbereichselementen erläutert wurde, soll jetzt auf die Spezifikation von Beziehungen zwischen Sichtbereichselementen eingegangen werden. Jedem Sichtbereichselement kann eine Menge von Referenzen zu anderen Sichtbereichselementen zugeordnet werden. Eine Referenz wird als deklarative Referenzbeschreibung angegeben; die Referenzpunktkoordinaten werden hieraus automatisch berechnet. Ein Sichtbereichselement zur Visualisierung einer Kante in einem Graphen könnte durch Referenzen zu zwei Knotensichtbereichselementen bestimmt werden. In der Zeichenmethode der Kante können die Referenzpunktkoordinaten erfragt werden. Zwischen den Punkten wird dann z.B. eine Linie gezeichnet. Die Syntax zur Definition von Referenzpunkten orientiert sich an dem bisher schon eingeführten Boxmodell; eine Box zur Referenzpunktdefintion heißt Referenzbox (kurz: `:rbox`).

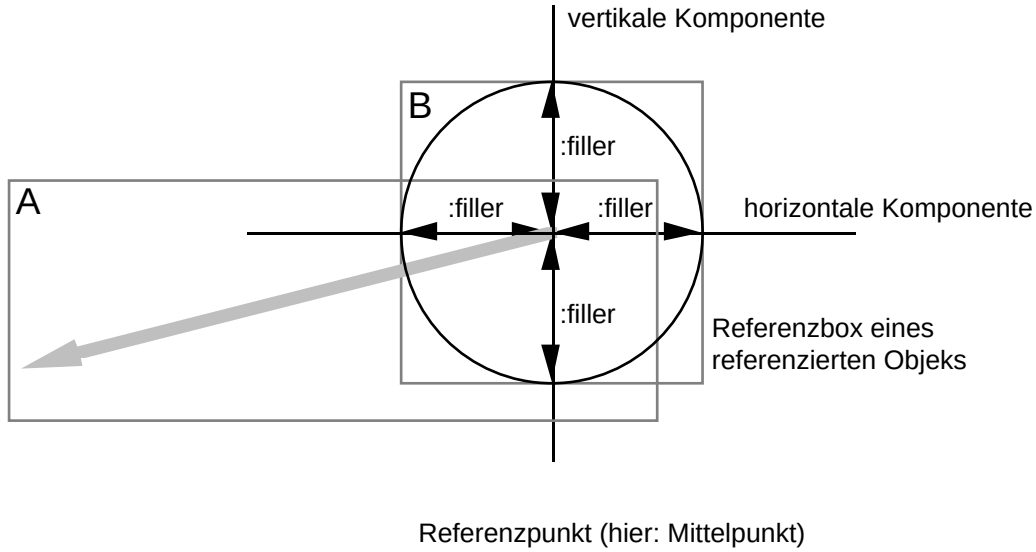


Abbildung 2: Referenzmodell. Ein Objekt A (Pfeil) sei durch einen Referenzpunkt bzgl. des Mittelpunkts eines Objekts B (Kreis) definiert. Die Zeichenrechtecke der beiden Figuren sind durch graue Rechtecke angedeutet.

Ein Referenzpunkt wird definiert durch Angabe einer horizontalen und einer vertikalen Beschreibungskomponente, d.h. einer Liste, eingeleitet durch das entsprechende Schlüsselwort `:horizontal` bzw. `:vertical`. In der jeweiligen Dimension wird nun die Lage der entsprechenden Referenzpunktkomponenten beschrieben. Die Beschreibungsform von Abständen ist analog zu der bei anderen Boxen. Es können relative, absolute und zu füllende Abstände definiert werden. Dabei wirken Füllelemente wiederum wie Federn. Anstelle der in anderen Boxen anzuordnenden Boxelelemente wird hier das Schlüsselwort `:reference` verwendet. Der in der Abbildung 2 von einem Objekt A (ein Pfeil)[3] referenzierte Mittelpunkt eines Objekts B (ein Kreis) wird wie folgt angegeben:

```
(layout-description (:rbox A B
                           (:horizontal :filler :reference :filler)
                           (:vertical :filler :reference :filler)))
```

Die Referenzbox des Objekts B ist in Abbildung 2 grau dargestellt. Sie ergibt sich aus der Position und dem Größenvektor des Sichtbereichselements B, d.h. aus dem Zeichenrechteck von B. Die Füller wirken wie Federn und „drücken“ den Referenzpunkt in die Mitte. Soll der Referenzpunkt drei Pixel vom unteren und rechten Rand entfernt liegen, so kann dieses erreicht werden, indem folgendes Muster als Argument der Funktion `layout-description` angegeben wird.

```
(layout-description (:rbox A B
                           (:horizontal :filler :reference 3)
                           (:vertical :filler :reference 3)))
```

Ist der durch relative oder absolute Abstandsangaben definierte Platz größer als der entsprechend der Referenzbox zur Verfügung stehende, so ist die Länge der Füller gleich null. Der Referenzpunkt kann also durchaus aus der Referenzbox herausragen.

Ein weiteres Beispiel für eine Klasse von speziellen Sichtbereichselementen verdeutlicht die Verwendung von Referenzen während des Zeichnens eines Sichtbereichselementes. Kanten in einem Graphen sind mithilfe von Referenzpunkten sehr einfach ihren Anfangs- und Endknoten zuzuordnen (Abbildung 3). Eine Zeichenmethode für Kanten (Klasse: `line-view-item`) erfragt die Referenzen (`references-of-this-item`) und ermittelt anhand dieser die Anfangs- und Endposition der Linie (`reference-position`):

```
(defclass line-view-item (view-item)
  ())

(defmethod view-item-draw :after ((item line-view-item) view dialog)
  (let* ((references (references-of-this-item item))
         (p1 (reference-position (first references)))
         (p2 (reference-position (second references))))
    (move-to dialog p1)
    (line-to dialog standard-gcontext p2)))
```

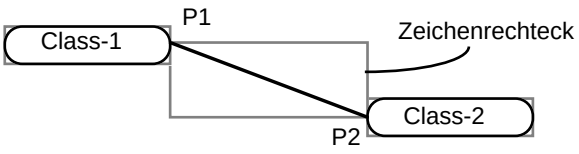


Abbildung 3: Positionierung von Kantenelementen mithilfe von Referenzen.

Referenzen können auch verwendet werden, um eine spezielle Löschfunktion für Linien anzugeben. Es wird nur die Linie selbst gelöscht.

```
(defmethod view-item-undraw ((item line-view-item) view dialog
                             position size references)
  (using-gcontext ((eraser-gcontext :pen-pattern *white-pattern*))
    (let ((p1 (reference-position (first references)))
          (p2 (reference-position (second references))))
        (move-to dialog p1)
        (line-to dialog eraser-gcontext p2)))))
```

Werden die gerundeten Rechtecke verschoben, so werden die Referenzpunkte `P1` und `P2` der Linie neu berechnet und die Linie wird neu gezeichnet. Wird ein referenzierendes Objekt (hier die Linie) explizit verschoben, so verschieben sich die Referenzpunkte entsprechend. Die Ansatzpunkte der Linie liegen also ggf. nicht mehr an den Seiten der gerundeten Rechtecke.[4] Referenzen können auch zur Plazierung von Sichtbereichselementen verwendet werden (auch die Linie wurde dadurch „plaziert"), wenn das Aussehen unabhängig von den Referenzpunkten ist.

### 3.2 Definition von Layoutbeschreibungen für spezielle Layouts

Sichtbereiche können ohne Aufwand mit anderen Dialogelementen wie etwa Schaltflächen oder Tabellen kombiniert werden. Vererbungsnetze sind in der Regel weitaus größer als die zur Verfügung stehende Darstellungsfläche. Es ist notwendig, sich durch Kontrollmechanismen einen Teilausschnitt auszuwählen. Kontrolle und Visualisierung sind also gemeinsam zu betrachten. Ein Doppelklick auf einen Eintrag der rechten Tabelle des Dialogs aus Abbildung 1 könnte z.B. eine Verschiebung der Tabelle nach links und ein Zeigen der Subklassen der selektierten Klasse in der freiwerdenden Tabelle zur Folge haben. Entsprechend würde die linke Tabelle nach rechts verschoben werden und dann die Superklassen in der linken Tabelle gezeigt werden. Auf Bedarf könnte ein Vererbungsnetz der Subklassen der selektierten Klasse angezeigt werden. Die Expansion und Verschiebung der Klassen in den Tabellen dient also der Fokussierung. Ein Vererbungsnetz läßt sich in Form eines gerichteten azyklischen Graphen darstellen und stellt ein Beispiel für ein Layoutmuster dar, das sich nicht an dem Boxmodell orientiert. Selbstdefinierte Layoutmuster können durch folgende Form definiert werden:

```
(deflayout :dag (roots successor-fcn expansion-depth expansion-predicate
                 node-generator edge-generator
                 left-edge-reference-fcn right-edge-reference-fcn)
  "Returns the (generated and) layouted items."
```

```
  (dag-pattern-interpretation …))
```

Selbstdefinierte Layoutbeschreibungen können innerhalb des speziellen Layoutmuster `(:gbox ...)` in das Boxenschema integriert werden. Anstelle der Punkte steht dann das selbstdefinierte Layoutmuster. Die benötigte Größe einer `:gbox` kann dann automatisch berechnet werden. Ein Layout für einen gerichteten azyklischen Graphen (DAG) läßt sich auf einfache Weise integrieren. Das Beispiel aus Abbildung 1 läßt sich mit einem DAG-Layoutmuster-Interpretierer vervollständigen (Abbildung 4).[5]

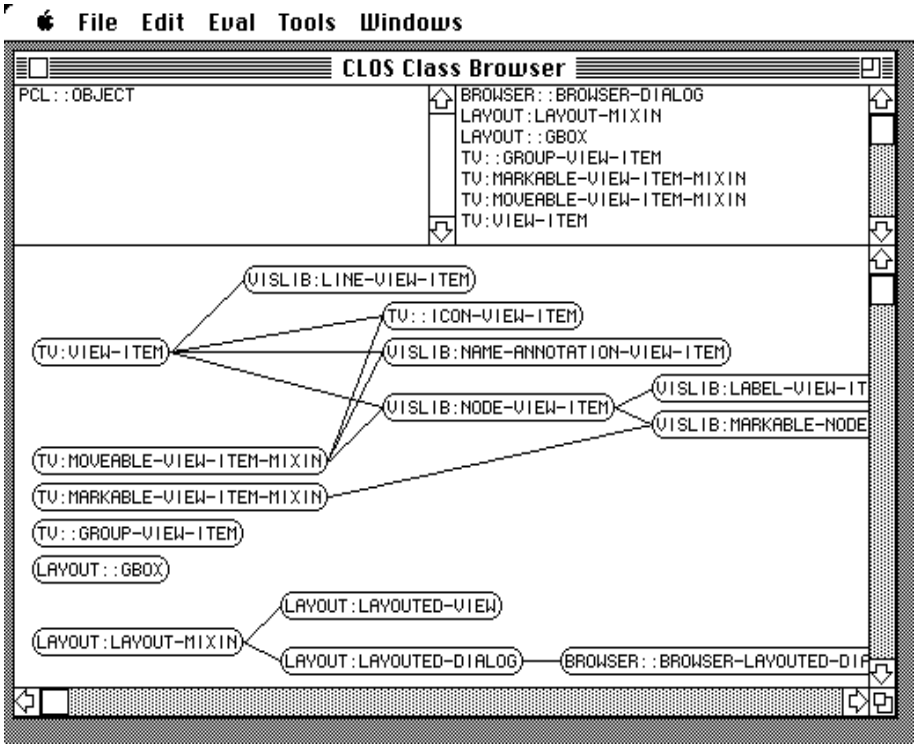


Abbildung 4: Visualisierung einer Vererbungshierarchie.

```
(defun class-browser (class-name)
  (let ((left-table …)
        (right-table …)
        (graph-view (make-layouted-view  :scroll-bars ':both
                                         :auto-scrolling t)))
    (make-layouted-dialog …
                          :layout (:vbox ()
                                         (:hbox (:height 1/4)
                                                (:fbox () left-table)
                                                (:fbox () right-table))
                                         (:fbox () graph-view)))
    (setf (layout graph-view)
          (:vbox ()
           10      ; 10 Pixel oberer Abstand
           (:hbox ()
            10     ; 10 Pixel linker Abstand
            (:gbox
             (:dag (list (find-class class-name))  ; Menge der Wurzeln des DAGs
                   #'pcl::class-direct-subclasses  ; Nachfolgerfunktion
                   *hierarchy-depth*      ; max. Expansionstiefe
                   #'(lambda (class) t)   ; Expansionsprädikat
                   #'(lambda (class)      ; Erzeugungsfunktion für
                       (make-label        ; Knotensichtbereichselement
                                          ; (sog. Knotenfunktion)
                        (class-name-as-string class)))
                   #'make-line-view-item  ; sog. Kantenfunktion
                   #'western-reference
                   #'eastern-reference)))))
    class-name))

(defun western-reference (referencing-object referenced-object)
```

```
  "Referenzpunktbestimmung durch :rbox Muster."
  (:rbox referencing-object
         referenced-object
         (:vertical :filler :reference :filler)
         (:horizontal :reference :filler)))

(defun eastern-reference (referencing-object referenced-object)
  "Referenzpunktbestimmung durch :rbox Muster."
  (:rbox referencing-object
         referenced-object
         (:vertical :filler :reference :filler)
         (:horizontal :filler :reference)))
```

Falls der Graph als Übersicht dienen sollte, so würden die Knoten evt. als Kreise ausgebildet und die obigen Referenzmuster dahingehend abgeändert, daß die jeweiligen Ansatzpunkte der Kanten in der Knotenmitte zu liegen kommen. Durch Verwendung von generischen Nachfolger-, Knoten- und Kantenfunktionen lassen sich mit einem `:dag`-Layoutmuster verschiedene anwendungsspezifische Darstellungsformen erzeugen. Das Beispiel zeigt, wie Layoutbeschreibungen auf zwei Ebenen analog verwendet werden. Auf der Ebene der Dialogelemente steht das Boxmodell zur Layoutbeschreibung zur Verfügung. Das Grundlayout wird bei einer (interaktiven) Vergrößerung eines Dialogfensters aufrechterhalten. Auf der Ebene der Sichtbereichselemente können zusätzlich Referenzen und selbstdefinierte Layoutmuster verwendet werden.

## 4 Anordnungsalgorithmen

Mit den bisherigen Beschreibungsformen lassen sich beliebige Objekte innerhalb eines rechteckigen, geschachtelten Schemas anordnen. Dialogelemente wie Schaltflächen oder Sichtbereiche können innerhalb einer vorgegebenen Flächenaufteilung angeordnet werden. Wichtig ist außerdem, daß die Größe der angeordneten Elemente sich auch nach der für ein Element vorgesehenen oder verfügbaren Fläche richten kann. In diesem Zusammenhang bieten Minimal- und Maximalangaben eine zusätzliche Flexibilität. Dieses sei hier noch einmal an einem Beispiel eines Inspektors für CLOS-Klassen genauer betrachtet. Der Inspektordialog wurde mit dem in diesem Beitrag vorgestellten System erstellt. In Tabellenform werden alle für eine Klasse relevanten Informationen wie Subklassen, Superklassen, Einträge (slots) und für die Klasse definierte Methoden dargestellt.

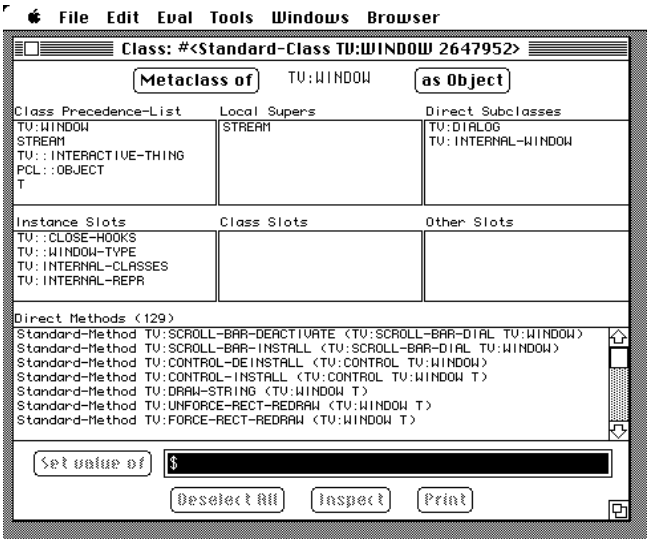


Abbildung 5: CLOS-Klasseninspektor-Dialog.

Die Tabellen können nach Bedarf gerollt werden, sofern der für eine Tabelle zur Verfügung stehende Platz nicht ausreicht, um alle Tabelleneintrage gleichzeitig anzuzeigen. Abbildung 5 vermittelt einen Eindruck über die Standardaufteilung der Darstellungs- und Schaltflächen eines Klasseninspektordialogs.

Die Layoutbeschreibung ist aus mehreren Teilen zusammengesetzt und zu umfangreich, um in diesem Rahmen geschildert zu werden. Wichtig ist hier nur folgendes: die obere Tabellenzeile ist durch eine horizontale Box (`:hbox`) realisiert, in der drei Rahmenboxen (`:fbox`) mit jeweils einer Tabelle angeordnet sind. Die Breite und Höhe der Rahmenboxen ist jeweils durch das Symbol `:filler` beschrieben; es soll also ein Füllraum ausgefüllt werden. Das bedeutet: innerhalb der horizontalen Dimension in den horizontalen Boxen konkurrieren drei Füllangaben um den zur Verfügung stehenden Platz. Die Füller reagieren wie Federn, d.h. nach einem „Einschwingvorgang“ erhält jede Füllangabe ein Drittel des zur Verfügung stehenden Platzes.

Die Höhe der Rahmenboxen ist durch die Höhe der umschließenden horizontalen Boxen bestimmt. Auch die mittlere Tabellenzeile und die untere Tabelle haben als Höhenangaben Füller. In der vertikalen Dimension konkurrieren also ebenfalls Füllangaben um Ausdehnungsraum. Nun wäre es sehr mißlich, auch hier eine Drittelung vorzunehmen. Die oberen Tabellen sind dünner besetzt. In diesen Anwendungsfällen werden Beschreibungen zur maximalen oder minimalen Füllerausdehnungen benötigt. Die Höhe der oberen Tabellenzeile ist zwar in weiten Teilen flexibel, kann aber durch Angabe einer maximalen Höhe zugunsten anderer Tabellen auf eine weitere Ausdehnung verzichten, wenn der Platz nicht für Tabellenelemente verwendet wird (dieses ist leicht zu ermitteln). Andererseits sollte eine Tabelle nicht zu einem Strich degradiert werden, wenn kein Eintrag enthalten ist. Dieses kann durch Angabe einer minimalen Höhen- bzw. Füllerangabe erreicht werden. Deklarative Layoutbeschreibungen dieser Art sind besonders vorteilhaft, wenn eventuell noch weitere Dialogelemente zu einem Dialog hinzukommen (z.B. eine Schaltfläche „Edit Definition“).

Unter Verwendung von Layoutbeschreibungen, wie sie oben vorgestellt wurden, werden Anordnung und Größenanpassung der in einem Dialog auftretenden Elemente automatisch durchgeführt, wenn das Dialogfenster (interaktiv) vergrößert bzw. verkleinert wird. Eine wesentliche Komponente der Beschreibungen sind die Füller. Der nächste Abschnitt enthält eine genauere Betrachtung der Berechnung von Füllelementen.

Anordnungsangaben können geschachtelt werden, d.h. in einer `:vbox` kann eine `:hbox`, in dieser wiederum eine `:vbox` auftreten. Das Federmodell für Füller gilt jedoch nur für Füller einer Schachtelungsebene. Abbildung 6 verdeutlicht den Zusammenhang.

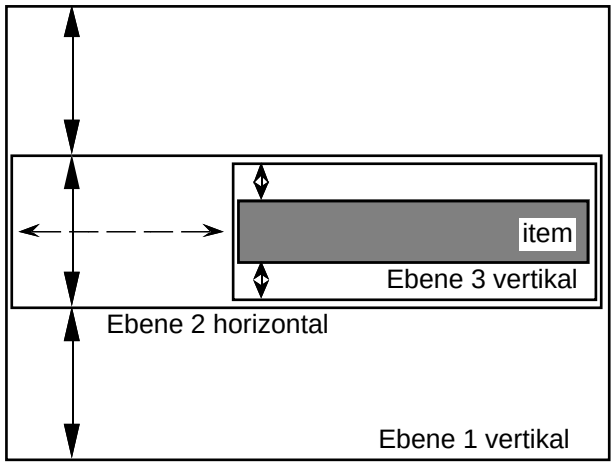


Abbildung 6: Schematische Darstellung von geschachtelten Boxen. Pfeile stellen Füllangaben dar. Füllangaben der Ebene 1 sind unabhängig von denen der Ebene 3.

Die Federn der Ebene 1 konkurrieren nicht mit denen der Ebene 3. Ansonsten müsste neben einer „Reihenschaltung“ auch noch eine „Parallelschaltung“ von Federn in die Berechnungen mit einbezogen werden. Der Berechnungsaufwand für der Länge der einzelnen Füller erhöhte sich erheblich (Gleichgewicht von Federsystemen). Außerdem wird die Benutzung der Beschreibungsmuster unnötig verkompliziert. Die „Semantik“ von Füllangaben wäre von einem Programmierer nicht einfach durchschaubar. Schon die Minimal- bzw. Maximalangaben machen es notwendig, ein Relaxationsverfahren für die Berechnung der Füllerlängen zu verwenden. Die Vorgehensweise ist allerdings noch recht anschaulich. Siehe hierzu Abbildung 7.

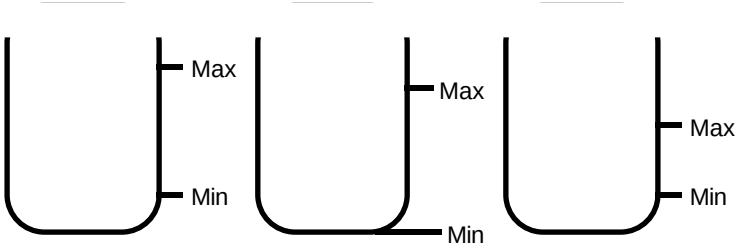


Abbildung 7: Wasserkrugmodell für Füllangaben.

Jeder Wasserkrug stellt einen Füller mit Minimal- und Maximalangaben bzgl. der Ausdehnung dar. Die Ausdehnung korrespondiert mit der Wasserfüllung. Der für alle Füller zur Verfügung stehende Platz wird durch ein externes Wasserreservoir dargestellt. Die Berechnung der Länge eines Füllers erfolgt nach folgendem Algorithmus:

– Jeder Wasserkrug wird zunächst aus diesem Wasserreservoir bis zum Minimum-Eichstrich gefüllt. Es wird sichergestellt, daß alle Krüge bis zum Minimum gefüllt sind (Restbedarf aus „Wasserleitung“[6]). Der Rest des Wasserreservoirs wird nun verteilt.

– Jedem Krug wird ein n-tel (n sei die Zahl der verbliebenen Wasserkrüge) des Wasserreservoirs zugeteilt. Durch die vorherige Auffüllung bis zum Minimum-Eichstrich sind jedoch alle Krüge unterschiedlich gefüllt. Es wird daher zunächst das minimale Füllniveau MIN aller Krüge bestimmt. Durch die 1/n-Zuteilung Z aus dem Wasserreservoir wird ein Füllniveau N = MIN + Z bestimmt. Alle Krüge werden zunächst bis zu diesem Füllniveau N gefüllt. Durch die Minimumfüllgarantie kann das spezielle Füllniveau eines einzelnen Kruges durchaus schon höher als MIN oder N liegen.

– Der zur Erreichung des Füllniveaus N nicht benötigte Teil der Zuteilung wandert zurück ins Reservoir. Wird ein Maximum-Eichstrich überschritten, so wird der Rest ins Reservoir zurückgeschüttet (der Krug sei ausreichend hoch) und der Krug wird beiseite gestellt (n <- n - 1).

– Sofern sich noch Wasser im Reservoir befindet und noch Krüge aufnahmefähig sind, beginnt der Verteilungszyklus von neuem. Der Algorithmus terminiert, wenn alle Krüge beiseite gestellt sind oder das Reservoir leer ist.

Eine Terminierung ist sichergestellt, da entweder ein Krug entfernt wird oder Wasser aus dem Reservoir verbraucht wird. Wird kein Krug beiseite gestellt, so wird in einem Iterationsschritt mindestens der Krug mit dem Füllniveau MIN mit Wasser aus dem Reservoir gefüllt. Rundungsfehler werden akkumuliert, am Ende gerundet und als Einzelzuteilung zu dem letzten Füller dazuaddiert[7].

# 5 Optimierungsalgorithmen

Durch die Realsierung eines (objektorientierten) Verwaltungssystems für allgemeine Sichtbereichselemente zusammen mit der Verwendung von Referenzen als Layoutbeschreibungen läßt sich die Reihenfolge von Zeichen– und Löschoperationen optimieren. Ein Beispiel soll dieses verdeutlichen: Die Graphknoten aus Kapitel 3.2 (Abbildung 4) sind beweglich. Wird z.B. der linke obere Knoten mit der Inschrift „TV:VIEW-ITEM“ verschoben, so müssen die referenzierenden Kanten über die Positionsänderung benachrichtigt werden. Nun ist es aber nicht erforderlich, die Layoutbeschreibung (der Kanten) neu zu interpretieren, sondern es reicht, zu den entsprechenden Referenzpunkten einen Translationsvektor zu addieren. Nur dieser Vektor wird „propagiert.“

Bei der korrespondierenden Aktualisierung der Graphik sollte nun nicht jede Kante ungeordnet neu gezeichnet werden (redundantes Zeichnen und Löschen von „später“ noch verschobenen Kanten, Ping-Pong-Effekt bei der graphischen Ausgabe). Statt die generischen Zeichenfunktionen (s.o.) direkt zu evaluieren, kann man zunächst beim „Propagieren“ der Verschiebung über die Referenzen zunächst nur Zeichen- bzw. Lösch*aufträge* in einer Art Zeichenpuffer (cache), der durch die Sichtbereiche verwaltet wird, ablegen. Nach einer Operation, die mehrere Zeichenaufträge generiert, wie etwa das Verschieben eines Graphknotens, kann der Cache gelöscht, werden d.h. nach *einmaliger* Koordinatentransformation

können zunächst alle eingetragenen Lösch- und dann die ggf. vermerkten Zeichenfunktionen evaluiert werden.

Es soll an dieser Stelle noch einmal betont werden, daß die geschilderten Verwaltungsalgorithmen nicht nur auf Graphknoten und -kanten sondern auf beliebige zweidimensionale Objekte (Sichtbereichselemente) angewendet werden können. Abbildung 8 zeigt eine problemnahe, konzeptionelle Visualisierung für einen Wegefindungsalgorithmus, der zwei Punkte verbindet, ohne eine Menge von horizontalen und vertikalen Liniensegmenten zu kreuzen. Der Algorithmus wird in diesem Beispiel jedoch eingesetzt, um (verschiebbare) Kreise zu verbinden. Für die Visualisierung sind die nicht zu kreuzenden Objekte also Kreise und keine Liniensegmente![8] Eine Darstellung der Koordinaten von Verbindung und Grenzsegmenten in einem Inspektorsystem ist aufgrund der Masse der Daten nicht interpretierbar. Die Visualisierung wurde mit geringem Aufwand mithilfe der in diesem Bericht vorgestellten Bausteine erstellt (ca. 4 Seiten Code).

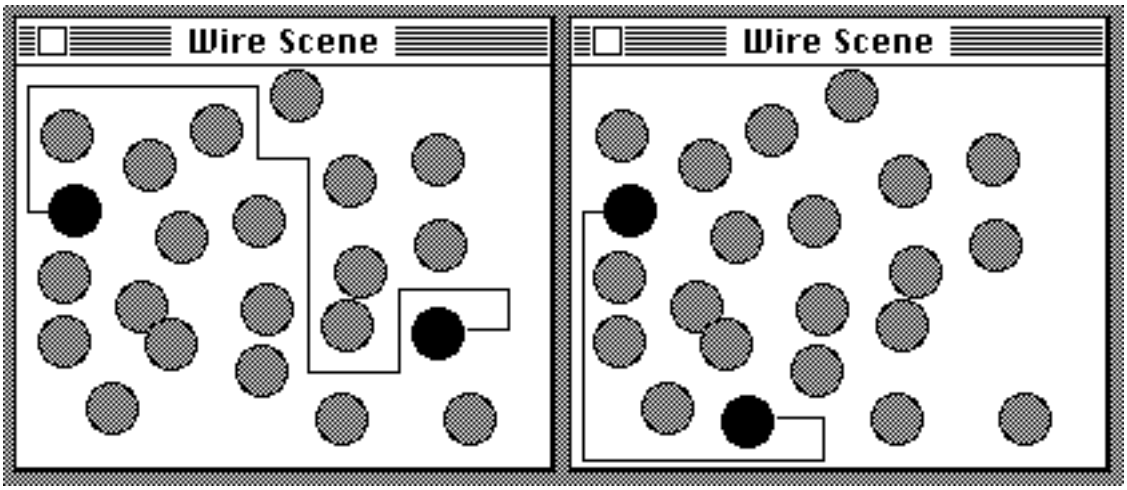


Abbildung 8: siehe Text. Der rechte schwarze Kreis wurde interaktiv verschoben.

## 6 Layoutangaben in anderen Systemen

Das zur Entwicklung von Benutzterschnittstellen konzipierte System InterViews [11] orientiert sich in Bezug auf die Anordnung von Elementen in einem Dialog ebenfalls an dem TEX-Modell mit vertikalen und horizontalen Boxen. Das in C$^{++}$ realisierte System erlaubt die Definition von Füllelementen mit maximaler und minimaler Ausdehnung und durch numerische Distanzangaben. Füller und Boxen werden hier als Objekte realisiert. Eine Ausdehnung der Boxen kann nicht festgelegt werden. Ein entsprechendes Ausdrucksmittel zu der in dieser Arbeit vorgestellten Rahmenbox (`:fbox`), die die Größe eines anzuordnenden Elements angibt, existiert u.W. im InterViews-System nicht. Selbstdefinierte Parser für Layoutbeschreibungen (`:gbox`) können in InterViews nicht auf einfache Weise integriert werden.

Layoutbeschreibungen für die Anordnung von Fenstern wurden als Teil des WISDOM-Projektes entwickelt [7]. Es existieren hier vordefinierte Klassen (Cluster) zur Anordnung von Fenstern in Form einer Sequenz, eines Stapels, eines Kreises usf. Layoutbeschreibungen werden als eigenständige Objekte realisiert. Es können nachträglich Fenster zu einem Cluster hinzugefügt werden. Füllangaben und Rahmenboxen sind in diesem System nicht vorgesehen. Dieses System ermöglicht u.W. keine Integration der Formalismen zur Anordnungsbeschreibung von Objekten innerhalb eines Fensters.

Die Symbolics-Programmierumgebung Genera[9] [15] bietet ebenfalls Mechanismen zur Bestimmung von Layouts für Fensteranordnungen.

```
(defflavor example-frame
  ()                            ; no instance variables
  (tv:bordered-constraint-frame) ; superclass
  :settable-instance-variables
  (:default-init-plist
   :panes '((pane-1 tv:window-pane …)  ; sub-windows
```

```
          (pane-2 tv:window-pane …))
   :configurations
      '((config1 (:layout (config1 :column pane-1 pane-2))
                 (:sizes (config1 (pane-1 :even) (pane-2 :even))))
        (config2 (:layout (config2 :row pane-1 pane-2))
                 (:sizes (config2 (pane-1
                                   :limit
                                    (5 10 :characters :even)
                                  (pane-2 :even)))))
   :configuration 'config1))     ; default configuration
```

Subfenster (panes) können in einem Rahmenfenster (frame) spaltenweise (`:column`) oder zeilenweise (`:row`) angeordnet werden. Größen können absolut festgelegt, relativ bestimmt oder bzgl. der anzuordnenden Objekte berechnet werden. Dieses ist vergleichbar mit dem Boxmodell (`:vbox` vs. `:column`, `:hbox` vs. `:row`, Verwendung von `:as-needed`). Auch Füllangaben werden unterstützt (`:filler` vs. `:even`). Absolute Distanzen können ebenso festgelegt werden. Als Erweiterung sind hier Angaben nicht nur in der Maßeinheit Pixel möglich, sondern es können auch Maßeinheiten wie Zeilen oder Zeichen verwendet werden. Anordnungen mit `:even` lassen sich durch Minimal- und Maximalanforderungen einschränken (`:limit 5 10` Dieses entspricht den Beschränkungsangaben für Füller in dem Boxmodell der in dieser Arbeit vorgestellten Anordnungsbeschreibungen. Die hier aufgeführten Layoutbeschreibungen von Genera beziehen sich auf Anordnungen von Fenstern in anderen Fenstern. Graphische Objekte, wie sie durch Sichtbereichselemente unterstützt werden, können dagegen nicht angeordnet werden. Referenzen (lokale Anordnungen) werden ebenfalls nicht unterstützt.[10]

Zur Defintion von Anordnungen für Dialogelemente wurden eine Reihe von interaktiven Werkzeugen entwickelt (Interface Builder, Dialog Designer). Die meisten Systeme ermöglichen jedoch nur eine statische Anordnung von Interaktionsobjekten in nicht in der Größe veränderbaren Dialogen. Sowohl Füllabstände als auch darzustellende Interaktionsobjekte passen sich nicht an die Fenstergröße an. Eine Ausnahme bildet das System FormsVBT von Avrahami et al. [1]. In diesem System werden sowohl textuelle als auch graphische Beschreibungsformen unterstützt. Die verwendeten Anordnungsbeschreibungen orientieren sich ebenfalls an dem Box-Modell von TEX. Für Boxanordnungen wurde eine ähnliche Beschreibungssprache entwickelt.[11] Es werden (in etwas anderer Terminologie) ebenfalls `:filler` und `:fbox` Beschreibungen unterstützt. Eine wichtige Erweiterung ist die Kopplung von textueller und graphischer Darstellung. Eine Manipulation der einen Darstellung bewirkt eine entsprechende Änderung der anderen Darstellung. Das Gesamtsystem ist allerdings nicht objektorientiert und die Layoutalgorithmen können nur auf Dialogelemente angewendet werden. Referenzen, wie sie von den in diesem Bericht vorgestellten Sichtbereichselementen verwaltet werden, sind nicht vorgesehen. Sichtbereichselemente bilden die Basis für komplexere Visualisierungen, lassen sich jedoch mit den gleichen Beschreibungen wie Standard-Interaktionsbausteine anordnen.

## 7 Zusammenfassung und Ausblick

Der in diesem Beitrag vorgestellte Ansatz zur Layoutspezifikation ist ein wichtiger Bestandteil einer integrierten Umgebung, die die Erstellung von Visualisierungen unterstützt [13]. Unser Ansatz gestattet es, für beliebige komplexe Objekte entsprechende graphische Repräsentationen zu definieren und diese mithilfe der angebotenen Spezifikationsmechnismen zweidimensional anzuordnen. Die gebotene Funktionalität reicht von der Erstellung mehr textorientierter Benutzungsoberflächen (Inspektor) bis zur allgemeinen graphischen Darstellung komplexer Datenstrukturen (DAGs) und Algorithmen (Wegefindung). Dabei hat sich die Anlehnung unserer Spezifikationssprache an das TEX-Modell gut bewährt.

Die unserem System zugrundeliegenden Algorithmen bieten durch ihre leichte Erweiterbarkeit eine gute Ausgangsbasis für weitergehende Entwicklungen. Beliebige zweidimensionale Layoutbeschreibungen können ohne Änderung des bestehenden Systems integriert werden. Als wichtigste Erweiterung wäre eine Beschreibung der vertikalen Überlagerungsstruktur der (zweidimensionalen) Sichtbereichselemente zu nennen. In der gegenwärtigen Version ist die Überlagerungsstruktur nicht determiniert. Es ist noch nicht

klar, wie eine Beschreibung hierfür aussehen könnte. Beispielsweise ist die Definition einer partiellen Ordnung für Elemente einer Ebene denkbar. Ebenen sollten wie Folien überlagert werden können.

Durch Verwendung von Layoutbeschreibungen tritt der Vorgang des Anordnens von Elementen (in einer Box) in den Hintergrund. Für ein Verarbeitungsmodell wurde eine Notationsform entworfen, die das zu lösende Problem beschreibt, nicht jedoch die Lösungsmethode widerspiegelt. Inwieweit die Anordnungsschemata als Primitve zur Repräsentation von Anordnungswissen verwendet werden können, ist nicht geklärt, da bisher noch keine formale Semantik für die Layoutsprache angegeben wurde. Hier könnte sich eine Untersuchung anschließen, inwieweit sich das Modell zur Repräsentation von zweidimensionalem Anordnungswissen eignet bzw. in welche Richtung Erweiterungen nötig sind.

## Danksagung

Der zweite Autor wurde waehrend der Anfertigung dieses Beitrags durch ein Stipendium des Wissenschaftsausschusses der NATO ueber den DAAD unterstuetzt.

## Literatur

1. Avrahami, G., Brooks, K.P., Brown, M.H.: *A Two-View Approach to Constructing User Interfaces*, ACM SIGGRAPH Computer Graphics, 23, 3, July 89, 137-146 (1989)

2. Borning, A.: *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM Trans. Programming Languages and Systems, 3, 4, Oct. 1981, 353-387 (1981)

3. Brown, M. H.: *Algorithm Animation*, ACM Distinguished Dissertations Series, MIT Press (1988)

4. Eisenstadt, M., Brayshaw, M.: *The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming*, Journal of Logical Programming, 5, 277-342 (1988)

5. Haarslev, V., Möller, R.: *Visualization of Experimental Systems*, in: Proceedings, 1988 IEEE Workshop on Visual Languages, Pittsburgh/PA, Oct. 10.-12. 1988, IEEE Computer Society Press, 175-182 (1988)

6. Haarslev, V., Möller, R.: *VIPEX: Visual Programming of Experimental Systems*, in: Visual Languages and Visual Programming, S. K. Chang (Hrsg.), Plenum Press, New York and London, 185-212 (1990)

7. Herczeg, M.: *INFORM-Manual: Clusters Version 1.9*, Universität Stuttgart, Institut für Informatik, Forschungsgruppe INFORM, Verbundprojekt WISDOM, Forschungsbericht, FB-INF-85-27 (1986)

8. Herczeg, M.: *USIT - Ein Benutzerschnittstellen-Baukasten für ein Interaktionskontinuum*, in: German Chapter of the ACM, Berichte Nr. 39, Software Ergonomie '89, S. Maaß, H. Oberquelle (Hrsg.), Teubner (1989)

9. Keene, S.: *Object-Oriented Programming in CLOS - A Programmer's Guide to CLOS*, Addison-Wesley (1989)

10. Knuth, D.E.: *TEX and Metafont – New Directions in Typesetting*, Digital Press (1979)

11. Linton, M.A., Vlissides, J.M., Calder, P.R.: *Composing User Interfaces with InterViews*, IEEE Computer, 22, 2, 8-22, (1989)

12. Matwin, S., Pietrzykowski, T.: *Prograph: A Preliminary Report*, Computer Languages, 10, 2, 91-126 (1985)

13. Möller, R.: *Entwicklung von Visualisierungswerkzeugen in objektorientierten Systemen unter Verwendung von KI-Programmiermethoden*, Diplomarbeit am Fachbereich Informatik, Universität Hamburg, Dezember (1989)

14. Stoyan, H.: *Programmiermethoden der Künstlichen Intelligenz*, Band 1, Studienreihe Informatik, Springer-Verlag (1988)

15. Symbolics: Handbücher zur Symbolics-Programmierumgebung, 7A Programming the User Interface – Concepts, Symbolics Inc. (1988)

16. Wright, S., Feuerzeig, W., Richards, J., *pluribus: A Visual Programming Environment for Education and Research*, in: 1988 IEEE Workshop on Languages for Automation, Symbiotic and Intelligent Robotics, Univ. of Maryland, College Park, Maryland, Aug. 29-31, IEEE Soc. Press (1985)

---

[1] Allegro Common Lisp und Apple Macintosh sind eingetragene Warenzeichen der Firma Apple Computer.

[2] Die Lage der Knoten kann auch von der Topologie der Kanten abhängig sein, wenn z.B. Kreuzungen vermieden oder ästhetische Gesichtpunkte berücksichtigt werden sollen.

[3] Die Pfeilform des Objekts A symbolisiert gleichzeitig die Referenzrichtung, d.h. A ist das referenzierende und B das referenzierte Objekt. Über die Definition der Pfeilspitze ist hier nichts ausgesagt. Diese könnte durch eine Referenz zu einem Objekt C festgelegt werden.

[4] Die Linie aus Abbildung 3 zwischen P1 und P2 ist nicht interaktiv verschiebbar, da deren Klasse `line-view-item` sinnvollerweise nicht `moveable-view-item-mixin` als Superklasse hat.

[5] Der verwendete Algorithmus zur Anordnung der Graphknoten aus Abbildung 4 stammt von M. Hußmann, Universität Hamburg.

[6] Sollte ein Restbedarf bestehen, so bedeutet dieses, daß der zur Verfügung gestellte Platz nicht ausreicht. Die Boxelemente stehen über den Rand der Box hinaus.

[7] Dieses ist im Zusammenhang mit Rahmenboxen notwendig, die sonst an unteren Boxrand nicht „anliegen“.

[8] Die Grenzsegmente eines Kreises sind die Seiten des minimalen umschließenden Rechtecks mit waagerechten bzw. senkrechten Kanten.

[9] Symbolics und Genera sind eingetragene Warenzeichen der Firma Symbolics, Inc., Cambridge, Massachussets.

[10] Genera unterstützt allerdings die spezielle Anordnung von azykilschen Graphen im Rahmen der sog. Presentation-Types. Presentation-Types können mit Klassen von Sichtbereichselementen verglichen werden, haben aber im Zusammenhang mit Eingaben weiterführende Aufgaben. Als Vorteil der hier vorgestellten Sichtbereichselemente ließe sich die orthogonalere Realisierung innerhalb des CLOS-Objektsystems nennen.

[11] Das Gesamtsystem FormsVBT wurde in eine Modula-2-Umgebung integriert. Im Gegensatz zu einer Lisp-Umgebung läßt sich bei Modula die Beschreibungssprache für Anordnungen jedoch nicht in die Sprache integrieren, sondern bildet eine eigene, isolierte Interpretationsschicht (vgl. Abschnitt 3.2).