Entwicklung von Visualisierungswerkzeugen in objektorientierten Systemen unter Verwendung von KI-Programmiermethoden

Ralf Möller

FBI-HH-B-149/90

15. März 1990

Universität Hamburg
Fachbereich Informatik
Bodenstedtstraße 16
D-2000 Hamburg 50

Zusammenfassung:

Dieser Bericht stellt die Konzepte eines Unterstützungssystems zur Visualisierung objektorientierter Programmsysteme vor. Wichtige Bestandteile dieses Systems sind erweiterbare Bausteine, die schon während der Programmentwicklung die Verwendung bzw. 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.

Zur Kopplung von Visualisierung und Anwendung existieren verschiedene Konzepte und Modelle. Dieser Bericht faßt die Grundkonzepte der wesentlichen bisher existierenden Techniken zusammen. Mithilfe des Metaobjektprotokolls einer objektorientierten Programmierumgebung wird ein konkreter Kopplungsmechanismus realisiert.

Abstract:

This report presents the concepts of a system to support the visualization of object-oriented programming systems. Important points are extensible tools that allow the use and construction of visualizations during program development. The geometrical arrangement of visualization objects is specified by declarative layout descriptions which use a box-and-glue model also found in the TEX system. Based on this model other layout mechanisms are presented.

Different models to separate application and visualization have been developed. This report summarizes their main concepts and presents a concrete realization of a coupling mechanism using the metaobject-protocol of the underlying object-oriented programming environment.

Inhaltsverzeichnis

1. Einleitung 1

1.1. Visualisierung 1

1.2. KI-Programmiermethoden 3

2. Dimensionen der Programmvisualisierung 6

2.1. Statische und dynamische Aspekte 6

2.2. Interpretation der Daten 7

2.2.1. Konzeptionelle, problemnahe Interpretation 7

2.2.2. Inhärent visuelle Interpretation 8

2.2.3. Strukturelle Interpretation 9

2.2.4. Metaphorische Interpretation 11

2.3. Direkte vs. synthetische Visualisierung 12

2.4. Manipulation und Editierung 12

2.5. Abstraktions- und Konzeptbildung 13

2.6. Zusammenfassung: grundlegende Komponenten 15

3. Grundbausteine 18

3.1. Integration in eine bestehende Umgebung 19

3.2. Ein Anwendungsbeispiel: ein Polygoneditor 21

3.3. Layoutangaben 26

3.3.1. Layoutmuster für Boxen 27

3.3.2. Layoutmuster für Füller 28

3.3.3. Eingespleißte Layoutangaben 28

3.3.4. Syntax der Layoutmuster 28

3.3.5. Verbesserung der Definition des Beispieldialoges mithilfe von Layoutangaben 31

3.4. Dialogfenster und Sichtbereiche 32

3.4.1. Ein Polygoneditor als Sichtbereichsinstanz 33

3.4.2. Vergleich mit bestehenden Systemen 34

3.5. Sichtbereichselemente 35

3.5.1. Fortsetzung des Beispiels:
Polygonstützpunkte als Sichtbereichselemente 36

3.5.2. Deutung der Sichtbereichselemente als Sub-Subfenster 39

3.5.3. Verschiebbarkeit, Gruppierung und Markierung 39

3.6. Layoutalgorithmen 40

3.6.1. Box-orientierte Layoutalgorithmen 40

3.6.2. Generelle Layoutangaben und -algorithmen 46

3.6.3. Referenzen 49

3.6.4. Beispiel: Visualisierung einer Vererbungshierarchie 54

3.6.5. Layoutangaben in anderen Systemen 59

4. Teilung der Zuständigkeiten 65

4.1. Smalltalk's Model-View-Controller Schema 65

4.2. Active Values in Loops 69

4.3. Algorithmenereignisse in Balsa 70

4.4. Contacts in CLUE 73

4.5. Artists in Incense 74

5. Ein Verarbeitungsmodell zur Erstellung von Visualisierungen 75

5.1. Indirekte Werte von Objekteinträgen 75

5.2. Dämonen für Eintragszugriffe 82

5.3. Dämonen für Methodenevaluationen 90

5.4. Elementarereignisse 92

5.5. Zeitführung 93

5.6. Vergleich und Zusammenfassung 96

5.7. Motivation zur Verwendung von Visualisierungen 97

6. Visuelle Sprachen - ein Ansatz zur Formalisierung von Visualisierungen 103

6.1. Visuelle Programmiersysteme 102

6.2. Visuelle Sprachen 104

6.3. Generelle Problematik in visuellen Programmsystemen für allgemeine Anwendungen 106

7. Zusammenfassung, Erweiterungen und Einordnung in die Forschungslandschaft 109

Literatur 113

Anhang A. Allegro Common Lisp und ObjectLisp 121

Anhang B. Überblick über die Klassenhierarchie 124

Anhang C. Auflistung des Quellcodes einiger Beispiele 125

C.1. Quellcode des Dialogs zur Visualisierung einer Vererbungshierarchie aus Kapitel 3.6.4. 125

C.2. Quellcode des Beispiels mit dem Einschränkungsnetz aus Kapitel 5.2. .. 128

C.2.1. Anwendungskomponente 128

C.2.2. Visualisierungskomponente 131

C.3. Quellcode des Beispiels der Kantenbeschriftung mithilfe eines Einschränkungsnetzes 134

C.3.1. Anwendungskomponente 134

C.3.2. Visualisierungskomponente 135

Abbildungsverzeichnis

Abbildung 2.1: Regeleditor-Dialog der Expertensystem-Entwicklungsumgebung Nexpert Object 10

Abbildung 2.2: „Frame-Inspector"-Dialog der Expertensystem-Entwicklungsumgebung Goldworks II/Macintosh 10

Abbildung 2.3: Darstellung eines Regelnetzwerks in Nexpert 11

Abbildung 2.4: Darstellung der „Frontplatte" eines simulierten Gerätes zur Generierung von Wellenformen 13

Abbildung 2.5: Dimensionen der Visualisierung 17

Abbildung 3.1: Ausschnitt aus der Klassenhierarchie 20

Abbildung 3.2: Ein Polygoneditor-Dialog 22

Abbildung 3.3: Dialog zur Bestätigung der Verbindung von Stützpunkten zu einem Polygon 24

Abbildung 3.4: Schematische Darstellung einer Anordnung mithilfe eines Layoutmusters 30

Abbildung 3.5: Verwendung einer `:fbox` 31

Abbildung 3.6: CLOS-Klasseninspektor-Dialog 41

Abbildung 3.7: CLOS-Klasseninspektor-Dialog mit vergrößerter Fläche 43

Abbildung 3.8: Inspektor-Dialog für eine Liste von Objekten 44

Abbildung 3.9: Schematische Darstellung von geschachtelten Boxen 44

Abbildung 3.10: Wasserkrugmodell für Füllangaben 45

Abbildung 3.11: Referenzmodell 49

Abbildung 3.12: Bestimmung des Zeichenrechtecks 52

Abbildung 3.13: Kante zwischen zwei Knoten 53

Abbildung 3.14: Visualisierung einer Vererbungshierarchie 54

Abbildung 3.15: Ein Ausschnitt aus der CLOS-Klassenhierarchie 59

Abbildung 3.16: Typen von sog. „Clustern" zur Anordnung von Fenstern 60

Abbildung 3.17: Systeminspektor der Programmierumgebung Smalltalk-80 63

Abbildung 3.18: Klasseninspektor für die Klasse `BrowserView` 63

Abbildung 4.1: MVC-Kommunikationsschema 66

Abbildung 4.2: Ein Terminalfenster in der Smalltalk-80 Programmierumgebung ... 67

Abbildung 4.3: Spezieller Inspektor für alle Komponenten des MVC-Schemas 67

Abbildung 4.4: Verteilungschema für „Algorithmenereignisse“ 72

Abbildung 4.5: „Artist“-Organisationsschema 74

Abbildung 5.1: Skizze einer Anzeige der Klasse `overview-gauge` 77

Abbildung 5.2: Obere und untere Schranke der Einschätzung 83

Abbildung 5.3: Einschränkungsnetz dargestellt als DAG 85

Abbildung 5.4: Zustand des Einschränkungsnetzes nach der Propagierung 89

Abbildung 5.5: Einschränkungsnetz mit Schaltfläche zur Überwachung der Propagierung 94

Abbildung 5.6: Konzeptuelle Visualisierung des Einschränkungsnetzes zur Kantenbeschriftung 99

Abbildung 5.7: Zustand des Einschränkungsnetzes zur Kantenbeschriftung nach der Propagierung mit einer eindeutigen Lösung 101

Abbildung 5.8: Zustand des Einschränkungsnetzes zur Kantenbeschriftung nach der Propagierung mit einer mehrdeutigen Lösung 102

Abbildung 6.1: Schnappschuß der Zuordnung im Algorithmus „Stabile Heirat“ .. 107

Abbildung 6.2: Dimensionen von Visuellen Programmiersprachen 108

Abbildung 7.1: Grobe Skizze einer visuellen Definition von Anordnungschemata 111

Abbildung A.1: Kombination von Methoden 122

1. Einleitung

1.1 Visualisierung

Der Begriff des „Visualisierens" läßt sich umgangsprachlich auf verschiedene Weisen umschreiben [Wahrig 80]:

a.) „in Bildform in Anschauung umsetzen",
b.) „mit den Mitteln der Werbung so sichtbar machen, daß es Aufmerksamkeit erregt",

oder auch nach [Langenscheidt 60] (s. engl.: to visualize):

c.) „(sich) vor Augen stellen",
d.) „sich ein Bild machen von".

Die Definitionen a.) und b.) rücken mehr den Prozeß der Erstellung einer Visualisierung und einen damit verbundenen Zweck (z.B. Werbung) in den Vordergrund. Hier soll ausgenutzt werden, daß entsprechend ausgewählte und dargestellte visuelle Informationen von einem Betrachter leicht aufgenommen und verarbeitet werden können. Die damit verbundene interne, mentale Verarbeitung wird in den Definitionen c.) und d.) angesprochen.

Diese Arbeit stellt grundlegende Verarbeitungsmodelle und programmtechnische Hilfsmittel zur Erstellung von Visualisierungen vor, betrifft also mehr den durch die Definitionen a.) und b.) ausgedrückten Interpretationsaspekt des Begriffs „visualisieren". Dabei ist der Terminus „Aufmerksamkeit erregen" durchaus auch in diesem Zusammenhang zutreffend. Nicht zuletzt sollen visuelle Darstellungen von Wissensbasen die Navigation durch größere Wissensbestände erleichtern [Rickert 86]. In objektorientierten Systemen ist es das Ziel, vordefinierte Objektklassen zu finden und mit „eigenen" Klassen zu kombinieren.

Visualisierungen werden in verschiedenen Teilgebieten der Informatik verwendet. Die Erzeugung von visuellen, graphischen Darstellungen wird im Bereich Computergraphik untersucht. Als grundlegendes Hilfsmittel werden Visualisierungen in CAD-Anwendungen verwendet. Innerhalb der Künstlichen Intelligenz (KI) wird die Repräsentation von visuellem Wissen in verschiedenen Teildisziplinen untersucht. Hierzu gehören die Bild- und Sprachverarbeitung. Innerhalb von sprachlichen Äußerungen auftretende Referenzen etwa auf reale Objekte machen eine Repräsentation und

Verarbeitung von visuellem und räumlichem Wissen erforderlich [Habel 87], um z.B. durch Konsistenzprüfungen Mehrdeutigkeiten aufzulösen [Pribbenow 88]. Die Untersuchungen hierzu betreffen u.a. die rechnerinterne Repräsentation von mentalen Bildern [Anderson 88] und korrespondieren mit den Definitionen c.) und d.) dieses Abschnitts. Bildverarbeitungssysteme können gesteuert werden, indem u. a. Formrepräsentationen von in einer untersuchten Szene „erwarteten“ Objekten zur Auswertung herangezogen werden [Ballard & Brown 82]. Einige Visualisierungsbeispiele für Algorithmen zur Kantenklassifizierung eines Würfels zeigen eine Verwendung der in dieser Arbeit entwickelten Visualisierungswerkzeuge.

Kennzeichnend u.a. für die KI-Gebiete ist die Rechnersimulation von erstellten Modellen. Dabei werden Hilfsmittel zur Programmierung von visuellen Darstellungen benötigt. Dieses betrifft nicht nur primär visuelle Anwendungen wie z.B. Bildverarbeitungssysteme, sondern umfaßt auch Benutzerschnittstellen z.B. von Wissensbasen. Ein vielzitierter Ansatz zur Wissensrepräsentation sind die Sprachen der KL-ONE-Familie. Die meisten Autoren verwenden eine von Brachman und Schmolze eingeführte visuelle Netzdarstellung [Brachman und Schmolze 85]. Graphische Darstellungen für KL-ONE-Systeme stellen [Kindermann & Quantz 88] und [Bergmann & Gerlach 87] vor. Entwicklungsumgebungen für Expertensysteme verwenden in hohem Maße graphische Editierwerkzeuge. Hier wird also eine visuelle Darstellung von Wissen bzw. Wissensrepräsentationsformalismen vorgenommen.

Auch bei der Untersuchung von Repräsentationen von visuellem und räumlichem Wissen ist es wichtig, geeignete Programmierumgebungen zur Verfügung zu haben, um Modelle und Theorien überhaupt adäquat testen zu können [Haarslev & Möller 88a]. Die vorherrschenden symbolischen Repräsentationen können aufgrund der Datenflut nicht in angemessener Zeit ausgewertet und manipuliert werden. Eine vergleichbare Situation tritt bei der Auswertung von Simulationsergebnissen (z.B. im VLSI-Bereich) auf.

Kapitel 2 dieser Arbeit betrachtet die verschiedenen Dimensionen und Sichtweisen von Visualisierungen. Es werden in diesem Zusammenhang relevante Teilaspekte von Benutzerschnittstellen von existierenden Wissensrepräsentations- und Programmierwerkzeugen vorgestellt und verglichen. Die Grundbausteine der vorgestellten Systeme werden in dieser Arbeit zu einem Rahmensystem zusammengefaßt, das die Erstellung von Visualisierungen (stark) vereinfacht.

1.2. KI-Programmiermethoden

Zur Realisierung des Rahmensystems werden bekannte KI-Programmiermethoden eingesetzt Insbesondere wird die objektorientierte Programmierung verwendet. Nach [Stoyan 88, Seite 6] kommt insbesondere in KI-(Programmier-)Methoden eine „Verknüpfung von theoretischer und praktischer Programmierung“ zum Ausdruck:

„Ein theoretischer kognitiver Apparat wird konzipiert ... und in ein Informationsverarbeitungsmodell umgesetzt, für das dann eine Programmiersprache definiert wird.“

Ein Informationsverarbeitungsmodell spiegelt sich in einem bestimmten Programmierstil wider. Dieser wiederum wird durch Sprachkonstrukte einer Programmiersprache (mehr oder weniger gut) unterstützt. Durch Verwendung der Programmiersprache für Testanwendungen wird dann iterativ versucht, das zugrundeliegende Informationsverarbeitungsmodell zu verifizieren bzw. zu verbessern. Es existieren verschiedene (KI-)Programmierstile wie z.B. der funktionale, der objektorientierte, der relationale und der regelbasierte Programmierstil. Eine vollständige Trennung zwischen den Stilen kann nicht gezogen werden. In allen praktisch einsetzbaren KI-Systemen sind aber auch mehr oder weniger gut getarnte imperative Konzepte realisiert.

Nun stellt diese Arbeit weder eine neue Programmiersprache noch einen neuen Programmierstil vor. Ein wesentliches Merkmal von KI-Systemen findet sich aber auch hier wieder: es wird versucht, für Teilverarbeitungsmodelle eine Programmrepräsentation zu entwickeln, die das Problem, das zu lösen ist, deklarativ beschreibt, nicht jedoch die Lösungsmethode widerspiegelt. Der entscheidende Punkt ist, daß die Beschreibungsformen in bestehende Verarbeitungsmodelle integriert und nicht isoliert betrachtet werden sollen. Als Untersuchungsgrundlage dient die standardisierte Common Lisp Erweiterung CLOS [Keene 89, X3J13], die Common Lisp [Steele 84] um objektorientierte Konzepte erweitert. Doch gerade die Tatsache, daß sich objektorientierte Systeme auch im „klassischen Software-Engineering“ bewähren (z.B. C++ [Stroustrup 87], auch: Smalltalk [Goldberg & Robson 83]), macht deutlich, daß die Untersuchungen zur Visualisierung in dieser Arbeit nicht nur KI-Systeme betreffen, sondern auch in anderen Systemen Verwendung finden können.

Kapitel 3 schildert die in dieser Arbeit entwickelten Grundbausteine zur Erstellung von Visualisierungen und vergleicht sie mit denen existierender Systeme. Der Kernpunkt der Überlegungen liegt in der Erstellung von Bausteinen für sog. Ad-hoc-Visualisierungen (im Sinne eines „Rapid Prototyping“), die schon während der Programmierphase eines „Anwendungsprogrammes“ erstellt werden. Hierauf aufbauend sollten aber durch Erweiterung und Verfeinerung auch spezialisierte Visualisierungen

erstellt werden können. Zur Erreichung dieser Ziele wurde ein erweiterbares Objektklassensystem zur Verfügung gestellt, das Standardverhalten auf einfache Weise bereitstellt, aber genügend Flexibilität bietet, um weiterführende Anforderungen zu unterstützen. Der Kerngedanke hierbei ist, ein Verwaltungssystem zu erstellen, mit dem beliebige graphische Objekte in Fenstern plaziert und (relativ zueinander oder relativ zur Gesamtfläche) angeordnet werden können. Das Aussehen der Objekte wird durch Methoden für vom Verwaltungssystem evaluierte generische Funktionen bestimmt. Das Verwaltungssystem stellt ein Standardverhalten bereit, das in einem gewissen Rahmen durch objektorientierte Erweiterungen an bestimmte Anforderungen angepaßt werden kann. Anordnungsschemata für graphische Objekte lassen sich deklarativ beschrieben. Sie sind also gleichberechtigte Datenobjekte, die z.B. von einem Parser verarbeitet werden. Die durch Anordnungsschemata beschriebene Grundanordnung wird von einem Interpreter in eine konkrete Plazierung einzelner spezieller Objekte umgesetzt.

Weiterhin wird untersucht, inwieweit sich Anwendungs- und Visualisierungssystem programmtechnisch trennen lassen. Es werden Methoden zur losen Kopplung der beiden Bereiche vorgestellt, die gewährleisten, daß die eine Seite von der anderen (fast) vollständig getrennt werden kann. Die Notwendigkeit zu einer Trennung und Eingrenzung wurde von vielen Autoren herausgestellt [z.B.: Young et al. 88, Myers 89b]. Durch eine Trennung wird zweierlei erreicht. Auf der einen Seite kann die Anwendung (weiter-)entwickelt werden, ohne auf eine Visualisierung angewiesen zu sein. Hier spielen Modularisierungsgesichtspunkte eine große Rolle. Dieses heißt nicht, daß es nicht einen „Kontrakt“ zur Kopplung geben kann. Gibt es diesen nicht, so sind besondere Techniken erforderlich, eine Visualisierung an eine Anwendung anzukoppeln, ohne letztere zu verändern. Es kann i.a. nicht die volle Kommunikationsbreite eines Kontraktes erzielt werden. Auf der anderen Seite ist eine Trennung auch seitens der Visualisierungskomponenten erwünscht. Es lassen sich wiederverwendbare Elementarbauteile leichter herstellen. Auf eine Verwendung von vordefinierten Bauteilen ist man aus Zeitgründen angewiesen.

Im Zusammenhang mit Visualisierungen ist der Begriff eines „interessanten Ereignisses“ relevant. So kann z.B. ein Schreibzugriff auf einen Eintrag (slot) eines Objektes für sich genommen als „interessantes Ereignis“ interpretiert werden, da sich der Objektzustand und damit eventuell die Visualisierung des Objekts dadurch ändert. Bei Vertauschung zweier Eintragswerte hingegen ist das „interessante Ereignis“ erst durch die Kombination der beiden Zugriffe definiert. Inwieweit „Zwischenzustände“ visualisiert werden sollen ist also interpretationsabhängig. Nachdem in Kapitel 4 die zu diesem Thema existierende Literatur sowie die entwickelten Modelle bzw. Systeme vorgestellt

werden, erläutert das fünfte Kapitel die in dieser Arbeit verwendeten Techniken zur deklarativen Definition von (bzgl. einer Visualisierung) elementaren Programmteilen.

Formale Aspekte aus dem Bereich der Visuellen Sprachen schildert das Kapitel 6. Einige der in diesem Bereich erstellten Systeme werden mit dem in dieser Arbeit vertretenen Ansatz verglichen. Die in dieser Arbeit erstellten Bausteine können als Grundlage zur Implementierung eines visuellen Programmiersystems dienen.

Kapitel 7 bildet den Abschluß mit einer Zusammenfassung, einem Ausblick auf weitere Entwicklungen und einer Einordnung der in dieser Arbeit vorgestellten Konzepte in verschiedene Teilforschungsgebiete der Informatik.

2. Dimensionen der Programmvisualisierung

In diesem Abschnitt werden einige Kriterien zur Klassifikation von Visualisierungssystemen vorgestellt. Übersichtsartikel zu diesem Thema wurden von [Myers 86a und 88] sowie von [Raeder 85] und [Brown & Sedgewick 84, 85] verfaßt. Umfangreiche Literaturangaben enthält auch [Haarslev & Möller 88c] und [Hsia & Ambler 88]. Myers definiert den Begriff „Programmvisualisierung“ als Illustration von einigen Aspekten eines Programms selbst bzw. von dessen Ablaufverhalten.

2.1. Statische und dynamische Aspekte

Im Zusammenhang mit Programmvisualisierungen sind statische und dynamische Aspekte zu unterscheiden. Eine statische Darstellung umfaßt einen Schnappschuß z.B. einer Datenstruktur, die in einem sog. Inspektor (beispielsweise zu Fehlersuchzwecken) dargestellt wird. Dabei wird eine Aktualisierung erst nach expliziter Anforderung vorgenommen[1]. Erfolgt eine „ständige“ Aktualisierung der Darstellung, so kann man von einem dynamischen System sprechen. Dabei ist das Wort „ständig“ nur sehr unscharf definiert. Die Definition von sog. „interessanten Ereignissen“ ist von dem einzelnen System abhängig. Ein „interessantes Ereignis“ tritt an Zeitpunkten auf, an denen sich der Systemzustand in einem für die Visualisierung relevantem Maße ändert. Bei dynamischen Systemen ist daher die Dauer der Intervalle zwischen zwei Aktualisierungen zu betrachten.

Die Bandbreite reicht von einfachen Systemen zur Markierung der „aktuellen Zeile im Quellcode“, einer Markierung des „aktuell bearbeiteten“ Knoten in einem Graphen [Brown et al. 85] bis hin zu animierten Abläufen von Programmen (algorithm animation) [London & Duisberg 84, Brown 88] und insbesondere visualisierten Simulationen [Smith 86]. Bei langwierigen Programmläufen ist es dagegen ratsam, einen diskreten Aktualisierungsmodus (Fortlassung von Zwischenschritten) wählen zu können.

---

[1] Dieses Verhalten kann u.U. durchaus erwünscht sein.

2.2. Interpretation der Daten

Der Dualismus von Daten- und Programmvisualisierungen sowie die Beziehung zur visuellen Programmierung wurde von Chang mit dem Terminus „generalisiertes Piktogramm“ (generalized icon) ausgedrückt [Chang et al. 86, Chang 87]. Generalisierte Piktogramme sind entweder Objektpiktogramme (für Daten) oder Prozeßpiktogramme (für Programme) bestehend aus einem Tupel (X_m, X_i). X_m bezeichnet den (kontextabhängigen) bedeutungstragenden Teil (meaning); X_i steht für den verkörperten Teil, hier das Bild (image) bzw. für die zu dessen Erzeugung benötigten geometrischen und graphischen Informationen. Zur Erzeugung von Visualisierungen ist es nötig, bei vorgegebenen Objekten und Prozessen, d.h. bei vorgegebenem X_m, ein geeignetes X_i zu ermitteln oder zu definieren. X_i ist durch die Deutung und Sichtweise der Objekte und Prozesse bestimmt, also deren Interpretation. Je nach Interpretation ist die Gewinnung von X_i mit unterschiedlichem Aufwand verbunden. Den umgekehrten Fall, also die Festlegung oder Ermittlung eines X_m bei gegebenem X_i, tritt z. B. in Systemen zur visuellen Programmierung auf. X_i stellt hier die zu übersetzenden Programmkonstrukte dar. Wenn nur Daten mit inhärent visueller Interpretation (durch konventionelle Programme) verarbeitet werden, so wäre als generalisiertes Piktogramm (ε, X_i) anzugeben, wobei ε für eine nicht definierte Bedeutung steht.

Für die Interpetation von Daten ist das dazu verwendete Domänenwissen (Wissen über das Anwendungsfeld eines Programmsystems) eine ausschlaggebende Kenngröße. Je mehr Domänenwissen zur Visualisierung herangezogen wird, desto mehr kann die Interpretation von einer strukturellen in eine konzeptionelle übergehen[1]. Es sind zum heutigen Zeitpunkt weder Repräsentationsformalismen für allgemeines Domänenwissen vorhanden, noch sind Verfahren bekannt, die dieses in irgendeiner Weise automatisch zur Visualisierung ausnutzen. Dieses gilt sowohl unter dem Aspekt der Erzeugung von Visualisierungen (Definitionen a.) und b.) aus Kapitel 1.1) als auch in Bezug auf eine Untersuchung der menschlichen Verarbeitungsmechanismen (Definitionen c.) und d.)) .

2.2.1. Konzeptionelle, problemnahe Interpretation

Die zur Visualisierung nötigen geometrischen und graphischen Informationen, können im allgemeinen Fall nicht aus den zu visualisierenden Daten gewonnen werden. Die intendierte Interpretation z.B. einer Datenstruktur wird zwar durch geeignete programmiersprachliche Techniken hervorgehoben. Hierzu gehören abstrakte Datentypen und objektorientierte Mechanismen sowie eine prozedurale Abstraktion und

[1] Zur Definition von strukturell und konzeptionell siehe unten.

Modularisierung. Es ist offensichtlich, daß die Namen von z.B. Klassen oder (generischen) Funktionen sich i.a. nicht als Ansatzpunkt zur Visualisisierung eignen[1]. Es wird i.a. sogar der Fall sein, daß die für eine angemessene Visualisierung benötigten Zusatzinformationen über die Anwendungsdomäne für den Algorithmus des diese Daten verarbeitenden Programmes nicht relevant und daher überhaupt nicht modelliert sind[2]. So kann eine Meßwerttabelle von Umweltdaten eines Statistikprogramms statt als Kurve auch als Karte des betreffenden Gebietes dargestellt werden, bei der verschiedene Größen durch verschiedene Farben gekennzeichnet sind. Eine Programm-Visualisierung eines Sortierprogramms kann in Form einer Animation durch Verschiebung von Plättchen o.ä. erfolgen [Brown 88] (sofern der Vorgang in angemessener Geschwindigkeit eine angemessene Zeit dauert). Systeme dieser Art sind speziell auf die einzelne Anwendung abgestimmt, d.h. die entwickelten Prozeduren etc. lassen sich nur beschränkt für andere Systeme nutzen. In diesem Rahmen fallen auch Visualisierungen von z.B. Molekülstrukturen bzw. chemischen Verbindungen [Mundie 88], deren Berechnung und Generierung die Rechenleistung von Supercomputern bzw. Spezialprozessoren voraussetzt, da die geometrischen Informationen zur Visualisierung von Molekülstrukturen nur durch komplexe Berechnungen gewonnen werden können. Bezüglich einer Visualisierung als Forschungshilfsmittel sind aber gerade diese Systeme sehr interessant („Seeing the Unseen“) [McCormick et al. 87].

2.2.2. Inhärent visuelle Interpretation

Bei Systemen der Computergraphik, bei denen vorgegebene geometrische Objekte durch Wiedergabeverfahren (rendering) dargestellt werden, liegt eine inhärent visuelle Interpretation zugrunde; die geometrischen Informationen sind hier explizit in den Daten enthalten. In diese Klasse fallen auch Systeme wie TRED [Strobel 89], eine Visualisierungskomponente für Trajektorien, die innerhalb des NAOS-Projekts [Neumann & Novak 84] entstanden ist. Daten einer propositionalen „Geometrischen Szenenbeschreibung“ können mit diesem System dargestellt und manipuliert werden. Ohne ein solches Hilfsmittel ist eine Handhabung der (symbolischen) Daten aufgrund der Datenflut nicht möglich.

Nicht alle Datenbestände, denen eine visuelle Interpretation zugrunde liegt, enthalten auch eine explizite Repräsentation der bzgl. einer Visualisierung notwendigen

---

[1] Hier wird einmal von der Tatsache abgesehen, daß ein Programm, dessen Bezeichner konsistent mit zufälligen Wortschöpfungen umbenannt werden, gleiches Verhalten zeigt. Bei anonymen Funktionen wie '(lambda (x) (+ x 17))' ist gar kein Name vorhanden.

[2] Gerade unter dem Abstraktionsgesichtspunkt wird versucht, möglichst allgemeine Algorithmen zu erstellen. Vergl. hierzu generische Typen z.B. in ADA.

geometrischen Informationen. Bei bildlichem Datenmaterial wie z.B. Computertomogrammen eines Schädels läßt sich eine dreidimensionale Darstellung eines auf mehreren Tomogrammen abgebildeten Gehirns durch Methoden der Bildverarbeitung gewinnen. Es ist jedoch a priori nicht bekannt, ob und wo z.B. ein Tumor vorliegt und wie dieser genau aussieht. Die relative Lage und die Form eines auf mehreren Tomogrammen abgebildeten Tumors kann nur unter Zuhilfenahme von Wissen über anatomische Formen (z.B. für Segmentationsprozesse) ermittelt werden[1]. Eine angemessene Visualisierung eines Tumors (etwa durch farbliche Hervorhebung) kann nicht erfolgen, ohne das zur Lokalisation benötigte anatomische Wissen mit einzubeziehen. In diesem Sinne enthalten diese Datenbestände die zur konzeptionellen Visualisierung nötigen geometrischen Informationen nicht einmal implizit.

2.2.3. Strukturelle Interpretation

Geometrische Informationen zur Visualisierung lassen sich bzgl. einer aufgeprägten, sog. strukturellen (kanonischen) Interpretation beliebiger zu visualisierender Daten bestimmen. Die strukturelle Interpretation betrachtet die Datenstruktur und versucht eine Visualisierung mithilfe geometrischer Informationen, die aus dieser Struktur gewonnen wurden. Die geometrischen Informationen sind gemäß dieser festgelegten Interpretation dann jeweils implizit vorhanden. Die Domänenunabhängigkeit wird mit dem Nachteil erkauft, daß sich die intendierte (konzeptuelle) Interpretation eines Programmierers z.B. einer Datenstruktur meist nur durch in die Darstellung übernommene Bezeichner des Programms indirekt widerspiegelt.

Die strukturelle Interpretation von Prozessen (vgl. Abschnitt 2.2) orientiert sich häufig am verwendeten Programmierstil bzw. dessen Verarbeitungsmodell. 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“ [Eisenstadt & Brayshaw 88]) entwickelt. Datenflußdarstellungen u.a. für funktionale Systeme sind z.B. in VIPEX [Möller 88, Haarslev & Möller 88b], Pluribus [Wright et al. 88], Prograph [Matwin & Pietrzykowski 85] zu finden. Beschränkungen (constraints) sind im ThingLab-System [Borning 81] graphisch darstellbar. Visualisierungen für KI-typische Verarbeitungsmodelle wie Regelsysteme (vorwärts- und rückwärtsverkettend) zusammen mit formularorientierten Darstellungen für Objekte und

---

1 Wissensrepräsentationsformalismen hierzu befinden sich noch im Forschungsstadium.

Klassen bieten z.B. Systeme wie Nexpert Object[1] und Goldworks/II Macintosh[2] (Abbildungen 2.1 und 2.2).

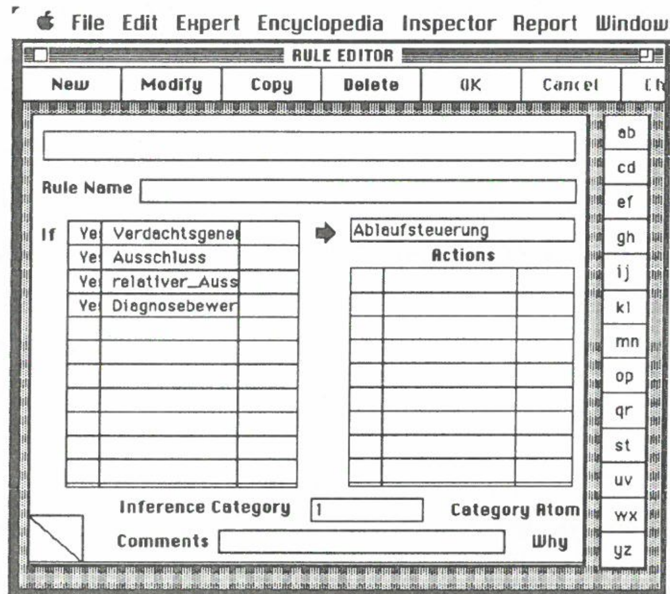


Abbildung 2.1: Regeleditor-Dialog der Expertensystem-Entwicklungsumgebung Nexpert Object.

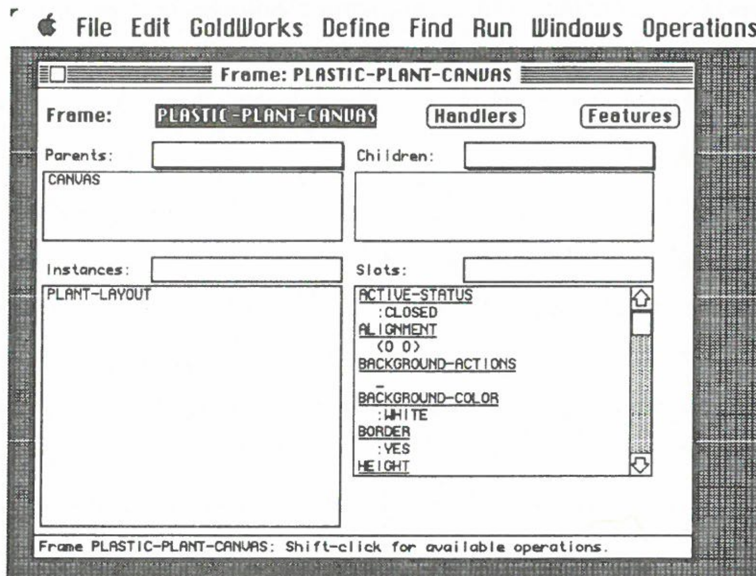


Abbildung 2.2: „Frame-Inspector"-Dialog der Expertensystem-Entwicklungsumgebung Goldworks II/Macintosh.

1 Nexpert Object ist eingetrages Warenzeichen der Firma Neuron Data, Inc.
2 Goldworks/II Macintosh ist eingetragenes Warenzeichen der Firma Gold Hill Computers, Inc.

Visualisierungen von Datenstrukturen mit struktureller Interpetation sind z.B. Baum- oder Graphdarstellungen [Rowe et al. 86, Myers 80] für durch „Pointer" verbundene Records, für Regeln (Abbildung 2.3) oder auch für Vererbungshierarchien innerhalb von objektorientierten Systemen. Felder oder „Records" sind oftmals in Tabellen- und Formularform dargestellt. Auch Rechenblattsysteme (spreadsheets) fallen in diese Kategorie. In Rechenblattsystemen können z.B. statistische Daten auch durch Diagramme und Kurven visualisiert werden.

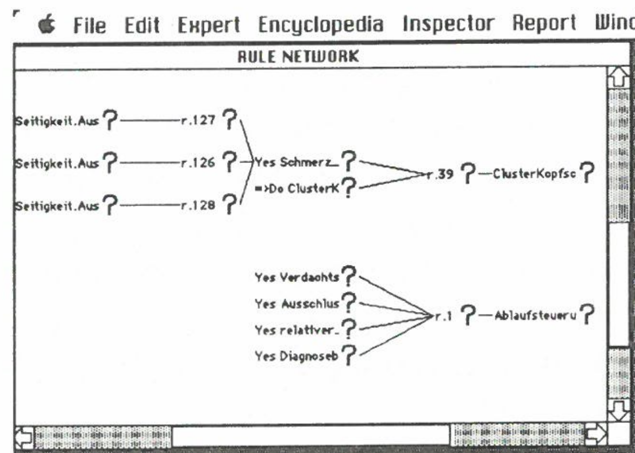


Abbildung 2.3: Darstellung eines Regelnetzwerks in Nexpert.

Die Beschränkungen der strukturellen Visualisierung von Daten z.B. in sog. Inspektor-Systemen liegen in der streng implementierungstechnischen, d.h. nicht problemnahen Präsentation der Daten bedingt durch die domänenunabhängige, aufgeprägte Interpretation. Dadurch kann zum anderen auch meist nicht ein Effekt erzielt werden, der charakterisierbar ist durch den bekannten Ausdruck: „Ein Bild sagt mehr als tausend Worte."

2.2.4. Metaphorische Interpretation

Falls die zu visualisierenden Objekte keine immanente Visualisierung besitzen, so kann durch Übertragung in einen anderen Bereich eventuell eine Metapher gebildet werden. Wie der Begriff Metapher schon ausdrückt, ist dann eine Visualisierungsform impliziert. Ein Beispiel hierfür sind die an einer Bürometapher orientierten Benutzerschnittstellen für Computersysteme. Der erste Vertreter dieser Gattung ist das Star-Bürosystem [Smith & et al. 82]. Diese Systeme sind heute weit verbreitet; es wird daher nicht mehr weiter darauf eingegangen. Wie die Vielfalt der Ausprägungen [Hayes

& Baran 89] zeigt, ist aber der Spielraum auch bei einer metaphorischen Interpretation enorm groß, dennoch finden sich in allen Systemen verwendete Standardbausteine wie etwa Fenster-, Menü- und Piktogrammobjekte (allerdings in sehr unterschiedlichen Ausprägungen).

2.3. Direkte vs. synthetische Visualisierung

Eine weitere Dimension zur taxonomischen Einteilung der Visualisierungsverfahren und -formen wurde von Brown vorgeschlagen [Brown 88]. Ist eine bijektive Abbildung möglich zwischen Daten und deren Visualisierung, so bezeichnet er die Visualisierung als direkt, sonst als synthetisch. Anders gesagt: werden die ausgegebenen Informationen wieder „eingelesen", so tritt bei direkter Visualisierung kein Informationsverlust auf. Beispiele für eine direkte Visualisierung in graphischer Form sind Systeme wie VennLisp [Lakin 86] oder IconLisp [Cattaneo et al. 86]. Listenstrukturen und damit Lisp-Programmtexte werden in struktureller Form graphisch visualisiert. Der Begriff „direkt" ist nicht identisch mit „strukturell", denn bei einer strukturellen Darstellungen kann Information herausgefiltert oder hinzugefügt worden sein.

2.4. Manipulation und Editierung

Gerade auch unter dem Gesichtspunkt der Editierung innerhalb von visuellen Darstellungen ist der Begriff der bijektiven Abbildung von Bedeutung. Ein großes Problem sind Inkonsistenzen in verschiedenen Darstellungen einer Struktur, die sich zwangsläufig ergeben, wenn mehrere Darstellungsformen vorgesehen sind (beispielsweise textuell und formularorientiert). Bei Editiervorgängen auch in einem syntaxorientierten[1] Texteditor entstehen zeitweilig syntaktisch inkorrekte Teilausdrücke, so daß eine grammatische Analyse, um eine andere Darstellung zumindest partiell zu aktualisieren, unmöglich ist. Dabei sind textuelle Editiervorgänge i.a. flexibler und schneller auf Kosten eines höheren Lernaufwandes. Menü- und Formulardarstellungen entlasten ungeübte Benutzer von syntaktischen Belangen und fördern die Exploration eines Systems [Raeder 85]. In Systemen wie LabView[2] können ganze technische Experimente durch graphische Interaktion aufgebaut und zusammengestellt werden. Ein Beispiel zeigt Abbildung 2.4. Technische Geräte können aus einfacheren Geräten

[1] „syntaxorientiert" ist im Gegensatz zu „syntaxbasiert" zu verstehen. In syntaxbasierten Editoren sind Editierfunktionen auf eine Manipulation von syntaktischen Konstrukten, nicht aber auf eine weitgehend textuelle Editierung ausgelegt. In syntaxorientierten Editoren erfolgt z.B. eine automatische Einrückung, die sich an die Syntax der Sprache des editierten Textes anlehnt.

[2] LabView ist eingetragenes Warenzeichen der Firma National Instruments.

zusammengesetzt werden, die wie primitive Geräte verwendet werden können. Dieses führt auf den Begriff der Abstraktion.

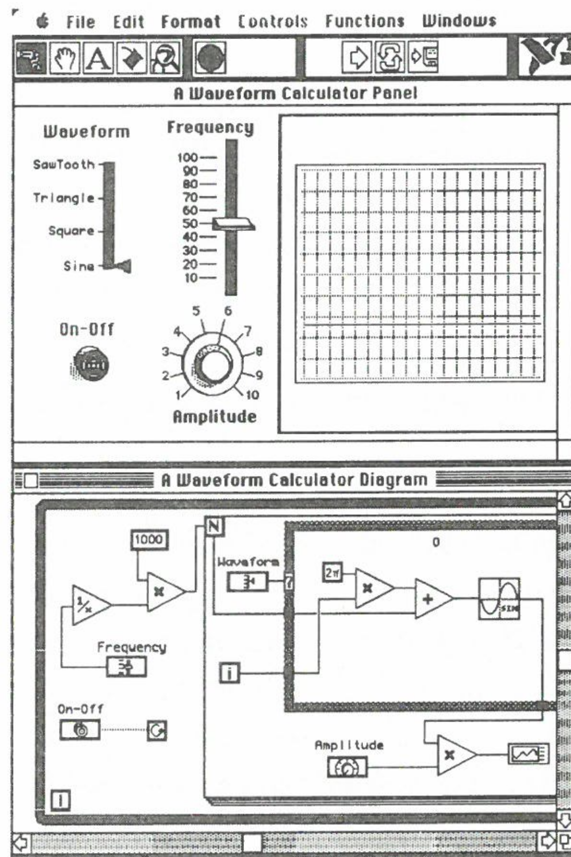


Abbildung 2.4: Darstellung der „Frontplatte" eines simulierten Gerätes zur Generierung von Wellenformen. Die „Schalter" und „Schieber" können mit der Maus bewegt werden. Das untere Fenster zeigt den Aufbau aus einfacheren „Geräten". Der Aufbau als Datenflußgraph entspricht der Domäne (Verkabelung).

2.5. Abstraktions- und Konzeptbildung

Wie bei Programmiersprachen können auch in visuellen Systemen Basisbausteine unter Verwendung spezieller Konstrukte zu neuen Bausteinen zusammengefügt werden. Beispiele hierfür sind LabView, Pict [Glinert & Tanimoto 84] und auch VIPEX [Haarslev & Möller 89]. Dabei sind insbesondere die Übergabe- und Kontrollmechanismen zu betrachten. Eine genauere Ausführung zu diesem Thema enthält [Möller 88]. Das entscheidende Manko an einer Abstraktionsbildung durch Hierarchisierung (etwa durch Bildung von Teilnetzen (Vergröberungen) wie in LabView oder VIPEX) ist jedoch, daß der Grundrahmen nicht verlassen werden kann. Ein

datenflußorientiertes System bleibt auch auf einer „höheren“ Ebene ein datenflußorientiertes System. Das System ConMan [Haeberli 88] bietet hier eine Erweiterung. Hier kann die Kontrollstruktur Datenflußsteuerung während der Benutzung des Systems ausgeblendet werden. Die für den Benutzer wichtigen Interaktionsobjekte rücken dann also stärker in den Vordergrund. Bei der Erstellung (Programmierung) eines Anwendungsystems jedoch bleibt die Datenflußorientiertheit erhalten. Gleichgelagerte Probleme treten bei der Verwendung von konventionellen Programmiersprachen wie ADA [Goos 80] oder Modula-2 [Blaschek et al. 87] auf. Das eingebaute Verarbeitungsmodell ist fixiert und kann (zumindest auf der syntaktischen Ebene) kaum verlassen werden. Übertragen auf ein Visualisierungssystem heißt das: es sind nur strukturelle nicht aber problemnahe, konzeptuelle Visualisierungen möglich.

Die Abstraktionsbildung sollte nicht nur als eine Hierarchisierung von Bauteilen, sondern im eigentlichen Sinne interpretiert werden: als Begriffs- oder Konzeptbildung. Im Gegensatz zu der strukturellen Abstraktion (z.B. durch Hierarchien) muß auch eine inhaltliche Abstraktion (visuell) unterstützt werden. Im Bereich der Programmiersysteme wird diese Sichtweise insbesondere von Lisp-Systemen unterstützt [Abelson & Sussman 85]. Die Verwendung von Prozeduren als gleichberechtigte Objekte, (first-class objects) zusammen mit der Möglichkeit, durch sog. Makros [Winston & Horn 89, Stoyan & Görz 84] neue syntaktische Formen zu definieren, reicht aus, um die Konzepte z.B. eines Systems von Klassen, Objekten und Prozeduren auch auf der syntaktischen Seite anzudeuten. Auch in Prolog ist es möglich, Meta-Interpreter zu integrieren [z.B. Sterling & Shapiro 88]. Weiterhin können Objektsysteme mit geringem Aufwand an eigene Anforderungen angepaßt werden, wenn – wie bei CLOS – auch der Interpreter mit den gleichen objektorientierten Mechanismen programmiert ist (Bildung angepaßter Metaklassen) [Bobrow & Kiczales 88]. Man kann „Subsprachen“ für neue Konzepte integrieren (z.B. Flavors [Bromsley & Lamson 87]), ohne diese als etwas „Aussätziges“ in eine Datei zu schreiben, diese mit einem extra für diese eine Sprache erstellten Kompilierer zu übersetzen und dann in einer getrennten „Welt“ auszuführen.

Alle bisherigen Visualisierungssysteme kranken an einer „Insellage.“ Die Darstellungen lassen sich nicht erweitern oder miteinander verknüpfen. Andererseits könnten aber gerade visuelle Darstellungen eine weitere Komponente zur syntaktischen Hervorhebung von Konzepten des Verarbeitungsmodells eines Programmes ausmachen. Ansätze zur Verwendung von Visualisierungen nicht nur als „Zusatz“, wie hier angedeutet, sondern als Programmierkonstrukte an sich (Visual Programming) [Haarslev & Möller 89], schildert das Kapitel 6.

2.6. Zusammenfassung: grundlegende Komponenten

Visuelle Darstellungen stehen in engem Zusammenhang mit Benutzerschnittstellen. Es ist i.a. nicht ausreichend, nur eine Rohdarstellung vorzusehen. Um den durch die Bildschirmgröße beschränkten Darstellungsraum zu strukturieren, müssen interaktive Kontroll- und Steuerungsmöglichkeiten integriert werden. Was für (wiederverwendbare) Bausteine, die dieses ermöglichen, finden in allen hier erwähnten Systemen Verwendung? Zunächst einmal ist ein Fenstersystem vorausgesetzt. Fenster werden in verschiedenen Ausprägungen verwendet: als Editierfenster, als Dialogfenster usf. Zum Aufbau von Dialogen werden Dialogkomponenten wie Schaltflächen (buttons), Tabellen, editierbare Textfelder etc. benötigt. Diese Anforderungen werden durch vielerlei Systeme unterstützt. Man kann die Systeme je nach Abstraktionsgrad unter dem Begriff Benutzerschnittstellen-Baukasten (User-Interface Toolkit) oder Benutzerschnittstellen-Entwicklungssystem (User-Interface Development System) einordnen [Dodani et al. 89], [Myers 89a]. Folgende Aufstellung gibt einen Überblick über die verschiedenen Ebenen:

Graphische Moduln

- Ausgabe von Text und geometrischen Formen (Kreise, Linien, ...)
- Ausgabe von Piktogrammen
- Verwaltung der Eingabegeräte (Tastatur, Maus)

Fenstersysteme

- Unterstützung mehrerer sich evt. überlappender Ausgabeflächen
- Clipping
- Lenkung von Eingabeströmen

Systeme zur Bereitstellung von Interaktionstechniken

- Rollbalken (scrollbars)
- Schaltflächen (buttons) in verschiedenen Ausprägungen
- Editierbare oder statische Textfelder
- Tabellen in verschiedener Form
- Menüs

Beschreibungs- und Spezifikationssysteme für Dialogformen

- graphische Anordnung von Interaktionskomponenten
- (textuelle) Spezifikation von Dialogabläufen

Ähnliche Grundelemente wie die hier aufgezählten statischen Bausteine für Dialoge, könnten auch in Systemen mit beliebigen graphischen Elementen verwendet werden. Immer wiederkehrende Elemente wie Piktogramme oder Kreise und Pfeile zur Darstellungen von Graphknoten und -bögen lassen sich in ähnlicher Weise wie Schaltflächen zu einer Art von Dialog i.w.S. zusammensetzen. Im Gegensatz zu den statischen Interaktionsobjekten wie Tabellen oder Schaltflächen sind die „graphischen Objekte" beweglich anzuordnen – sowohl innerhalb ihres (Sub-)Fensters (scrolling) als auch relativ zueinander.

Das Layout der Einzelteile einer Interaktionskomponente spielt eine wichtige Rolle. In formularorientierten Darstellungen (Abbildungen 2.1 und 2.2, s.o.) 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 (z.B. in Nexpert, vgl. Abbildung 2.3) werden die Graphknoten nach bestimmten Kriterien (z.B. als Baum oder Graph) angeordnet [Reiss 86]. Hier liegt eine lokale Beschränkung vor: die Lage der Kanten im Graph ist von der Lage der jeweils angrenzenden Knoten abhängig[1].

Einige der in den vorigen Abschnitten vorgestellten Dimensionen werden noch einmal in der Abbildung 2.5 zusammengefaßt [nach Brown 88]. Ein System zur Erstellung von Visualisierungen sollte also sowohl diskrete und inkrementelle Visualisierungsformen[2] unterstützen als auch Hilfsmittel bereitstellen, direkte und synthetische Darstellungen zu erzeugen. Dabei sind direkte Darstellungen von Daten gemäß einer strukturellen Dateninterpretation zu deuten, während sich eine synthetische Darstellung mehr auf den konzeptionellen Interpretationsaspekt bezieht.

---

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

[2] Eventuell kann auch ein dynamischer Übergang von der einen in die andere Form zwecks Beschleunigung oder Detailierung einer Visualisierung wünschenswert sein.

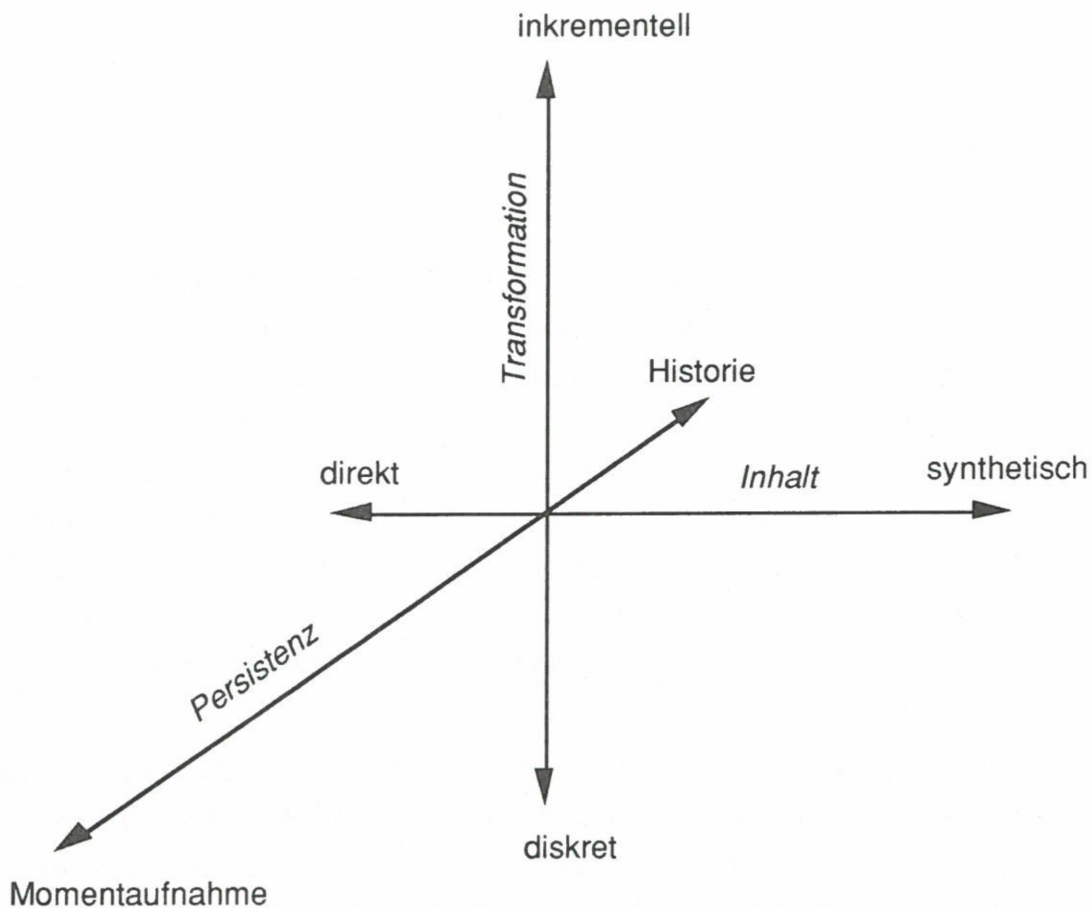


Abbildung 2.5: Dimensionen der Visualisierung (nach [Brown 88]).

Die dritte Dimension der Graphik, Persistenz, deutet an, daß es u.U. wünschenswert ist, einen Ablauf einer dynamischen Visualisierung in Form einer Historie zur Verfügung zu haben, um – in Analogie zu einem Videoband – Szenen wiederholt betrachten zu können.

3. Grundbausteine

Zur Unterstützung der Programmentwicklung können strukturelle wie auch konzeptionelle Visualisierungen verwendet werden. Aus den Überlegungen des vorigen Kapitels wird deutlich, das sich konzeptionelle Visualisierungen nur durch Erstellung „von Hand“ gewinnen lassen. Hierzu werden geeignete vordefinierte Bausteine und Konstruktionsmethoden benötigt (siehe hierzu auch [Herczeg 89]). Selbsterstellte Bauteile sollen in anderen Kontexten wiederverwendbar sein. In diesem Kapitel wird ein System vorgestellt, das beide Visualisierungsformen durch bestimmte erweiterbare Grundbausteine unterstützt. Es wurde in Allegro Common Lisp[1] auf einem Apple Macintosh implementiert und baut auf eine im Xerox Palo Alto Research Center entwickelte CLOS-Implementation auf (PCL).

Die Implementierungsgrundlage aller Programme für das Standardbetriebssystem auf dem Macintosh ist die sogenannte „Toolbox“, eine Bibliothek von PASCAL-Datentypen und Funktionen bzw. Prozeduren u.a. zur Benutzerschnittstellenprogrammierung auf dem Werkzeugkasten-Abstraktionsniveau (siehe oben).

Aufbauend auf dem Grundgerüst der Allegro-Umgebung wurden weiterführende Mechanismen realisiert. Hier sind insbesondere Anordnungsmechanismen für allgemeine graphische Objekte zu nennen. Neben vordefinierten Anordnungsalgorithmen können selbstdefinierte Schemata nahtlos integriert werden. Weiterhin wurde ein Ersatz für die in der Macintosh „Toolbox“ (bisher) nicht unterstützten hierarchisch angeordneten Fenster geschaffen (sog. Sichtbereiche mit eigenem Koordinatensystem).

In einem Sichtbereich können allgemeine graphische Objekte dargestellt werden. Für ein solches Element eines Sichtbereichs braucht nur das „Aussehen“ dieses Elements als Methode einer generischen Zeichenfunktion (i.a. prozedural) definiert zu werden. Sichtbereichselemente können beliebig aus ihrem Sichtbereich entfernt und wieder eingefügt werden, d.h. die notwendige Verwaltung (z.B. Löschung und Neuzeichnung von überschriebenen Objekten) wird von der Sichtbereichsverwaltung übernommen. Durch Spezialisierung entsprechender Methoden kann eine Verfeinerung der Algorithmen erreicht werden. Näheres hierzu schildert Abschnitt 3.5.

---

[1] Allegro CommonLisp, Apple Macintosh und Toolbox sind eingetragene Warenzeichen der Firma Apple Computer.

3.1. Integration in eine bestehende Umgebung

Die Konstruktion des in dieser Arbeit erstellten Systems zur Unterstützung der Erstellung von Visualisierungen sollte nicht von Grund auf neu erfolgen. Die schon von der zugrundeliegenden Common Lisp-Programmierumgebung gebotene Funktionalität kann in ein Gesamtsystem integriert werden. Die Common Lisp-Umgebung „Allegro“ bietet eine eigene objektorientierte Lisp-Erweiterung (ObjectLisp, [Drescher]). Diese wird u.a. dazu verwendet, die Ankopplung des Macintosh-Fenstersystems an das Allegro-Laufzeitsystem zu organisieren sowie eine Lisp-Schnittstelle zur Toolbox zu schaffen. Während in den Bereichen der elementaren Fensterverwaltung, der Ereignisverwaltung (event handling), der Verwendung von Menüs und der Gestaltung von Dialogboxen eine umfangreiche Funktionalität zur Verfügung gestellt wird, so wird einem (Anwendungs-)Programmierer in einigen Bereichen (z.B. Darstellung von Piktogrammen, deren Verschiebung etc.) abverlangt, zumindest den Kern recht „maschinennah“ zu programmieren (Zugriff auf PASCAL-„Records“, explizite Speicherverwaltung, sog. „Trap“-Aufrufe). In diesen Fällen wird also bisher keine Abstraktion z.B. durch Klassen bereitgestellt. CLOS bietet außerdem ein weiterentwickeltes Objektmodell u.a. mit selbstdefinierbaren Metaklassen. Aus programmtechnischer Sicht sind CLOS-Programme weniger fehleranfällig (in ObjectLisp treten häufig Fehler durch die Interpretation der Objekte als (dynamische) Bindungsumgebungen auf).

Durch die Verfügbarkeit einer guten, portablen CLOS-Implementation (PCL) in Common Lisp lassen sich schon jetzt Anwendungsprogramme erstellen, welche die von CLOS gebotenen objektorientierten Programmierkonzepte verwenden. Offensichtlich wird es sich als nachteilig erweisen, die Benutzerschnittstelle in ObjectLisp zu erstellen, während die Anwendungsfunktionen CLOS verwenden. Dieses gilt besonders, da zu erwarten ist, daß das Allegro-System früher oder später mit einer CLOS-Erweiterung erhältlich sein wird (wie in der Dokumentation angekündigt). Wenn also erweiterte Funktionen zur Visualisierung und Benutzerschnittstellenprogrammierung in CLOS entwickelt werden sollen, wird es unumgänglich sein, diese auf einem CLOS-Fundament aufzubauen. Daher wurde die Funktionalität des bestehenden Allegro Systems in die CLOS-Welt integriert. Die Einflechtung der CLOS-Klassen und Methoden wird kurz in Anhang A zusammengefaßt; es ist (fast) die volle Funktionalität der in der Allegro-Dokumentation (Version 1.2.2) erläuterten ObjectLisp-Klassen in CLOS-Syntax verfügbar. Dieses betrifft die Erzeugung von Menü-, Fenster- und Dialogobjekten sowie die Schnittstelle zur Verarbeitung von Ereignissen (events). Ein kurzes Beispiel für

Menüs und Dialoge enthält der nachfolgende Abschnitt. Die darauffolgenden Abschnitte schildern die neuentwickelten Werkzeuge und Beschreibungsformen. Die erstellten Teilsysteme wurden in verschieden Moduln aufgeteilt. Der Teil, der die Programmierung der Benutzerschnittstelle im engeren Sinne betrifft, wird aus dem Modul (package) `tv` exportiert. Die Anordnungsalgorithmen sind im Modul `layout` zusammengefaßt. Grundbausteine wie Graphknoten stehen im Modul `vislib`. Zur Verdeutlichung wird in allen nachfolgenden Beispielen bei jedem verwendeten Symbol aus diesen Moduln der Modulname vorangestellt. So lassen sich vordefinierte und im Rahmen des betrachteten Beispiels definierte Funktionen, Klassen etc. unterscheiden. Verwendete Symbole des CLOS-Systems sind nicht durch den Modulnamen kenntlich gemacht. Hier sollte eine explizite Kenntlichmachung nicht unbedingt erforderlich sein.

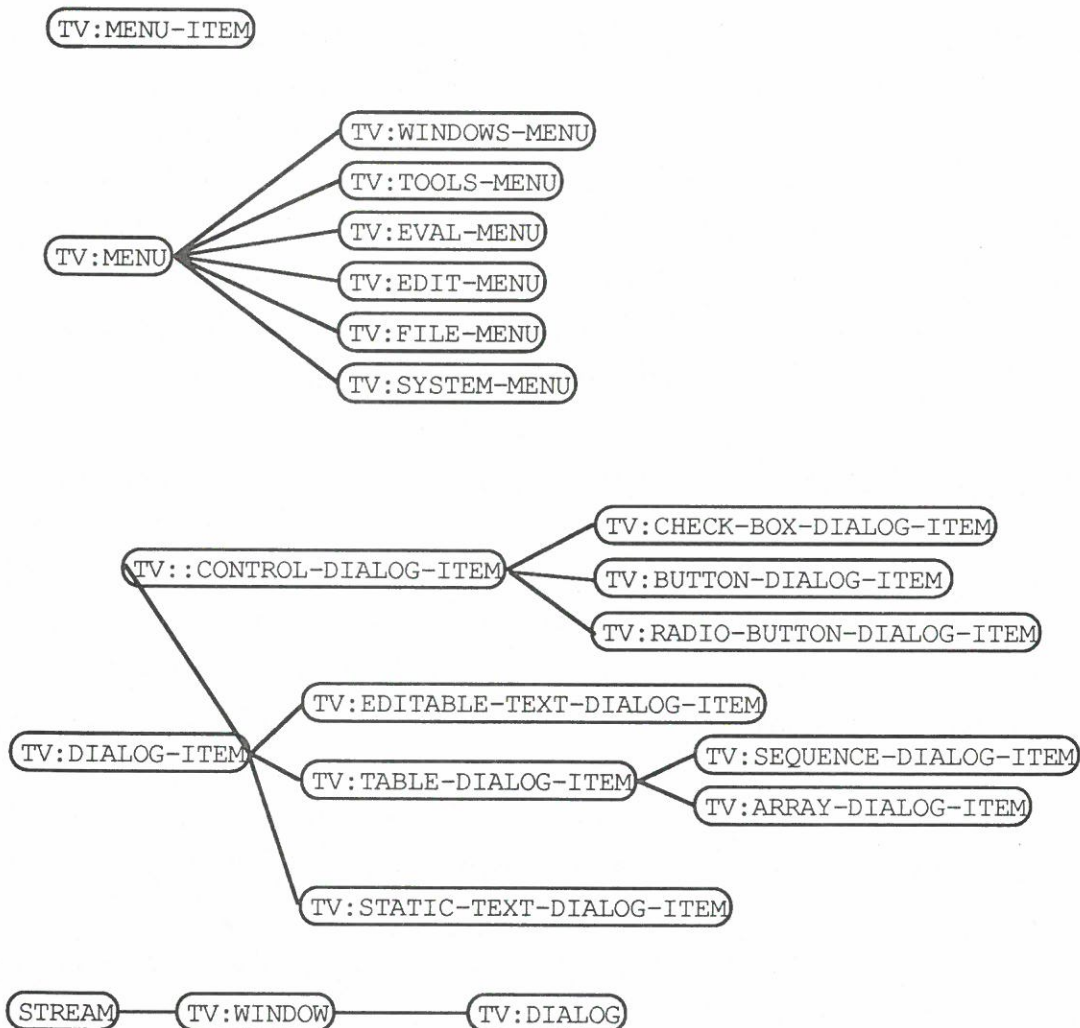


Abbildung 3.1: Ausschnitt aus der Klassenhierarchie. Enthalten sind die CLOS-Klassen, die für von der „Allegro"-Umgebung übernommene ObjectLisp-Klassen stehen.

Abbildung 3.1 vermittelt einen Eindruck über die von Allegro übernommenen Klassen, die einen Anschluß an die für die Erstellung von Visualisierungswerkzeugen wichtigen Teile der „Toolbox" organisieren. Es stehen Klassen zur Erzeugung von Menüs (`tv:menu`) sowie von Menüauswahlelementen (`tv:menu-item`) zur Verfügung. Für vordefinierte Menüs existieren eigene Klassen (`tv:windows-menu` bis `tv:system-menu`). Interaktionsdialoge (`tv:dialog`) enthalten statische und editierbare Textfelder (`tv:static-text-dialog-item`, `tv:editable-text-dialog-item`), Tabellen zur Darstellung von Sequenzen (`tv:sequence-dialog-item`) oder Matrizen (`tv:array-dialog-item`) oder auch Schaltflächen (buttons) in verschiedenen Ausprägungen.

3.2. Ein Anwendungsbeispiel: ein Polygoneditor

Um einen kurzen Eindruck von der Programmierung der Menüs, Fenster und Dialoge zu vermitteln, sei ein kleines Anwendungsbeispiel aus dem Bereich eines Zeichenprogrammes geschildert. In einem Fenster sollen Polygonzüge erstellt werden (siehe Abbildung 3.2). Dazu werden mit der Maus durch Klicken und Gedrückthalten der Umschalttaste die Stützpunkte des Polygonzuges definiert. Durch Doppelklick wird die Eingabe eines Polygons beendet. Da ein Doppelklick unbeabsichtigt ausgeführt werden könnte, erscheint zur Bestätigung ein Dialog, der es gestattet, das Polygon zu erzeugen oder den Doppelklick zu ignorieren (Abbildung 3.3). Außerdem kann durch Ankreuzen eines Auswahlfeldes bestimmt werden, ob die Stützpunkte gelöscht werden sollen oder für das nächste Polygon zu verwenden sind.

Dieses Anwendungsbeispiel ist nicht primär auf Visualisierungsmethoden bezogen. Zur Erstellung einer Visualisierung werden noch andere Werkzeuge benötigt, die in Kapitel 5 beschrieben sind. Es sollte allerdings deutlich werden, daß die in diesem Beispiel erläuterten Techniken, wie überhaupt das ganze Themengebiet der Benutzerschnittstellenprogrammierung, auch für den Aufbau einer Visualisierung in hohem Maße relevant sind. Im weiteren Verlauf werden Darstellungsmittel geschildert, die sich direkt für Visualisierungen einsetzen lassen. Wichtiges Merkmal ist hierbei die objektorientierte Sicht: ob Menüeintrag oder Polygon, beide können aus ihrem Kontext ohne Aufwand entfernt oder zu diesem hinzugefügt werden. Entfernen eines Polygons bedeutet hierbei nicht primär ein Löschen der Darstellungsfläche, sondern eine „Manipulation" eines Polygonobjekts. Bei der Erstellung von Visualisierungen läßt sich so der Programmieraufwand in überschaubaren Grenzen halten. Die in dieser Arbeit

entwickelten Methoden sind auch für eine Benutzerschnittstellenprogrammierung einsetzbar.

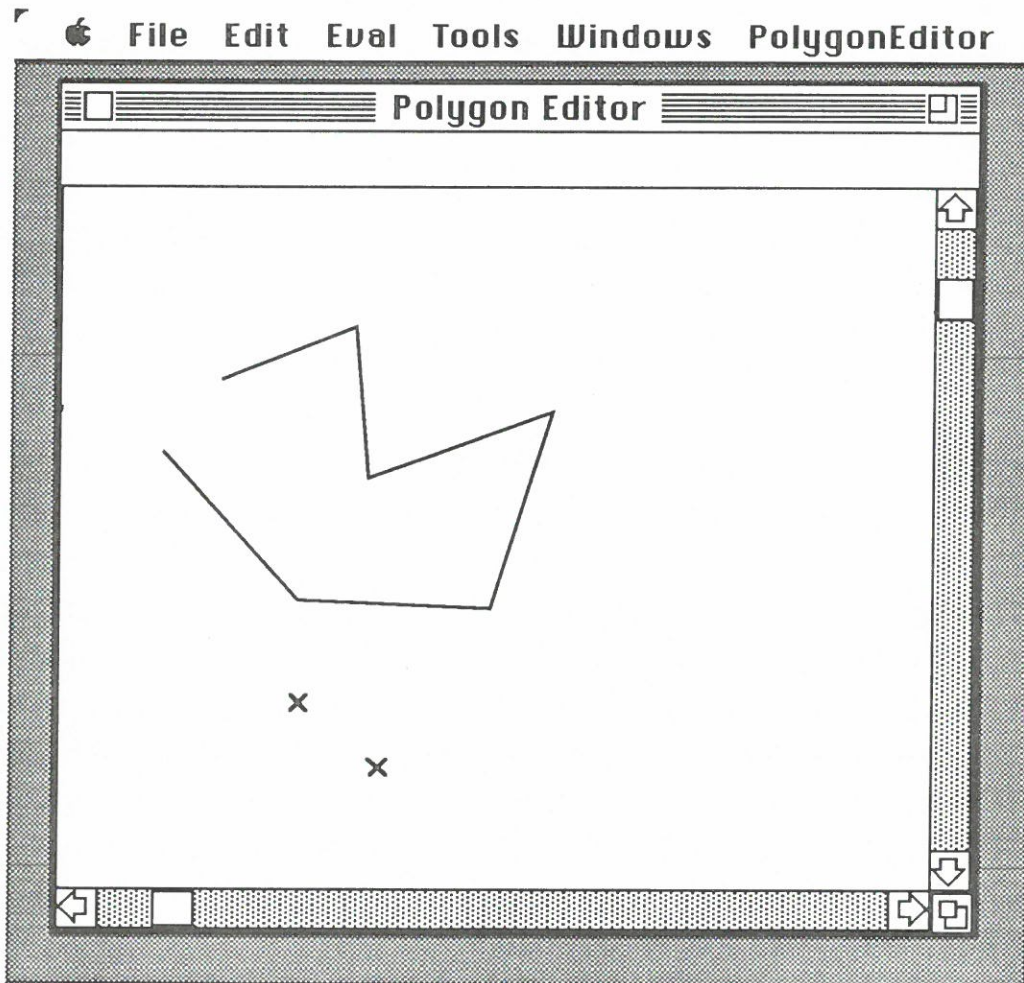


Abbildung 3.2: Ein Polygoneditor-Dialog.

Dieses Minibeispiel erhebt nicht den Anspruch, ergonomischen Anforderungen von Benutzerschnittstellen (von Zeichenprogrammen) gerecht zu werden. Es dient der Veranschaulichung der Möglichkeiten und Einführung in die Konzepte des erstellten Systems, wobei der Übersichtlichkeit halber nur rudimentäre Operationen möglich sind. Zunächst sei die Erstellung eines Menüs mit einem Menüauswahlelement (Synonym: Menüelement) zur Erzeugung einer Polygonwelt betrachtet.

```
(defvar *polygon-editor-menu*
  (tv:make-menu :menu-title "PolygonEditor"))
```

Menüs sind Instanzen einer Klasse und werden durch Evaluierung der Funktion `tv:make-menu` erzeugt. Es können verschiedene Schlüsselwortargumente z.B. zur Festlegung eines Menütitels angegeben werden. Anschließend wird das Menü in der Menüleiste installiert.

```
(tv:menu-install *polygon-editor-menu*)
```

Ein Menüelement wird ebenfalls als Objekt repräsentiert und kann auf einfache Weise erzeugt, mit einem Titel versehen und zu einem Menü hinzugefügt werden.

```
(defvar *new-polygon-menu-item*
  (tv:make-menu-item :menu-item-title "New"))

(tv:add-menu-items *polygon-editor-menu*
                   *new-polygon-menu-item*)
```

Das Menüelement muß nun noch zum Leben erweckt werden, d.h. nach dem Auswählen des Elements durch die Maus soll eine bestimmte Funktion evaluiert werden. Das Laufzeitsystem evaluiert bei jedem Anklicken eines Menüelements die generische Funktion `tv:menu-item-action`. Das Standardverhalten besteht darin, die Menüauswahl zu ignorieren, es kann jedoch für das oben an `*new-polygon-menu-item*` gebundene Menüelement eine spezielle Methode definiert werden. Hierbei kann man sich die Möglichkeit zunutze machen, daß im CLOS-Formalismus Methoden nicht nur für Klassen, sondern auch für einzelne Lisp-Objekte definiert werden können.

```
(defmethod tv:menu-item-action
           ((command-item (eql *new-polygon-menu-item*)))
  "Erzeugung eines Polygoneditors."
  (make-polygon-editor))

(defun make-polygon-editor ()
  "vorläufige Fassung."
  (print "Editor not yet implemented."))
```

Beim Anklicken des Menüelements wird nun diese Methode evaluiert, d.h. es wird die Funktion `make-polygon-editor` ausgeführt, die ein Dialogfenster zur interaktiven Definition von Polygonen bereitstellt.

Ein Dialog ist ein Fenster mit speziellen Methoden zur Verwaltung von sog. Dialogelementen. Dialogelemente sind Bausteine wie Schaltflächen, Tabellen, editierbare bzw. statische Texte usf. Zur Erzeugung eines Polygoneditors wird jedoch eine erweiterte Funktionalität benötigt. Die allgemeinen Bausteine wurden in dieser Arbeit entwickelt. Die Konzepte sind in den anschließenden Kapiteln geschildert. An dieser Stelle sei jedoch noch der nach einem Doppelklick verwendete Dialog betrachtet (siehe

Abbildung 3.3). Der Dialog wird als Subklasse einer vordefinierten Klasse `tv:dialog` implementiert.

```
(defclass connect-dialog
  (tv:dialog)
  ((polygon-editor :accessor manipulated-polygon-editor)))
```

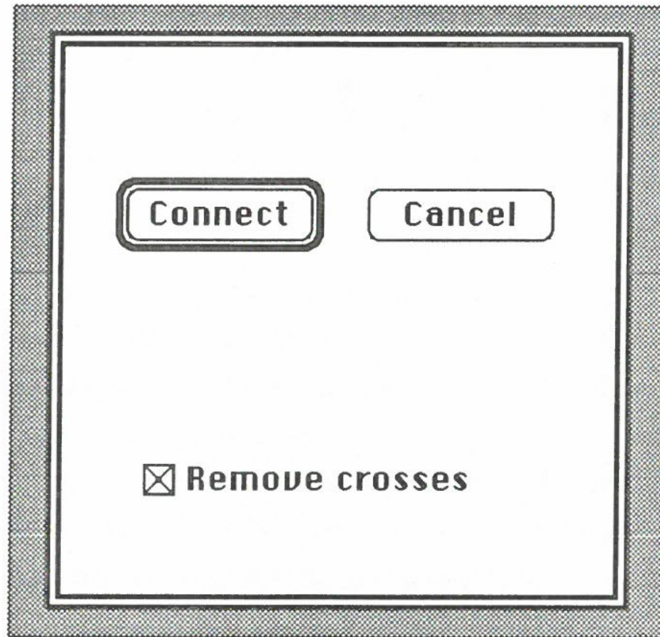


Abbildung 3.3: Dialog zur Bestätigung der Verbindung von Stützpunkten zu einem Polygon.

In diesem Dialog sind drei Dialogelemente enthalten. Auf Wunsch können die Stützpunkte des Polygons nach dem Doppelklick für das nächste Polygon weiterverwendet werden. Dieses kann durch Anklicken eines Ankreuzkästchens festgelegt werden (siehe Abbildung 3.3). Eine solches Ankreuzkästchen kann aus vorgefertigten Teilen gewonnen werden.

```
(setf *remove-crosses-check-box*
  (tv:make-dialog-item :class 'tv:check-box-dialog-item
                       :dialog-item-text "Remove crosses"
                       :dialog-item-size #@(150 20)
                       :dialog-item-position #@(42 149)))
```

Des weiteren werden zwei Schaltflächen zur Akzeptierung (`*connect-button*`) und zum Rückgängigmachen (`*cancel-button*`) des Polygonabschlusses verwendet. Ähnlich wie bei dem oben erzeugten Menüelement soll das (asynchrone) Anklicken einer Schaltfläche zur Evaluierung einer Funktion führen. Das Laufzeitsystem evaluiert bei

jedem Anklicken eines Dialogelements die generische Funktion `tv:dialog-item-action`. Das gewünschte Verhalten kann jeweils durch Definiton einer speziellen Methode erreicht werden.

```
(setf *connect-button*
  (tv:make-dialog-item :class 'tv:button-dialog-item
                       :dialog-item-text "Connect"
                       :dialog-item-size #@(70 20)
                       :dialog-item-position #@(22 44)))

(defmethod tv:dialog-item-action ((item (eql *connect-button*)))
  "Verbinden der Stützpunkte zu einem Polygon."
  (connect-crosses
   (manipulated-polygon-editor (tv:own-dialog item)) ; Polygondialog
   (tv:check-box-checked-p *remove-crosses-check-box*)) ; Stützpunkte
                                                         ;  beibehalten ?
  (tv:return-from-modal-dialog t))             ; Rückgabe der Kontrolle
```

Diese Methode delegiert die Polygonerstellung an die Funktion `connect-crosses` mit Aktualparametern für den manipulierten Polygoneditor und der Information, ob die Stützpunkte beibehalten werden sollen.

```
(setf *cancel-button*
  (tv:make-dialog-item :class 'tv:button-dialog-item
                       :dialog-item-text "Cancel"
                       :dialog-item-size #@(70 20)
                       :dialog-item-position #@(118 44)))

(defmethod tv:dialog-item-action ((item (eql *cancel-button*)))
  (tv:return-from-modal-dialog :cancel))
       ; Rückgabe d. Kontr. ohne Aktion


(setf *connect-dialog*
  (tv:make-dialog :class 'connect-dialog
                  :window-size #@(200 200)
                  :window-type :double-edge-box
                  :window-position :centered
                  :window-show nil)) ; Das Dialogfenster wird
                                     ;        nur bei Bedarf gezeigt.
```

Für die Anwendung des Dialogs wird die Funktion `dialog-interaction` definiert. Sie öffnet das Dialogfenster und gibt die Kontrolle durch Evaluierung der Funktion `tv:modal-dialog` an das Dialogfenster ab, d.h. weitere Interaktionen eines Benutzers werden durch dieses Dialogfenster verwaltet.

```
(defun dialog-interaction (connect-dialog polygon-editor)
  (setf (manipulated-polygon-editor connect-dialog) polygon-editor)
  (tv:modal-dialog connect-dialog nil)) ; Übergabe der Kontrolle
```

Die Dialogelemente müssen noch dem Dialog mitgeteilt werden.

```
(tv:add-dialog-items *connect-dialog*
                     *connect-button*
                     *cancel-button*
                     *remove-crosses-check-box*)
```

Es erweist sich als sehr umständlich, die Positionen der Dialogelemente explizit zu bestimmen. Daher existieren Teilsysteme von Entwicklungsumgebungen, die es erlauben, vordefinierte Dialogelemente interaktiv durch Verwendung von Graphik nach dem WYSIWYG-Prinzip anzuordnen (Resource Editoren, Interface Builder). Der erstellte zugehörige „Code“ wird entweder in einer sog. Resource gespeichert oder als Programmtext in eine Datei ausgegeben. Der Nachteil dieser Systeme ist die Einschränkung auf vordefinierte Elemente[1]. Weiterhin muß beim späteren Hinzufügen eines weiteren Dialogelements ein erneuter Editiervorgang erfolgen. Außerdem wird die Dialoggröße als statisch vorausgesetzt. Es lassen sich also in den meisten System keine Einschränkungen ausdrücken, wie etwa: ein Dialogelement soll in der Mitte eines Dialogs plaziert werden. Weiterhin kann die Größe der Dialogeinträge nicht an die Platzverhältnisse angepaßt werden. In dieser Arbeit wurden Mechanismen entwickelt, die eine flexible Anordnungsbeschreibung ermöglichen und die hier genannten Nachteile anderer Systeme vermeiden.

3.3. Layoutangaben

Das zugrundeliegende Modell der Anordnung von Objekten ist in Anlehnung an das „Box-and-glue-Modell“ des TEX-Satzsystems konzipiert [Knuth 79]. 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. Die Anordnung kann auch als Zusammenkleben (glueing) der Elemente innerhalb einer Box aufgefaßt werden. Als konkretes Beispiel für eine Box wurde im vorigen Abschnitt ein Dialogfenster genannt, in dem Dialogelemente anzuordnen sind. 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

[1] Einige Systeme sind zwar erweiterbar, doch ist eine Erweiterung alles andere als offensichtlich.

Interpreterfunktionen bleiben unverändert. Durch Verwendung dieses abstrakten Protokolls ist außerdem ein leichtere Portierung der Layoutfunktionen gewährleistet.

Ein Layoutmuster ist zunächst eine Liste mit besonderen Einträgen (box-specifier): 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 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. der Ausdehnung versehen werden.

3.3.1. 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.

3.3.2. Layoutmuster für Füller

Angaben zu Füllabständen haben folgende vollständige Form: `(:filler :min` *m* `:max` *n*`)`, `:min` und `:max` sind optional.

Schlüsselwort	Typ	Voreinstellung	Bedeutung
`:min`	*integer*	0	minimale Füllgröße
`:max`	*integer*	*calculated*[1]	maximale Füllgröße

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. Es werden hierzu die Funktionen `tv:recommended-hbox-size` und `tv:recommended-vbox-size` verwendet. `:as-needed` gilt nur für `:vbox` und `:hbox` Layoutangaben.

3.3.3. Eingespleißte Layoutangaben

Mit den bisher definierten Layoutangaben läßt sich auf recht einfache Weise ein Layout für *vorgegebene* Boxelemente definieren. Falls jedoch eine Liste von Boxelementen – die Anzahl der Elemente ist also variabel – mit einem Layoutmuster plaziert wird, ist es notwendig, diese Liste von Boxelementen zusammen mit anderen Layoutangaben in ein Layoutmuster einzupassen. Eine Einspleißung einer Liste von Layoutangaben kann durch Verwendung des „Spleißmusters“ `(:splice` *expression*`)` erreicht werden. Der Wert von *expression* muß eine Liste von Layoutangaben sein; diese Layoutangaben werden dann so behandelt als stünden sie direkt im Layoutmuster.

3.3.4. Syntax der Layoutmuster

Die Syntax der Layoutmuster sei durch folgende EBNF-Grammatik gegeben (Terminale in Fettdruck, nicht weiter erklärte Nicht-Terminale kursiv):

```
<layout-pattern>        ::=  <fbox-pattern>
                             | <vbox-pattern>
                             | <hbox-pattern> .

<fbox-pattern>          ::=  ( :fbox <box-size-specification> <item> ) .

<vbox-pattern>          ::=  ( :vbox   <box-size-specification>
                                       { <layout-specification>
```

[1] Gesamtbreite bzw. -höhe der umschließenden Box

```
                                                | <layout-pattern>
                                                | <item>
                                                | <splice> } ) .

<hbox-pattern>              ::=   ( :hbox   <box-size-specification>
                                            { <layout-specification>
                                              | <layout-pattern>
                                              | <item>
                                              | <splice> } ) .

<layout-specification>      ::=   <size-specification>
                                  | <filler-specification>

<box-size-specification>    ::=   ( [ :width <layout-specification> ]
                                    [ :height <layout-specification> ] ) .

<size-specification>        ::=   <integer> | <rational> | <float> .

<filler-specification       ::=   :filler
                                  | ( :filler      [ :min <filler-size-spec> ]
                                                   [ :max <filler-size-spec> ]) .

<filler-size-spec>          ::=   <integer>
                                  | :as-needed[1] .

<item>                      ::=   <S-expression> .

<splice>                    ::=   ( :splice <S-expression>) .
```

Terme eines Layoutmusters, denen in der Grammatik ein nicht weiter erklärtes (kursives) Nicht-Terminal entspricht, werden vor dem Parsen des Musters evaluiert. Die Terminale `:vbox`, `:hbox`, `:fbox`, `:width` und `:height` sind als Makros definiert, so daß eine Quotierung des Musters nicht erforderlich ist. Ein Beispiel für ein Layoutmuster (die Variablen `item-1` bis `item-4` seien gebunden) sieht folgendermaßen aus:

```
(:vbox ()
       1/8
       (:hbox (:height 50)
              20
              item-1
              (:vbox (:width 60)
                     item-2
                     :filler
                     item-3
                     1/3))
       :filler
       (:hbox (:height 12)
              :filler
              :filler
              item-4
              :filler)
       :filler)
```

[1] Nur für `:vbox` und `:hbox` Layoutangaben

Dadurch ergibt sich die ungefähre Anordnung aus Abbildung 3.4 (die Einrückung der Boxen erfolgt nur zur besseren Übersicht, die Größe der äußeren Box (vbox) sowie die Ausdehnung der Beispielboxelemente (`item-1` bis `item-4`) ist willkürlich festgelegt):

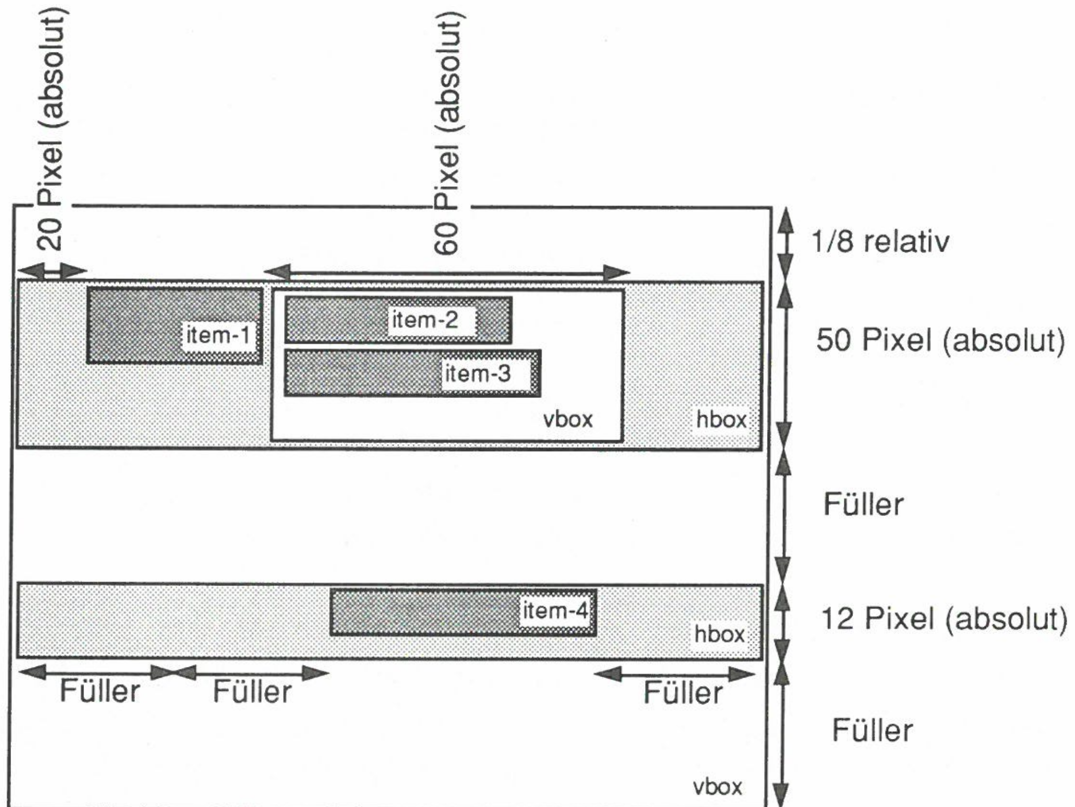


Abbildung 3.4: Schematische Darstellung einer Anordnung mithilfe eines Layoutmusters.

Zur Anpassung der Größe des Elementes `item-1` könnte folgendes Muster dienen:

```
(:vbox ()
       1/8
       (:hbox (:height 50)
              20
              (:fbox (:width 40 item-1))
              (:vbox (:width 60)
                     item-2
                     :filler
                     item-3
                     1/3))
       :filler
       (:hbox (:height 12)
              :filler
              :filler
              item-4
              :filler))
       :filler)
```

Es ergibt sich damit das in Abbildung 3.5 gezeigte Layout:

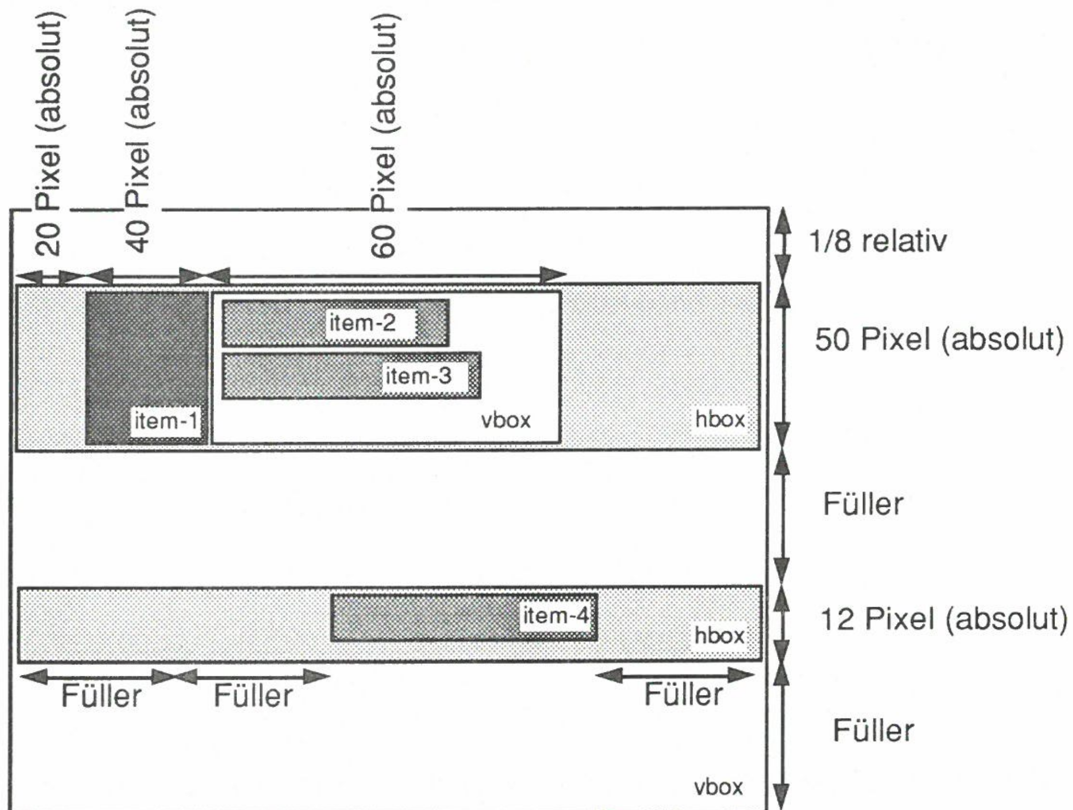


Abbildung 3.5: Verwendung einer `:fbox`.

3.3.5 Verbesserung der Definition des Beispieldialoges mithilfe von Layoutangaben

Mithilfe der im vorigen Kapitel aufgeführten Layoutangaben läßt sich die Definition des Abfragedialogs `*connect-dialog*` wesentlich vereinfachen. Die expliziten Positionsangaben können durch relative Angaben innerhalb eines Layoutmusters ersetzt werden. Dieses ist insbesondere vorteilhaft, wenn sich die Fenstergröße des Dialogfensters eventuell noch ändern kann oder absehbar ist, daß noch weitere Dialogelemente hinzukommen. Um eine Verwendung von Layoutmustern zu ermöglichen, wird für die Klasse `connect-dialog` eine neue, vordefinierte Superklasse festgelegt.

```
(defclass connect-dialog
  (layout:layouted-dialog)
```

```
((polygon-editor :accessor manipulated-polygon-editor)))
```

Das Layoutmuster wird z.B. mit dem Schlüsselwort `:layout` bei der Instantiierung des Dialogs festgelegt.

```
(setf *connect-dialog*
  (tv:make-layouted-dialog
                    :class 'connect-dialog
                    :window-size #@(200 200)
                    :window-type :double-edge-box
                    :window-position :centered
                    :window-show nil
                    :layout (:vbox ()
                                   :filler
                                   (:hbox ()
                                          :filler
                                          *connect-button*
                                          :filler
                                          *cancel-button*
                                          :filler)
                                   :filler
                                   (:hbox (:height :as-needed)
                                          :filler
                                          *remove-crosses-check-box*
                                          :filler)
                                   :filler)))
```

Die Höhe der unteren horizontalen Box wird durch eine Verwendung der Höhenangabe `:as-needed` automatisch an das Dialogelement `*remove-crosses-check-box*` angepaßt. Die Dialogkomponenten sind die gleichen wie oben; die Positionsangaben `:dialog-item-position` entfallen.[1] Die Dialogelemente brauchen nicht wie oben explizit mit der Funktion `tv:add-view-items` dem Dialogobjekt mitgeteilt zu werden. Dieses erfolgt durch die Layoutanordnung.

3.4. Dialogfenster und Sichtbereiche

In der Macintosh-Toolbox ist eine Verwendung von hierarchischen Fenstern nicht vorgesehen. Selbst Rollbalken müssen „selbsttätig" erzeugt und so ausgerichtet werden, daß sie – sofern gewünscht – am Fensterrand erscheinen. Um nun allgemeine graphische Objekte in beliebigen Bereichen eines Fensters in einem eigenen Koordinatensystem etc. mit der Möglichkeit eines Rollens darzustellen, wurde eine besondere Klasse von Dialogelementen geschaffen: Sichtbereiche. Sichtbereiche lassen sich als einstufig hierarchische Fenster verstehen, gehen aber ansonsten in ihrer Funktionalität weit über die eines (Sub-)Fensters hinaus (z.B. Neuzeichnung und Rollen des sichtbaren Inhalts).

---

[1] Die oben definierten Methoden für `tv:dialog-item-action` müssen erneut evaluiert werden, da sie nur für die oben erzeugten Instanzen, nicht jedoch für die evt. neu erzeugten Instanzen eine Spezialisierung der entsprechenden generischen Funktionen vornehmen.

Sie bilden die Grundlage für die Darstellung und Verwaltung sog. Sichtbereichselemente. Sichtbereichselemente können z.B. Knoten und Kanten eines Graphen sein. Sie sind analog zu den Dialogelementen als allgemeine wiederverwertbare Bausteine zu verwenden.

3.4.1. Ein Polygoneditor als Sichtbereichsinstanz

Ein Polygoneditor, um das Beispiel von oben wieder aufzugreifen, ist ein Dialogobjekt, das einen speziellen Sichtbereich enthält, der eine Definiton von Polygonen erlaubt. Der Sichtbereich wird innerhalb des Dialoges mit einer `:fbox` angeordnet. Für Erweiterungen wird in diesem Beispiel oberhalb des an `*polygon-view*` gebundenen Sichtbereichs ein Teilbereich des Dialogs freigelassen. Dieser Platz könnte in einer weiteren Ausbaustufe des Polygoneditors mit Schaltflächen o.ä. gefüllt werden. Durch Verwendung einer `:fbox` werden die Rollbalken am rechten bzw. unteren Fensterrand plaziert. Bei Vergrößerung des Fensters werden die Dialogelemente automatisch neu ausgerichtet.

```
(defclass polygon-editor
  (layout:layouted-dialog)
  ((current-crosses :accessor polygon-crosses :initform nil)
   (polygons :accessor polygons :initform nil)
   (editor-view :accessor editor-view))
  (:documentation "A polygon editor dialog."))

(defclass polygon-view[1]
  (tv:view-dialog-item)
  ()
  (:documentation "A polygon editor view. Crosses for polygons may be
                   generated by clicking into the view.
                   Double-clicking connects existing
                   crosses to a polygon."))

(defun make-polygon-editor ()
  (let* ((view (tv:make-dialog-item :class 'polygon-view
                                    :auto-scrolling t
                                    :bordered-p t
                                    :scroll-bars ':both))
         (editor (layout:make-layouted-dialog
                  :class 'polygon-editor
                  :window-size #@(342 300)
                     ; Seitenverhältnis sqrt(2):1
                   :layout (:vbox ()
                                  20
                                   (:fbox () view))
                  :window-title "Polygon Editor"
                  :window-type :document-with-zoom)))
```

[1] Die Definition einer eigenen Klasse für den Polygoneditorsichtbereich ist notwendig, da im weiteren Verlauf noch Methoden für diese spezielle Sichtbereichsklasse benötigt werden.

```
    (setf (editor-view editor) view)
    editor))
```

Der erzeugte Sichtbereich wird mit einem Rahmen versehen (`:bordered-p t`). Er führt ein automatisches Rollen durch, wenn ein Sichtbereichselement durch Verschiebung aus der Darstellungsfläche herausbewegt wird (`:auto-scrolling t`) und sieht ein horizontales und vertikales Rollen vor (`:scrollbars ':both`). Ein Rollen könnte auch ganz ausgeschlossen oder auf eine Richtung beschränkt werden.

3.4.2. Vergleich mit bestehenden Systemen

Die Aufgabe eines Sichtbereichs besteht in der Verwaltung von Bildlauf, verbunden mit der Berechnung der Rollbalkengrößen und -positionen. Weiterhin verfügt jeder Sichtbereich über ein eigenes Koordinatensystem.[1] Von Entwicklungssystemen zur Benutzerschnittstellenprogrammierung werden diverse unterschiedliche Techniken zur Realisierung solcher Sichtbereiche angeboten. Für den Apple-Macintosh, dessen Fenstersystem[2] keine hierarchischen Fenster unterstützt, wird zur Programmierung häufig das Entwicklungssystem „MacApp" verwendet [Schmucker 86]. Als Analogon zum Sichtbereich wird in diesem System ein Typ „Frame" angeboten. Instanzen dieses Typs können in Fenstern dargestellt werden. Für den Inhalt eines solchen „Frames" ist eine Instanz einer Objektklasse „View" verantwortlich. Es entsteht hier eine Dreiteilung: in Fenstern werden „Frames" angeordnet, in diesen wiederum ist jeweils ein „View" für die Darstellung verantwortlich. Durch ein festgelegtes Nachrichtenprotokoll ist die Kommunikation der Komponenten untereinander festgelegt. Layoutangaben von „Frames" in Fenstern können nicht deklarativ angegeben werden.

In hierarchisch organisierten Fenstersystemen wie im Smalltalk-System, im X-Windows-System [X-Windows] oder im Symbolics-Fenstersystem [Symbolics 88a] können Sichtbereiche schlicht durch Subfenster realisiert werden. Die Tiefe kann dabei beliebig sein. Im Gegensatz zum X-Windows-Fenstersystem kann jedoch im Fenstersystem der Symbolics-Rechner ein Subfenster nicht teilweise außerhalb seines Vaterfensters angeordnet werden, was als Einschränkung zu sehen ist.

3.5. Sichtbereichselemente

---

1 Alle Koordinatensysteme sind linkshändige Koordinatensysteme.

2 Hier ist das Standardbetriebssystem angesprochen.

Die Bestandteile eines Sichtbereichs sollten analog zu den Bestandteilen (Elementen) eines Dialoges als eigenständige Objekte und nicht als eine Ansammlung von Pixeln aufgefaßt werden. Ähnliche Ideen sind in einem Aufsatz „There's More to Menu Systems Than Meets the Screen“ von H. Lieberman [Lieberman 85] aufgeführt:

„...each graphical Object displayed is not just a collection of lines, points and shaded areas, etc. but a visual representation of some object of interest in the problem domain.“

Innerhalb dieser Arbeit wurde ein System erstellt, das gestattet, in einem Sichtbereich beliebig viele solcher Sichtbereichselemente durch einfache Operationen anzuordnen. Die Einordnung eines Sichtbereichselements ist vergleichbar mit der Einordnung eines Menüeintrags in ein Menü oder eines Dialogelements in einen Dialog. Hier wurde versucht, für diese verschiedenen Bereiche eine weitgehend konsistente Sicht zu schaffen, wobei zu bedenken ist, daß die Behandlungsweise der Menüs und Dialoge von der Common Lisp-Programmierumgebung Allegro übernommen wurde.

Ein Sichtbereichselement läßt sich auf einfache Weise wieder aus seinem Sichtbereich entfernen. Kennzeichnend dabei ist, daß das erstellte Verwaltungssystem berechnet, welche Objekte, die noch im Sichtbereich verbleiben, neu gezeichnet werden müssen. Das Aussehen eines einzelnen Sichtbereichsobjekts kann durch Subklassenbildung und Methodenspezialisierung der generischen Zeichenfunktion (z.B. durch Verwendung von `:after` Methoden) angepaßt werden. Ein Sichtbereichselement ist gekennzeichnet durch eine Position innerhalb seines Sichtbereichs und einer rechteckigen Überdeckungsfläche mit horizontalen und vertikalen Kanten. Dieses Rechteck wird auch als das Zeichenrechteck eines Sichtbereichselements bezeichnet.

Wird ein Element aus seinem Sichtbereich entfernt, so wird vom System die Fläche, die das Element überdeckt, mit der Fensterhintergrundfarbe gefüllt. Für die hierdurch zerstörten anderen Sichtbereichselemente wird anschließend die generische Zeichenfunktion evaluiert. Spezielle Sichtbereichselemente wie z.B. Kanten in einem Graphen können eine große Rechteckfläche besitzen, obwohl die eigentliche Körperform diese Fläche nicht ausfüllt. Um nun zu verhindern, daß es zu einem unschönen „Blitzeffekt“ kommt, wenn z.B. bei Entfernung einer Kante aus ihrem Sichtbereich das zugehörige Rechteck gelöscht und dadurch ein anderes Objekt zerstört wird, obwohl es von der „eigentlichen“ Kante nicht überdeckt wurde, kann auch die vom System evaluierte generische Funktion zum Löschen eines Sichtbereichselements mit einer geeigneten Methode spezialisiert werden. In einer Prototypphase ist dieses jedoch nicht notwendig. Das vom System zur Verfügung gestellte Standardverhalten reicht in vielen Fällen für eine erste Anwendung aus, kann aber (in gewissem Rahmen) spezialisiert

werden. Dieses gilt nicht nur für Zeichen- bzw. Löschfunktionen, sondern auch für vordefinierte Funktionen, die eine Mausinteraktion durch einen Benutzer steuern (z.B. Verschiebung und Markierung von Sichtbereichselementen).

3.5.1. Fortsetzung des Beispiels: Polygonstützpunkte als Sichtbereichselemente

Eine Fortsetzung des in den vorigen Kapiteln begonnenen Beispiels verdeutlicht die Ausnutzung und Erweiterung des Standardverhaltens von Sichtbereichselementen. Ein Polygonstützpunkt wird durch Mausklick[1] erzeugt und als Kreuz innerhalb eines Sichtbereichs dargestellt. Ein Kreuz soll nachträglich noch verschiebbar sein. Unter Verwendung von vordefinierten Klassen sieht die Klassendefinition für die Kreuze inklusive einiger Hilfsfunktionen wie folgt aus.

```
(defclass cross-view-item
  (tv:moveable-view-item-mixin  tv:view-item)
  ()
  (:documentation "A cross of size 6x6 to be used as a vertex of a
                   polygon."))

(defun make-cross (p)
  "Make a cross view-item out of a point (in view coordinates)."
  (make-instance 'cross-view-item
                 :view-item-position (add-points p #@(-3 -3))
                 :view-item-size #@(6 6)))

(defun cross-point (cross)
 "Returns the cross-point of the diagonals in view coordinates."
 (tv:add-points (tv:view-item-position cross) #@(3 3)))
```

Die Basisklasse aller Sichtbereichselemente `tv:view-item` wird mit einer Zusatzklasse `tv:moveable-view-item-mixin` zu einer neuen Klasse `cross-view-item` kombiniert. Hier wird ein Konstruktionsprizip deutlich: erweitertes Verhalten von Sichtbereichselementen wie Verschiebbarkeit kann durch Verwendung von multipler Vererbung zu dem Standardverhalten „hinzugemischt" werden.[2] Zunächst ist für die Klasse `cross-view-item` noch das Aussehen der Instanzen zu bestimmen. Dieses kann durch eine spezielle Methode erfolgen. Hier liegt das gleiche Konstruktionsprinzip vor wie bei den Methoden zur Verwaltung von Menüelementen. Das System evaluiert bei der Auswahl eines Elements eine generische Funktion. Für ein

---

[1] bei gedrückter Umschalttaste

[2] Der Ausdruck `mixin` stammt aus dem Flavors-System [Symbolics 88a] und deutet darauf hin, daß die so bezeichnete Klasse primär innerhalb einer multiplen Vererbung zum Hinzufügen von speziellem Verhalten zu Standardverhalten gedacht ist. Klassen, die mit einem Suffix `mixin` benannt sind, werden als abstrakte, nicht instantiierbare Klassen betrachtet.

spezielles Element kann das gewünschte Verhalten durch Definition einer Methode für die generische Funktion bestimmt werden. Bei Sichtbereichselementen wird das „Aussehen" durch folgende `:after` Methode festgelegt.

```
(defmethod tv:view-item-draw
                  :after ((item cross-view-item) view dialog)
  (draw-cross dialog (cross-point item)))
```

Die Zeichenfunktion wird in einem Kontext mit entsprechenden Koordinatentransformationen evaluiert. Durch Position und Größenvektor eines Sichtbereichselements ist ein Rechteck definiert. Das System setzt den Klippbereich während der Evaluierung der Zeichenfunktion so, daß nur innerhalb dieses Rechteck gezeichnet werden kann (siehe auch Abschnitt 3.6.3).

```
(defclass polygon-view-item
  (tv:moveable-view-item-mixin tv:view-item)
  ((polygon-points :initarg :polygon-points
                   :accessor polygon-points))
  (:documentation "A polygon consists of a list of points
stored in a normalized form, i.e. relative to (0,0). This
is necessary because polygons can be moved around."))

(defun make-polygon (points)
  "Make a polygon view-item out of a list of points
   (in view coordinates)."
  (let* ((position (compute-polygon-position points))
         ;; Position of the smallest surrounding rectangle
         ;; with horizontal and vertical edges, respectively.
         (size (compute-polygon-size points position)))
         ;; Size of this rectangle.
    (make-instance 'polygon-view-item
                   :view-item-position position
                   :view-item-size size
                   :polygon-points (normalized-points points
                                                       position))))

(defmethod tv:view-item-draw :after ((item polygon-view-item)
                                      view dialog)
  (let* ((position (tv:view-item-position item))
         (points (mapcar #'(lambda (p)
                             (tv:add-points position p))
                         (polygon-points item))))
         ;; Map the normalized points to the right niveau in the view.
    (draw-polygon dialog points)))
    ;; delegation to an obvious drawing routine
```

Zur interaktiven Erzeugung der Stützpunkte durch Mausinteraktion kann für die vom System bei jedem Mausklick evaluierte generische Funktion `tv:dialog-item-click-event-handler` eine Methode für den Polygonsichtbereich definiert werden.

```
(defmethod tv:dialog-item-click-event-handler
```

```
  :before ((v polygon-view) where)
  "Handles a mouse click at point where (in dialog coordinates)
by creating a cross when shift is pressed and/or
opening a dialog (double-click)."
    (let ((where (tv:global-to-local v where)))
      (let ((polygon-editor (tv:own-dialog v)))
        ;; Every item in a dialog "knows" its dialog, i.e
        ;; polygon view-items have access to their editors.
        (if (tv:double-click-p)
          (dialog-interaction *connect-dialog* polygon-editor)
          (when (tv:shift-key-p)
            (let ((cross (make-cross where)))
              (push cross
                    (polygon-crosses polygon-editor))
              (tv:add-view-items v cross)))))))
```

Wird nach einem Doppelklick die Schaltfläche „Connect" des daraufhin gezeigten Dialoges angeklickt, so wird die Funktion `connect-crosses` evaluiert (siehe Abschnitt 3.2). Deren Definition sieht wie folgt aus.

```
(defun connect-crosses (polygon-editor remove-points-p)
  "Connect the current crosses of a polygon editor to
an open polygon. The sequence of clicks determines the
form of the new polygon."
  (let* ((polygon-view (editor-view polygon-editor))
         (crosses (polygon-crosses polygon-editor))
         (polygon (make-polygon (mapcar #'cross-point
                                        crosses))))
    ...
    (tv:add-view-items polygon-view polygon)
     (when remove-points-p
       (apply #'tv:remove-view-items polygon-view crosses)
      (setf (polygon-crosses polygon-editor) nil))))
```

Die Plazierung und Zeichnung der Kreuze bzw. Polygone, nachdem sie innerhalb der obigen Methode zu dem Polygoneditor-Sichtbereich mittels der Funktion `tv:add-view-items` hinzugefügt wurde, wird durch den Sichtbereich selbst initiiert. Für Kreuze bzw. Polygone wurde oben eine spezielle Methode definiert.

An diesem Beispiel wird deutlich, daß Sichtbereichselemente ohne Aufwand auch wieder aus ihrem Sichtbereich entfernt werden können (`tv:remove-view-items`). Der Verwaltungsaufwand für die Sichtbereiche (Koordinatenberechnung, Rollen, Wann muß was neugezeichnet werden?) wird von dem erstellten System übernommen und braucht nicht für jede Visualisierungsanwendung neu erstellt zu werden.

3.5.2. Deutung der Sichtbereichselemente als Sub-Subfenster

Eine Aufgabe von Sichtbereichen ist die Bereitstellung eines Unterkoordinatensystems in einem Dialogfenster. Ein Sichtbereich kann also – wie oben schon angedeutet – in hierarchischen Fenstersystemen als (Sub-)Fenster implementiert werden. Koordinatentransformationen werden durch das zugrundeliegende Fenstersystem übernommen. Eine weitere Aufgabe der Sichtbereiche ist die Verwaltung des Zeichnens der zugeordneten Sichtbereichselemente (Rollen, Neuzeichnen nach Änderung der Fensterüberlappungsstruktur). Sichtbereichselemente sind als eigenständige Objekte realisiert. In hierarchischen Fenstersystemen könnten nun auch die Sichtbereichselemente als Sub-Sub-Fenster gedeutet werden. Im X-Windows-System z.B. ist ein Fenster nicht zwangsläufig mit einer Hintergrundfarbe gefüllt, sondern kann auch „durchsichtig" sein. Der Zeichenfunktion für Sichtbereichselemente in dem hier erstellten System für den Macintosh entspricht dann eine Funktion, die jedem Fenster zugeordnet ist und die das Neuzeichnen des Fensterinhaltes koordiniert. Die in dieser Arbeit eingeführten Sichtbereichselemente sind jedoch flexibler was die Angabe einer Löschfunktion betrifft. Bei einer Realisierung durch Fenster wäre nur eine durch das Fenstersystem vordefinierte Löschfunktion verfügbar. Es wäre jedoch eine konsistente Sicht, alle „Rechteckbereiche" mit Hilfe von Fenstern zu implementierten. In einigen Fenstersystemen (basierend auf Postscript) sind sogar beliebige Fensterformen möglich. Hier läßt sich ein Ansatzpunkt zur Verallgemeinerung von Fenstersystemen erkennen.

3.5.3. Verschiebbarkeit, Gruppierung und Markierung

Die Polygonstützpunkte sind ohne Programmieraufwand mit der Maus verschiebbar. Für jeden Sichtbereich kann festgelegt werden, ob ein automatisches Rollen durchgeführt werden soll, wenn durch Mausinteraktion Sichtbereichselemente aus dem aktuell sichtbaren Teil ihres Sichtbereichs herausbewegt werden. Verschiebbarkeit von Sichtbereichselementen wird einfach durch Hinzufügen der vordefinierten Klasse `tv:moveable-view-item-mixin`, welche die gesamte Verwaltung der Verschiebung übernimmt, erreicht. Sichtbereichselemente, die diese Klasse nicht in ihrer Vererbungsliste führen, sind nicht verschiebbar.

```
(defclass example-view-item
  (tv:moveable-view-item-mixin  tv:view-item)
  ()
  ...)
```

Während einer Verschiebung wird – wie von zahllosen Mal- und Zeichenprogrammen bekannt – ein Rahmen um das zu verschiebende Objekt gezeichnet.

Dieser Rahmen umschließt die Rechteckgrundfläche eines Sichtbereichselements. Es kann z.B. eine an die Kontur des Sichtbereichselements angepaßte Verschiebungsanzeige durch Definition einer Methoden für die generische Zeichenfunktion einer Verschiebungsanzeige definiert werden. In einigen Anwendungsfällen zieht die Verschiebung eines Sichtbereichsobjekts die synchrone Verschiebung einer Menge anderer Sichtbereichsobjekte nach sich. In Zeichensystemen ist dieses als Gruppierung bekannt. Eine Menge von Sichtbereichsobjekten kann auf einfache Weise gruppiert werden.

```
(tv:as-group view-item-1 view-item-2 ... view-item-n)
```

Gruppierte Sichtbereichselemente werden gemeinsam verschoben. Doch auch diese Möglichkeit bietet i.a. noch keine ausreichende Funktionalität: wenn z.B. Knoten eines Graphen interaktiv verschoben werden sollen, so sind die korrespondierenden Kanten nicht einfach um den gleichen Betrag mitzuverschieben. Die Kante muß in diesem Fall nur denselben relativen Ansatzpunkt am Knotensichtbereichselement beibehalten. Im weiteren Verlauf dieser Arbeit werden deklarative Beschreibungsformen zur Beschreibung solcher Abhängigkeiten geschildert.

Eine weitere häufig benötigte Interaktionsform ist die Markierung von Objekten innerhalb eines Sichtbereichs. Menüinteraktionen beziehen sich auf jeweils markierte Objekte. Sichtbereichselemente sind markierbar, wenn sie die Klasse `tv:markable-view-item-mixin` „hinzugemischt" bekommen. Das Standardverhalten (Invertierung des umschließenden Rechtecks) kann geeignet spezialisiert werden. Gruppierte Elemente werden ohne Programmieraufwand gemeinsam markiert.

3.6. Layoutalgorithmen

Layoutangaben ermöglichen eine flexible, deklarative Repräsentation von Anordnungsbeschreibungen. Dieser Abschnitt schildert die verwendeten Algorithmen und führt weitergehende Anordnungskonzepte ein.

3.6.1. Box-orientierte Layoutalgorithmen

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[1] genauer betrachtet. Der Inspektordialog wurde mit dem in dieser Arbeit vorgestellten Systemen erstellt.[2] In Tabellenform werden alle für eine Klasse relevanten Informationen wie Subklassen, Superklassen, Einträge (slots) und für die Klasse definierte Methoden dargestellt. Die Tabellen können nach Bedarf automatisch gerollt werden, sofern der für eine Tabelle zur Verfügung stehende Platz nicht ausreicht, alle Tabelleneinträge gleichzeitig anzuzeigen[3]. Abbildung 3.6 vermittelt einen Eindruck über die Standardaufteilung der Darstellungs- und Schaltflächen eines Klasseninspektordialogs.

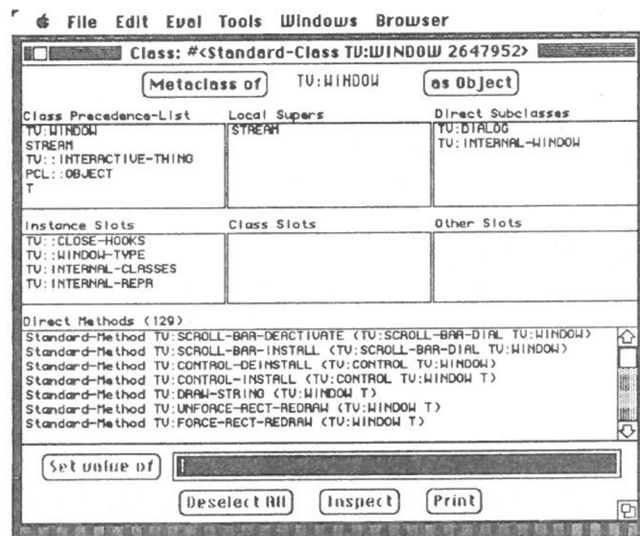


Abbildung 3.6: CLOS-Klasseninspektor-Dialog.

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

---

1 Neben Dialogen zur Inspektion von Klassen existiert auch ein Dialog für Objekte mit anderer Anordnung (siehe auch Schaltfläche `as Object` in Abbildung 3.6). Da Klassen auch als Objekte interpretiert werden, können Klassen auch in der allgemeineren „Objektdarstellung" untersucht werden. Der Übersichlichkeit halber sei an dieser Stelle nur der Klasseninspektordialog betrachtet.

2 Der Autor des Inspektor-Dialogs ist V. Haarslev.

3 Die Verwaltung der Tabellen und Schaltflächen wurde, wie im Abschnitt 3.1 erläutert, von der Macintosh-Toolbox bzw. von Allegro übernommen und in das erstellte System integriert.

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 Füllerausdehnung 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 Höhe erreicht werden. Abbildung 3.7 zeigt die Anordnungsstruktur nach einer (interaktiven) Vergrößerung des Dialogfensters.

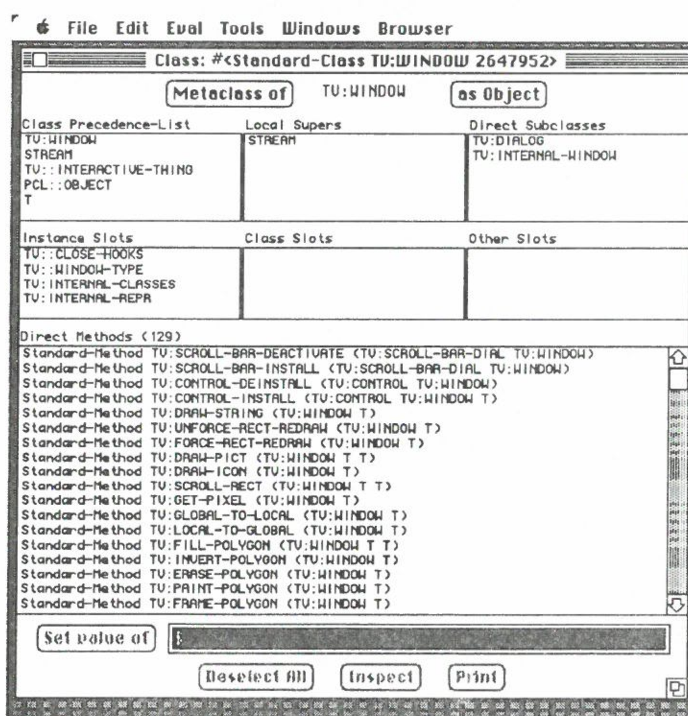


Abbildung 3.7: CLOS-Klasseninspektor-Dialog. Die Tabelle für die Methoden der gezeigten Klasse wird bei vergrößerter Darstellungsfläche auch entsprechend vergrößert. Die oberen Tabellen benötigen nicht mehr Raum.

Eine Von-Hand-Programmierung der durch Anordnungsbeschreibungen ermöglichten Darstellungen erhöht den Programmerstellungsaufwand erheblich, so daß Dialoge meist statisch angeordnet sind. Die Abbildung 3.8 zeigt ein Beispiel mit statischer Anordnung wie sie der Allegro-Inspektor verwendet. Selbst bei einer in ausreichendem Maße zur Verfügung stehenden Bildschirmfläche kann die Darstellung nicht vergrößert werden[1]. Abbildung 2.2 zeigt einen statischen Dialog einer kommerziellen Expertensystem-Entwicklungsumgebung.

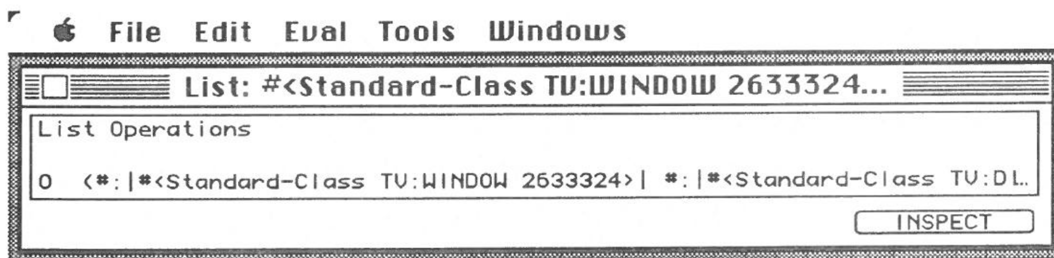


Abbildung 3.8: Inspektor-Dialog für eine Liste von Objekten.

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 3.9 verdeutlicht den Zusammenhang.

---

[1] In horizontaler Richtung werden fehlende Teile durch Punkte angedeutet. In vertikaler Richtung kann bei Bedarf ein Rollen erfolgen. Die vollständige Darstellung kann durch Markierung des Tabelleneintrags und Betätigung der Schaltfläche "Inspect" erfolgen. Der Nachteil ist, daß evt. eine Flut von Fenstern geöffnet werden muß, wenn mehrere Einträge geöffnet werden müssen.

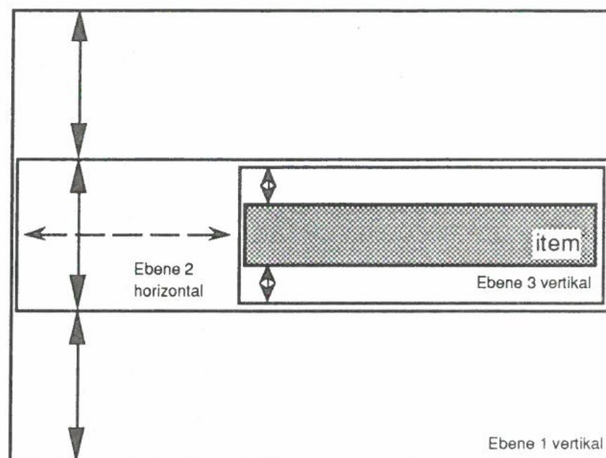


Abbildung 3.9: Schematische Darstellung von geschachtelten Boxen. Pfeile stellen Füllangaben dar. Füllangaben der Ebene 1 sind unabhängig von denen der Ebene 3.

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 die 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 3.10.

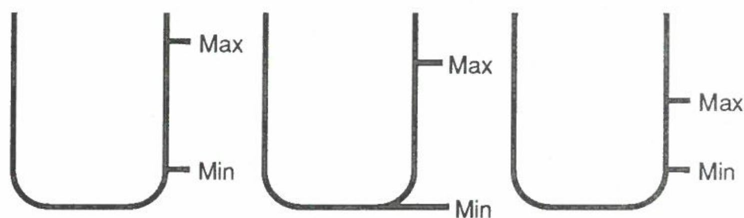


Abbildung 3.10: Wasserkrugmodell für Füllangaben.

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“[1]). 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 Z 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[2].

Das Anordnungsprinzip durch Boxmuster wurde erweitert, auch Anordnungen von Elementen innerhalb eines Sichtbereichs - also nicht nur innerhalb eines Dialogs - zu erlauben, um eine konsistente, integrierte Verwendungsweise der Werkzeuge zu erreichen. Denkt man aber z.B. an eine Baum- oder Graphanordnung von Sichtbereichselementen, die eine Vererbungshierarchie visualisieren, so wird deutlich,

---

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

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

daß eine Boxbeschreibung noch nicht ausreichend ist. Es wird eine Schnittstelle zur angepaßten Definition von Interpretern und Parsern für Anordnungsschemata, die in beliebigen Kontexten weiterverwendet werden können, benötigt. Da die Beschreibungen nicht als Programmtext i.e.S. gedeutet werden und die Anordnung schon während des Parsens erfolgt, werden die Bezeichnungen Interpreter und Parser hier synonym verwendet. Die nächsten Abschnitte schildern das Protokoll zur Erweiterung des Layoutmusterparsers.

3.6.2. Generelle Layoutangaben

Neudefinierte Parser für spezielle Layoutangaben sollten in die bestehenden Formalismen für horizonale und vertikale Boxen integriert werden können, um eine konsistente Verwendung zu gewährleisten. Hierzu dient ein spezielles Layoutmuster der Grundform `(:gbox ...)`. Anstelle der Punkte steht dann das selbstdefinierte Layoutmuster. Dieses hat die Form `(<pattern-name> ...)`. Das `:gbox` Muster stellt Datenstrukturen bereit, die die Verwaltungsinformation für die Schnittstellte zu den selbstdefinierten Layoutangaben aufnehmen. Die speziellen Angaben sind anwendungsabhängig. Ein Parser für das Layoutmuster kann durch Spezialisierung von generischen Parserfunktionen integriert werden. Ein Beispiel verdeutlicht diesen Sachverhalt.

Eine häufig einsetzbare Anordnungsstruktur ist ein gerichteter azyklischer Graph (DAG). Mit diesem Anordnungsinstrument können beliebige Knoten angeordnet und durch Kanten verbunden werden. Der Graph wird durch Angabe einer Nachfolgerfunktion für jeden Knoten definiert. Durch Angabe einer generischen Nachfolgerfunktion werden beliebig geschichtete Graphen möglich.

Die Syntax eines DAG-Anordnungsmusters lautet:

```
<dag-pattern> ::= ( :dag  <list-of-roots>
                          <successor-function>
                          <object-node-item-map-function>
                          <edge-definition-function> ) .
```

Die Bedeutung und die erwarteten Typen der verschiedenen „Parameter“ seien wie folgt (informell):

<list-of-roots>

Hier wird eine Liste aller Wurzeln des DAGs erwartet. Die Wurzeln werden durch Evaluierung der Nachfolgerfunktion (s.u.) expandiert. Gemeinsame Teilbäume verschiedener Wurzelobjekte werden vereinigt.

<successor-function> *object*

Durch die an dieser Stelle anzugebende Funktion soll die Menge der – aus der Sicht einer Anwendung geeignet definierten – Nachfolger geliefert werden. Die Verwendung einer generischen Funktion ermöglicht eine elegante typabhängige Nachfolgergenerierung.

<object-node-item-map-function> *object*

Die hier anzugebende Funktion wird für jeden generierten Knoten evaluiert. Der Wert dieser Funktion muß ein Sichtbereichselement sein, d.h. eine Instanz der Klasse `tv:view-item` oder einer Subklasse hiervon.

<edge-definition-function>

Generierungsfunktion eines als Kante verwendeten Sichtbereichsobjekts.

Folgende Funktionen integrieren den Parser für eine DAG-Anordnung[1]:

```
(defmethod layout:layout-spec-p-using-key ((key (eql ':dag)))
   t)

(defmethod layout:parse-layout-spec-using-key
                                                ((key (eql ':dag)))
                                                 layout-pattern)
   (parse-dag-layout-pattern ...))
```

Für jedes Muster der Form (`<pattern-name>` ...) wird innerhalb einer :gbox mithilfe der Funktion `layout:layout-spec-p-using-key` (Parameter ist jeweils `<pattern-name>`) festgestellt, ob es sich um ein bekanntes Layoutmuster handelt. Ist dieses der Fall, so wird die generische Parserfunktion `layout:parse-layout-spec-using-key` für das zu parsende Anordnungsmuster evaluiert. Der Parser bzw. Interpreter setzt als Seiteneffekt die Positionen der (erzeugten) Sichtbereichselemente relativ zu einem Nullniveau (Referenzkoordinate (0, 0)). Da die `:gbox` in einer horizontalen oder vertikalen Box auftreten kann, werden alle zunächst relativ zu (0, 0) angeordneten Sichtbereichselemente durch den Parser für horizontale und vertikale Boxmuster auf ein gewünschtes Niveau verschoben. Im Unterschied zu einer nach allen

[1] Die Bezeichner ...`using-key` erinnern an das CLOS Metaobjekt-Protokoll. Hier lautet die für verschiedene Metaklassen definierbare Eintragszugriffsfunktion (Slot accessor) `slot-value-`*`using-class`*. Anstelle des Schlüsselwortes `key` tritt dort die Metaklasse der Instanz auf, auf deren Eintrag zugegriffen wird. Die Metaklasse bestimmt die Implementation von Instanzen einer Klasse und damit auch die Zugriffe auf die Einträge einer Instanz.

Seiten beschränkten horizontalen oder vertikalen Box ist eine generelle Box nach rechts unten nicht beschränkt. Die Größe einer `:gbox` wird a posteriori nach Anordnung der Sichtbereichselemente in der `:gbox` ermittelt[1]. Die Grammatik zur Beschreibung der Syntax von Standardlayoutmustern wird geringfügig erweitert:

```
<layout-pattern>        ::=  <fbox-pattern>
                             | <vbox-pattern>
                             | <hbox-pattern>
                             | <gbox-pattern> .

<gbox-pattern>          ::=  ( :gbox ... ) .
```

Layoutmuster können mit der nachfolgenden Funktion, die die Schnittstelle zu allen Layoutmusterinterpretern bildet, auch in beliebigen Kontexten verwendet werden.

`layout:layout-description` *pattern*

Beauftragt das System, *pattern* mit den bisher integrierten Layoutmusterinterpreter zu bearbeiten. Das Resultat ist die Menge der angeordneten Sichtbereichselemente, die während des Parsens von *pattern* erzeugt wurden.

[1] Dieses ist u.a. der Grund, warum überhaupt eine `:gbox` verwendet wird, und nicht das selbstdefinierte Muster direkt eingesetzt werden kann. Durch die `:gbox` wird die gesamte Verwaltung übernommen.

3.6.3. Referenzen

Im vorangehenden Kapitel wurde stillschweigend vorausgesetzt, daß die Ansatzpunkte für Kanten sich an den jeweiligen Seiten der Knoten befinden. Der Standardparser für DAG-Layoutmuster liegt in einer gegenüber der obigen Darstellung erweiterten Form vor. Die Ansatzpunkte der Kanten an den Knoten können deklarativ festgelegt werden. Dieses Konzept läßt sich verallgemeinern: jedem Sichtbereichselement kann eine Menge von Referenzen zugeordnet werden. Eine Referenz wird als deklarative Referenzbeschreibung angegeben; die Referenzpunktkoordinaten werden hieraus automatisch berechnet. Ein Sichtbereichselement zur Visualisierung einer Kante wird durch Referenzen zu zwei Knotensichtbereichselementen bestimmt. 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 Referenzpunktdefinition heißt Referenzbox (kurz: `:rbox`).

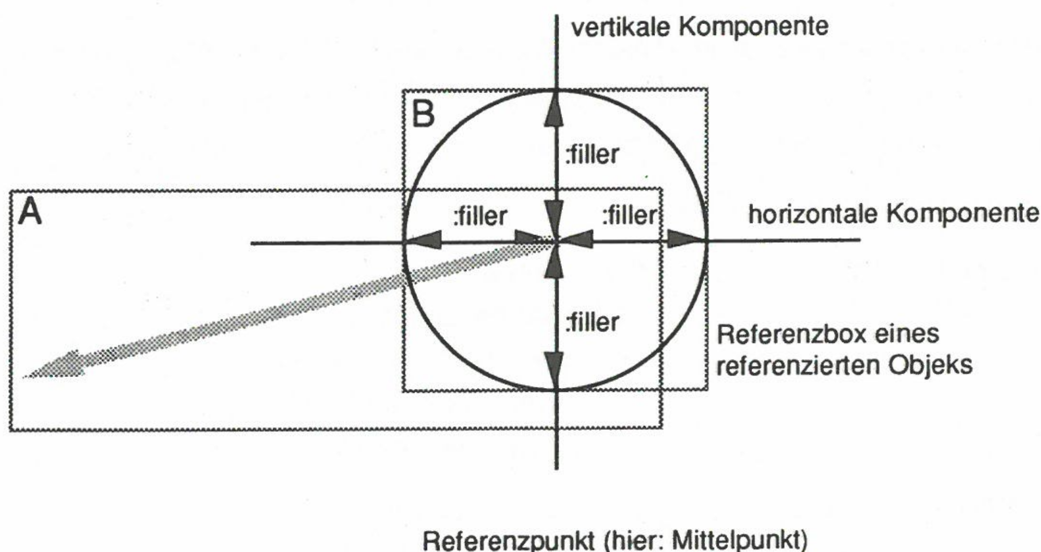


Abbildung 3.11: Referenzmodell. Ein Objekt A (ein Pfeil) ist durch einen Referenzpunkt bzgl. des Mittelpunkts eines Objekts B (ein Kreis) definiert. Die Zeichenrechtecke der beiden Figuren sind durch graue Rechtecke angedeutet.

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 Boxelemente wird hier das Schlüsselwort `:reference` verwendet. Der in der Abbildung 3.11 von einem Objekt A (ein Pfeil)[1] referenzierte Mittelpunkt eines Objektes 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 3.11 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 der Funktion `layout:layout-description` als Argument übergeben 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.

Die genaue Syntax eines Referenzboxmusters lautet wie folgt:

<rbox-pattern> ::= (**:rbox**
<referencing-object>
<referenced-object>
<horizontal-description>
<vertical-description>[2]).

<horizontal-description> ::= (**:horizontal**
<distance-description>
:reference
<distance-description>).

<vertical-description> ::= (**:vertical**
<distance-description>
:reference
<distance-description>).

---

[1] 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.

[2] Die horizontale und die vertikale Beschreibungskomponente können vertauscht werden.

<distance-description>	::=	{ { **:filler** } { <absolute-distance> } { <relative-distance> } }.
<absolute-distance>	::=	*positive integer* .
<relative-distance>	::=	*rational* ∈ [0, 1] | *float* ∈ [0, 1][1] .
<referencing-object>	::=	*view-item.*
<referenced-object>	::=	*view-item.*

Für jedes Sichtbereichselement sind Position und Größe festzulegen. Position und Größe definieren ein umschließendes Rechteck (Zeichenrechteck). Nur innerhalb dieses Rechtecks kann das Objekt gezeichnet werden. Zeichenoperationen, die außerhalb liegen, werden geklippt (siehe Abschnitt 3.5.1). Doch wie wird nun das umschließende Rechteck eines Objektes A, das nur durch Referenzen definiert ist, bestimmt? Für alle definierten Referenzpunkte eines Objekts A existiert ein kleinstes Rechteck mit horizontalen bzw. vertikalen Seiten, das alle Referenzpunkte umschließt (siehe Abbildung 3.12). Bezüglich dieses Rechtecks kann für ein Objekt A (also für ein referenzierendes Objekt) durch Angabe von vier Distanzangaben (linker, oberer, rechter und unterer Abstand, siehe Abbildung 3.12) ein äußeres Rechteck definiert werden. Durch dieses äußere Rechteck sind Positon und Größe des Objekts A und damit dessen Zeichenrechteck (implizit) festgelegt.

Ein Sichtbereichselement, das Referenzpunkte zu anderen Sichtbereichselementen führt, sollte nicht durch andere Layoutbestimmungsalgorithmen angeordnet werden, denn die Referenzen haben nicht die Mächtigkeit von allgemeinen Beschränkungen (constraints). Die Beeinflussungsrichtung ist unidirektional, und zwar von den referenzierten Objekten zu den referenzierenden Objekten.

---

[1] Relative Angaben in einer Referenzbox beziehen sich auf das umfassende Rechteck des referenzierten Sichtbereichselements. Eine Angabe von 1/4 bedeutet also eine Ausdehnung von n/4 Pixeln in vertikaler bzw. horizontaler Richtung, wenn n die Höhe bzw. Breite der Referenzbox ist.

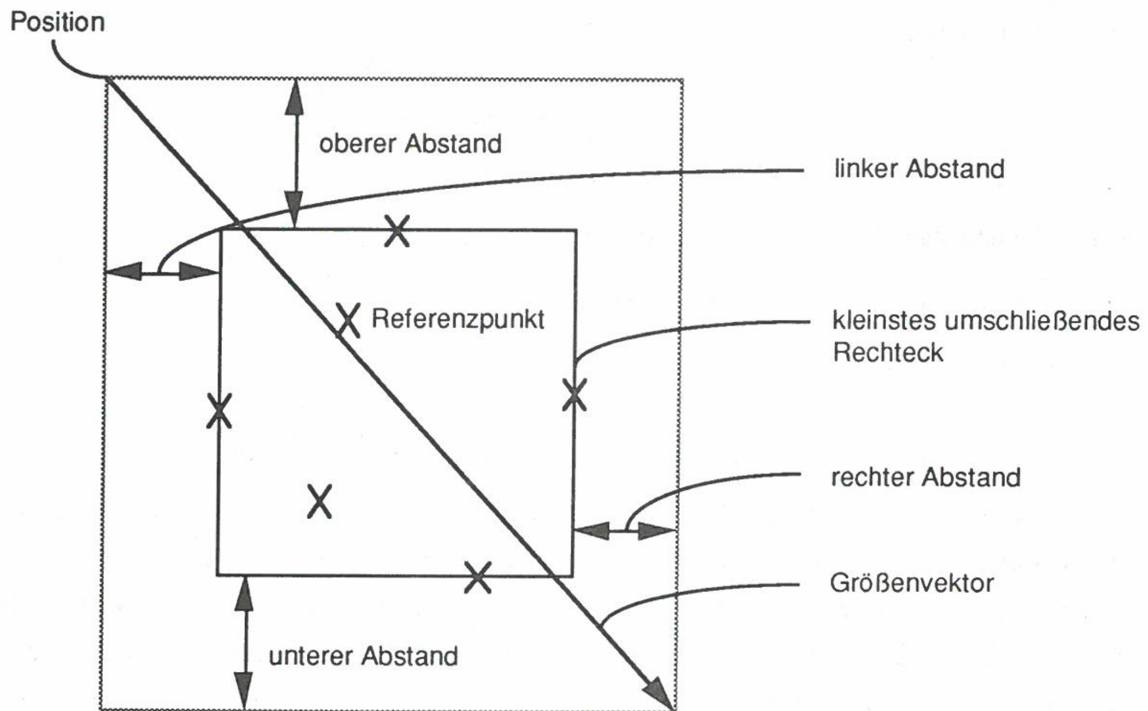


Abbildung 3.12: Bestimmung des Zeichenrechtecks aus einer Menge von Referenzpunkten und vier Abstandsangaben zu den Seiten des kleinsten umschließenden Rechtecks der Referenzpunkte.

Referenzangaben können wie andere Layoutmuster verwendet werden. Um nun ein Objekt innerhalb einer `:gbox` relativ zu einem anderen anzuordnen, könnte folgendes definiert werden:

```
(defmacro :relative (&rest forms)
   `(list ':relative ,@forms))
(defmethod layout:layout-spec-p-using-key ((key (eql ':relative )))
   t)

(defmethod layout:parse-layout-spec-using-key
            ((key (eql ':relative ))
             layout-pattern)
  (let ((referenced-object (second layout-pattern))
        (difference-vector (third layout-pattern))
        (referencing-object (fourth layout-pattern)))
   (setf (tv:view-item-right-offset referencing-object)
         (tv:point-h (tv:view-item-size referencing-object)))
   (setf (tv:view-item-bottom-offset referencing-object)
         (tv:point-v (tv:view-item-size referencing-object)))
   (layout:layout-description
                 (:rbox referencing-object
                        referenced-object
                         (:horizontal (tv:point-h difference-vector)
                                      :reference
                                     :filler)
```

```
                    (:vertical (tv:point-v difference-vector)
                               :reference
                               :filler)))
   (list referenced-object referencing-object))
```

Der Verschiebungsvektor `difference-vector` hat in diesem Beispiel als Ansatz- bzw. Endpunkt jeweils die linke obere Ecke der betreffenden Sichtbereichelemente.

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

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

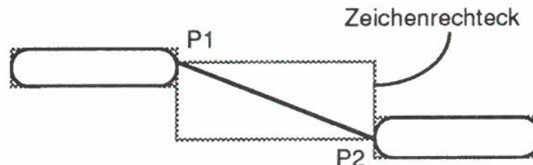


Abbildung 3.13: Kante zwischen zwei Knoten.

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

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

```
(setf (tv:pen-pattern dialog) previous-pen-pattern))))
```

Werden die gerundeten Rechtecke verschoben (`setf tv:view-item-position`), so werden die Referenzpunkte P1 und P2 der Linie neu berechnet und die Linie wird neu gezeichnet. Wird die Linie explizit verschoben, so werden die Referenzen entsprechend um den gleichen Betrag verschoben. Die Ansatzpunkte der Linie werden um den Verschiebungsvektor versetzt.

3.6.4. Beispiel: Visualisierung einer Vererbungshierarchie

Sichtbereiche können ohne Aufwand mit anderen Dialogelementen wie etwa Schaltflächen oder Tabellen kombiniert werden. Ein Anwendungsbeispiel hierzu wäre ein Visualisierungssystem für eine Vererbungshierarchie in einem objektorientierten System. Es bietet sich an, hier das Objektsystem CLOS, das als Implementierungsgrundlage dient, selbst darzustellen. 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.

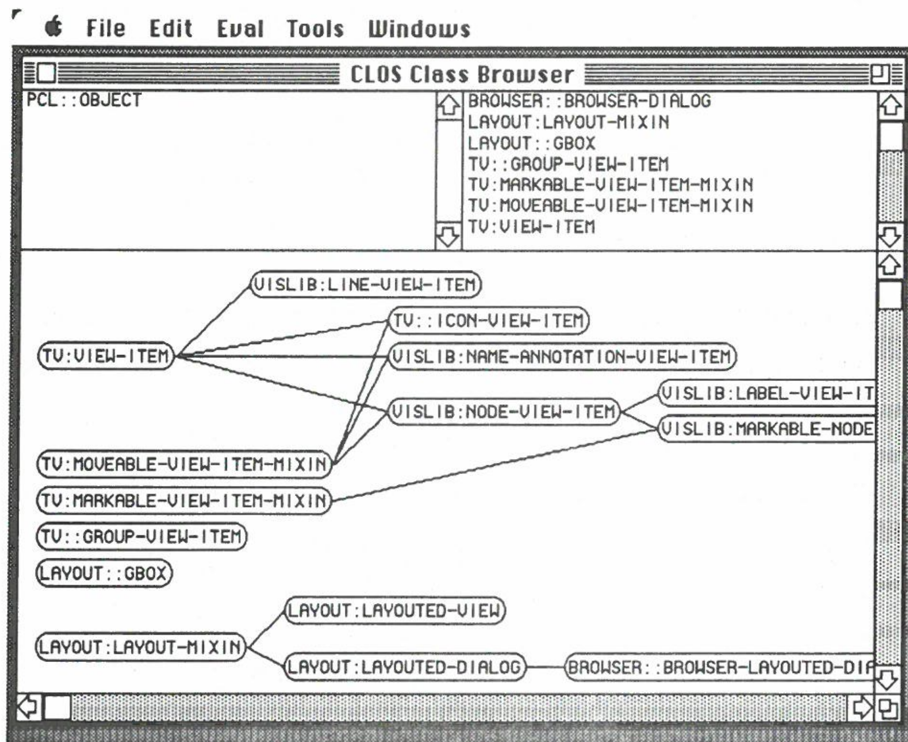


Abbildung 3.14: Visualisierung einer Vererbungshierarchie.

In einem Fenster werden, wie in Abbildung 3.14 dargestellt, im oberen Viertel zwei Tabellen für Klassennamen und darunter ein Sichtbereich für einen Graphen angeordnet.[1] Mit den bisher eingeführten Bauelementen läßt sich diese (strukturelle) Visualisierung auf recht einfache Weise erstellen. Es wird zunächst eine Klasse zur Beschreibung der Tabellen benötigt.

```
(defclass class-table
   (tv:sequence-dialog-item)
   ;; vordefiniert: Tabelle f. Sequenzen
  ((graph-view :accessor graph-view)          ; Sichtbereich für Graph
   (partner-table :accessor partner-table)    ; rechte bzw.
                                              ; linke Tabelle
   (double-click-shift-direction              ; Verschiebungsrichtung
          :accessor double-click-shift-direction ; bei Doppelklick
          :initarg :double-click-shift-direction))
  (:documentation "Tabelle zur Darstellung von CLOS-Klassen."))

(defun make-class-table (initial-class-sequence shift-direction)
  "Erzeugung einer Klassentabelle mit best. Voreinstellungen."
  (tv:make-dialog-item
   :class 'class-table
   :table-sequence initial-class-sequence
                        ; Angabe einer darzustellenden
                        ; Sequenz
   :double-click-shift-direction shift-direction
   :table-hscrollp nil          ; kein horizontales Rollen
   :selection-type ':disjoint))   ; Mehrfachmarkierung von Einträgen
```

Die in einer Tabelle dargestellte Sequenz (hier eine Liste) besteht aus Klassenobjekten. Die Standardausgabefunktion für Klassenobjekte erzeugt einen zu umfangreichen Text (z.B. `#<Standard-Class TV:WINDOW 2633324>`). Für die Ausgabe in einer Tabelle reicht der Name (Symbol). Für den Inhalt jeder Zelle evaluiert das System noch eine Filterfunktion `tv:cell-contents`, die den auszugebenden Zellinhalt festlegt. Durch Definition einer speziellen Methode wird der Filter auf Instanzen von `class-`

[1] Die Visualisierung hat folgende Funktionalität: Wird eine Klasse in der linken Tabelle markiert, so werden in der rechen Tabelle ihre Subklassen angezeigt. Wird umgekehrt in der rechten Tabelle eine Klasse markiert, so werden in der linken Tabelle ihre Superklassen dargestellt. Durch Doppelklick auf eine Klasse links, wird die Tabelle nach rechts verschoben und links die Superklassen der doppelt angeklickten Klasse angezeigt. In der rechten Tabelle bewirkt ein Doppelklick auf eine Klasse eine Verschiebung der Tabelle nach links und eine Darstellung der Subklassen der angeklickten Klasse in der rechten Tabelle. Erfolgt ein Klick bei gedrückter Wahltaste, so werden die Subklassen der ausgewählten Klasse(n) (einschließlich dieser) als Graph im unteren Sichtbereich dargestellt. Durch Betätigung der Umschalttaste können – wie bei Macintosh-Programmen üblich – auch mehrere Tabelleneinträge ausgewählt und im Sichtbereich dargestellt werden.

`table` abgestimmt. Filter oder Konvertierer sind elementare Werkzeuge, auf die im Kapitel 5 noch eingegangen wird.

```
(defmethod tv:cell-contents ((item class-table)
                             h &optional (v nil))
  "Filterfuntion zur Bestimmung des Aussehens eines Tabelleneintrags.
In der Tabelle sind Klassenobjekte enthalten. Es sollen
jedoch nur deren Namen erscheinen."
  (let ((contents (call-next-method)))
    (if (not (equal contents ""))
      (format nil "~S" (class-name contents))
      contents)))
```

Wird ein Dialogelement angeklickt, so evaluiert das Laufzeitsystem die generische Funktion `tv:dialog-item-action`. Für die Tabelle wird nun eine geeignete `:after` Methode definiert, mit der die Mausinteraktionen verwaltet werden.

```
(defmethod tv:dialog-item-action :after ((item class-table))
  "Behandlung der Mausinteraktion
Einfachklick = Anzeige der Super- bzw. Subklassen
               in der Partnertabelle
Klick mit Wahltaste = Anzeige der Subklassen in Graphform
Doppelklick = Anzeige der Sub- bzw. Superklassen mit
Verschiebung der Tabelle nach links bzw. rechts."
  ;; Siehe Anhang C
  ...
  (show-graph (graph-view item))
              ; Erzeugung des Graphen innerhalb des Sichtbereichs
  ...)
```

Es werden noch die Definitionen einiger Hilfsfunktionen benötigt, um die Tabellen zu verwalten. Tabellen sind aus Zellen, den Tabellenelementen, aufgebaut. Zellen können durch Mausklick markiert und entmarkiert werden. Weiterhin läßt sich die gezeigte Sequenz auf einfache Weise ändern. Das System führt eine Anpassung der Darstellung durch. Die Funktionen zur Tabellenverwaltung wurden von der Allegro-Umgebung übernommen, etwas verfeinert und in das erstellte System integriert.

Die Funktion `show-graph` übernimmt die Generierung des Graphen. Dieses erfolgt durch Festlegung der Anordnung von Sichtbereichselementen in einem DAG. Für das in obigen Abschnitten besprochene DAG-Layoutmuster liegt ein Parser mit etwas erweiterter Funktionalität vor.[1] Neben den bekannten Angaben wie Nachfolgerfunktion und Funktionen zur Erzeugung der Sichtbereichselemente, die Knoten und Kanten

---

[1] Der Algorithmus zur Anordnung von Knoten in Graphform stammt von M. Hußmann, Universität Hamburg. Die übernommenen Lisp-Funktionen und Datenstrukturen wurden an die Verwendung von Sichtbereichselementen und generellen Boxen angepaßt. Die Erzeugung von Knoten und Kanten wurde so umgestaltet, daß die Erzeugungsfunktionen als Funktionen an die Layoutalgorithmen übergeben werden können. Weiterhin wurden Expansionsbeschränkungen, Expansionsprädikate und Kantenreferenzpunkte integriert.

darstellen, kann noch eine maximale Expansionstiefe sowie ein Expansionsprädikat angegeben werden. Es wird versucht, durch Umordnungen von Geschwisterknoten Kantenkreuzungen zu vermeiden.

Ähnlich wie bei Dialogen kann mit den gleichen Werkzeugen ein Layout auch für Sichtbereiche definiert werden. Wird das Layout neu bestimmt, so werden alle im Sichtbereich befindlichen Elemente aus diesem entfernt. Anschließend wird das neue Layoutmuster geparst und die angeordneten Sichtbereichselemente werden zum Sichtbereich hinzugefügt. Es kann also jederzeit eine Layoutveränderung vorgenommen werden.

```
(defvar *hierarchy-depth* 9
  "Tiefe eines DAGs zur Visualisierung einer Vererbungshierarchie.")

(defun show-graph (view selected-classes)
  (setf (layout:layout view )
  ;; Festlegung des Layouts des unteren Sichtbereichs.
        (:vbox
        ()
        10        ; 10 Pixel oberer Abstand
         (:hbox
         ()
         10       ; 10 Pixel linker Abstand
          (:gbox
           (:dag selected-classes         ; Menge der Wurzeln des DAGs
                 #'pcl::class-direct-subclasses ; Nachfolgerfunktion
                 *hierarchy-depth*       ; max. Expansionstiefe 9 Knoten
                 #'(lambda (class) t)    ; Expansionsprädikat
                 #'(lambda (class)       ; Erzeugungsfunktion für
                     (vislib:make-label ; Knotensichtbereichselement
                      (class-name-as-string class)))
                 #'make-line-view-item
                 #'vislib:western-reference
                 ;; Festlegung des Anfangsreferenzpunkts einer
                 ;; Kanten durch eine Funktion, die ein
                 ;; Referenzboxmuster liefert (s.o.)
                 #'vislib:eastern-reference))))))
                 ;; dito für den Kantenendpunkt

(in-package 'vislib)

(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)))
```

Die Visualisierung der Vererbungshierarchie verwendet ein DAG-Layout innerhalb einer generellen Box. Für die Knoten und Kanten des Graphen können vordefinierte Bauteile verwendet werden. Knoten zeigen den Klassennamen in einen gerundeten Rechteck und werden durch die Funktion `vislib:make-label` erzeugt. Die Knoten sind durch Mausinteraktion verschiebbar. Kanten werden als Instanzen der Klasse `vislib:line-view-item` realisiert.[1]
Ein neuer Klassenvisualisierer kann mit einem Funktionsaufruf erzeugt werden.[2]

```
(defun class-browser (&optional (roots (list (find-class 't))))
  (let ((supers-table (make-class-table roots ':right))
        (subs-table (make-class-table '() ':left))
        (class-browser-view (layout:make-layouted-view
                              :scroll-bars ':both
                                          ; Rollen in beiden Richtungen
                              :bordered-p nil      ; keine Umrandung
                              :auto-scrolling t
                              ...)))
    ...
    (layout:make-layouted-dialog
     ...
     :layout (:vbox ()
                  (:hbox (:height 1/4)
                         (:vbox ()
                                (:hbox ()
                                       (:fbox () supers-table)
                                       1
                                       (:fbox () subs-table)))
                         1)
                  1
                  (:fbox () class-browser-view )))))
```

Unabhängig von der Dialoggröße wird die Anordnung der Dialogteile durch ein weiteres Layoutmuster festgelegt. Die Höhe der oberen Tabellen beträgt ein Viertel der Dialoghöhe; der Rest wird für den Graphen verwendet (vgl. Abbildungen 3.14 und 3.15).

---

[1] Kanten sind nicht maussensitiv, haben aber eine spezielle Methode zum Löschen (vgl. auch Abschnitt 3.6.3).

[2] Wie in Kapitel 3.2 gezeigt wurde, kann die Funktion `class-browser` ohne weiteren Aufwand auch von einem Menüelement aus evaluiert werden. Dieses wird hier vernachlässigt.

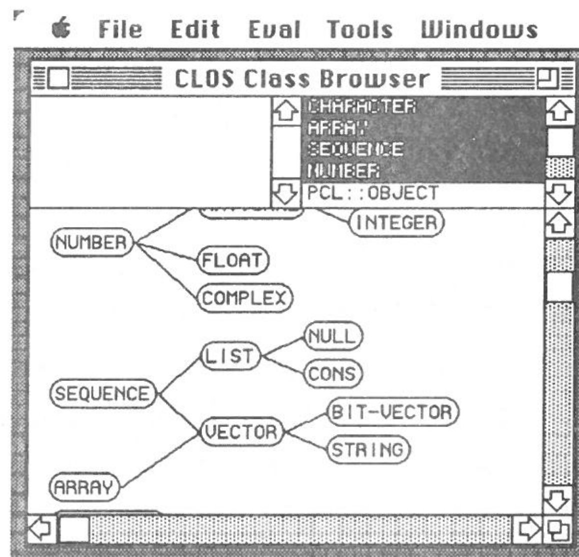


Abbildung 3.15: Ein Ausschnitt aus der CLOS-Klassenhierarchie.
Gemeinsame Teilbäume verschiedener Wurzeln des DAG werden vereingt.

Damit ist das Beispiel zur Visualisierung einer Vererbungshierarchie komplett. Das Beispiel zeigt, wie vorgefertigte Bauteile verwendet und erweitert werden können. Layoutbeschreibungen lassen sich in gleicher Weise für verschiedene Ebenen (Dialogelemente und Sichtbereichselemente, Referenzpunkte) verwenden.

3.6.5. Layoutangaben in anderen Systemen

Das Benutzerschnittstellenentwicklungssystem InterViews [Linton et al. 89] 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, sowie von numerischen Distanzangaben. Dem Ausdruck

```
(:hbox ()
      :filler
      push-button-1
      :filler)
```

des in dieser Arbeit vorgestellten Systems entspricht der C++-Ausdruck

```
new HBox(new HGlue(0, hfil),
         push-button-1,
         new HGlue(0, hfil))
```

des InterViews-Systems. Füller und Boxen werden hier als Objekte realisiert und mit new erzeugt. Eine Ausdehnung der horizontalen Box kann nicht festgelegt werden[1]. Ein entsprechendes Ausdrucksmittel zu der in dieser Arbeit vorgestellten Rahmenbox (:fbox), die die Größe eines anzuordnenden Elements angibt, existiert m.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 [Herczeg 86]. Es existieren hier vordefinierte Klassen zur Anordnung von Fenstern in Form einer Sequenz, eines Stapels, eines Kreises usf. Layoutbeschreibungen werden als eigenständige Objekte realisiert (siehe Abbildung 3.16).[2] Es können nachträglich Fenster zu einem „Cluster" hinzugefügt werden.[3]

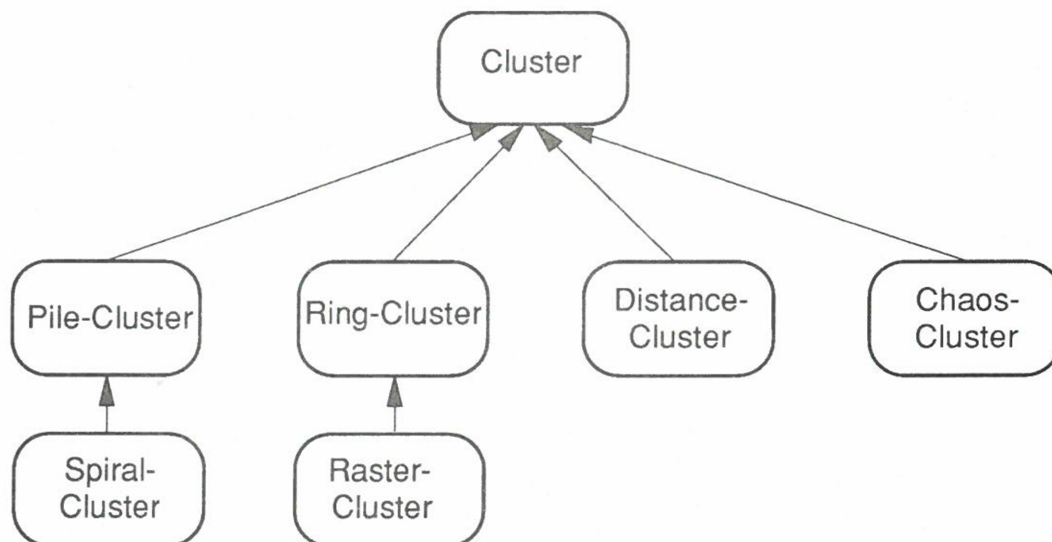


Abbildung 3.16: Typen von sog. „Clustern" zur Anordnung von Fenstern.
Die Abbildung wurde aus [Herczeg 86] übernommen.

---

1 In dem in dieser Arbeit vorgestellten System kann eine Boxausdehnung, die bei den Anordnungsalgorithmen entsprechend berücksichtigt wird, innerhalb des Klammerpaares nach dem Symbol :hbox erfolgen. Siehe Abschnitt 3.3.1.

2 Eine objektorientierte Strukturierung für (selbstdefinierte) Layouts wie sie in Abbildung 3.15 angedeutet wird, ist in dem in dieser Arbeit vorgestellten System nicht ausgeschlossen. Layoutbeschreibungsausdrücke lassen sich als Makros realisieren. Wie diese expandiert werden ist implementationsabhängig.

3 Chaos-Cluster werden verwendet, um Fenster zu gruppieren. Operationen können damit auf mehrere bereits (zufällig) plazierte Fenster angewendet werden.

Füllangaben und Rahmenboxen sind in diesem System nicht vorgesehen. Dieses System ermöglicht m.W. keine Integration der Formalismen zur Anordnungsbeschreibung von Objekten innerhalb eines Fensters.

Die Symbolics-Programmierumgebung Genera[1] bietet ebenfalls Mechanismen zur Bestimmung von Layouts für Fensteranordnungen. 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`, absolute Angaben, Verwendung von `:as-needed`). In Genera können mehrere Layoutkonfigurationen definiert werden. Ein Beispiel verdeutlich die Definition von Layoutangaben (entnommen aus [Symbolics 88b]).

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

Die Angabe `:even` entspricht einem Füller, d.h. beiden Subfenstern `pane-1` und `pane-2` wird jeweils gleichviel Platz zugeteilt. In der Konfiguration `config1` erfolgt eine vertikale Anordnung, in `config2` wird horizontal angeordnet. 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 beziehen sich

---

1 Symbolics und Genera sind eingetragene Warenzeichen der Firma Symbolics, Inc., Cambridge, Massachussets.

auf Anordnungen von Fenstern in anderen Fenstern. Graphische Objekte, wie sie durch Sichtbereichselemente unterstützt werden, können nicht angeordnet werden (vgl. aber Abschnitt 3.5.2). Referenzen (lokale Anordnungen) werden nicht unterstützt.

Spezielle Systeme ermöglichen die Erzeugung und Editierung von (gerichteten) Graphen. In dem in C++ implementierten System EDGE [Newbery 88] kann für die Knoten und Kanten in ähnlicher Weise eine spezielle Zeichenmethode definiert werden. Das System realisiert aber keine allgemeinen Sichtbereichselemente, die für verschiedene Anwendungen auch außerhalb eines Grapheditors verwendbar sind. Das EDGE-System ermöglicht eine Editierung des Graphen, d.h. es können Knoten und Kanten hinzugefügt bzw. gelöscht werden. Ein weiteres System zur Visualisierung von gerichteten Graphen wird in [Rowe 86] vorgestellt. Diese speziell auf Graphendarstellung ausgerichteten Systeme optimieren die Darstellung stark in Bezug auf Vermeidung von Kantenüberkreuzungen. Weiterhin schneidet eine Kante niemals einen Knoten. Ggf. werden virtuelle Knoten eingeführt, an denen die Kanten scheinbar „Knickstellen" aufweisen.

Eine weitere Möglichkeit zur Festlegung von (relativen) Anordnungen ist die Verwendung von Einschränkungen (constraints). [Vander Zanden 89] schlägt vor, Techniken zur Erfüllung von Einschränkungen (constraint satisfaction) zur Berechnung von Attributen in attributierten Grammatiken zu verwenden. Eine Grammatik bildet eine hierarchische (Teil-Ganzes-)Struktur. Bei geeignet gewählten Attributen (Abstände von und zu Sohnknoten) kann ein Baumlayout bestimmt werden. [Bothner 88] schlägt vor, aus Einzelteilen zusammengesetzte Figuren mittels Einschränkungen zu definieren. So kann beispielsweise festgelegt werden, daß entsprechende Seitenhöhen von Teilfiguren identisch sind oder in einem bestimmten Verhältnis zueinander stehen. Eckpunkte können als gemeinsam deklariert werden etc. Ähnliche Konzepte sind auch im Juno-System realisiert [Nelson 85]. Die Aufrechterhaltung von Einschränkungen während einer Interaktion (z.B. Verschiebung von Eckpunkten bei festgelegten Fixpunkten einer zusammenhängenden Figur) sind in dem in Smalltalk integrierten Thinglab-System möglich [Borning 81]. Die zur Erzeugung der Graphik notwendigen Methoden werden vor der „Bewegung" der Eckpunkte kompiliert, um die nötige Geschwindigkeit zu gewährleisten. Es erfolgt hier eine Vorausberechnung (Kompilierung) und Spezialisierung der Einschränkungsbeschreibung für den aktuellen Anwendungsfall. Es ist also ein beträchlicher Aufwand nötig, um eine an eine interaktive Anwendung angepaßte Einschränkungserfüllung zu gewährleisten. Für viele Anwendungen sollten die in dieser Arbeit vorgestellten Referenzen zwischen Sichtbereichselementen ausreichen, obwohl sie nicht so ausdrucksstark sind wie allgemeine Einschränkungsbeschreibungen.

Die Smalltalk-Programmierumgebung ermöglicht eine Anordnung von Subfenstern in ihrem Vaterfenster durch relative Angabe von prozentualen Anteilen (normierte Koordinaten aus [0.0, 1.0]) gegenüber der insgesamt zur Verfügung stehenden Fläche.

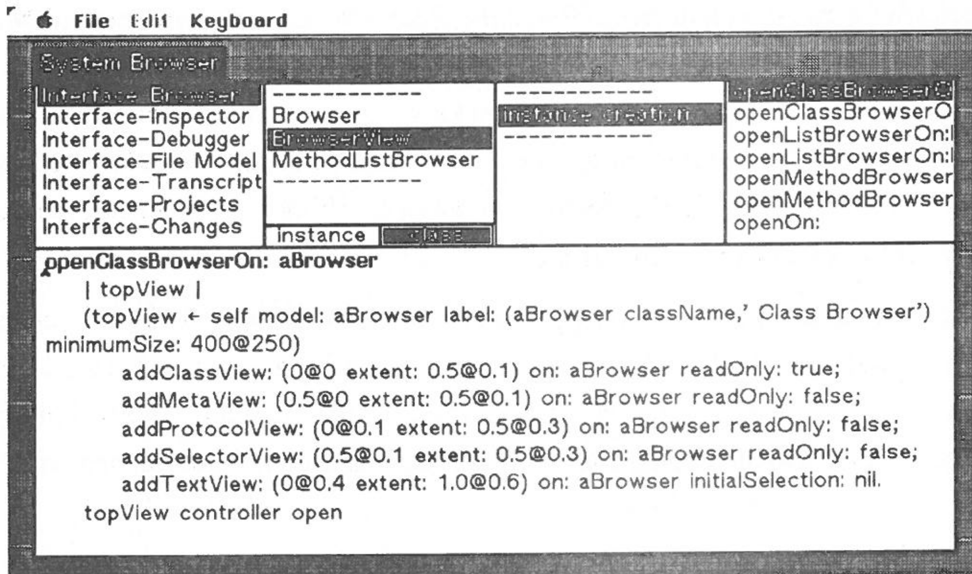


Abbildung 3.17: Systeminspektor der Programmierumgebung Smalltalk-80.
Gezeigt wird der Quelltext einer Methode zur Erzeugung eines Klasseninspektors.

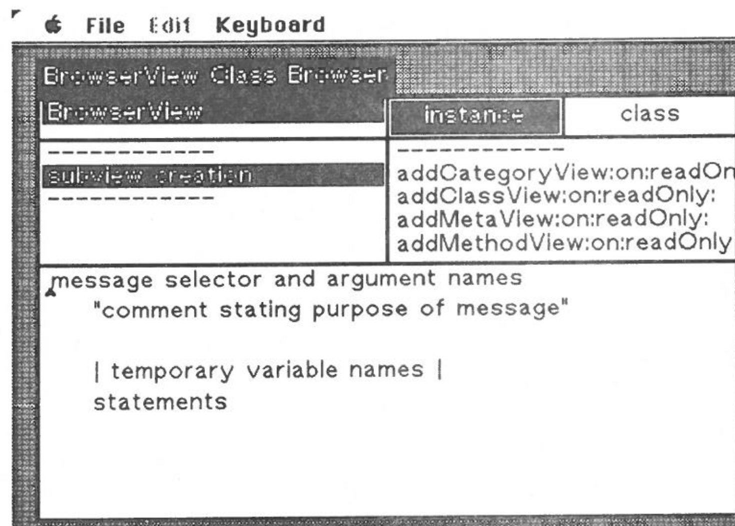


Abbildung 3.18: Klasseninspektor für die Klasse `BrowserView`.
Die Aufteilung der Teilflächen wurde in der in Abbildung 3.16 gezeigten Methode definiert.

Abbildung 3.17 deutet das Konzept an. Ein Fenster wird in verschiedene Unterfenster aufgeteilt. Hierzu erzeugt das System Rechtecke, die die Lage und relative Größe des betreffenden Subfensters beschreiben. Ein Punkt wird erzeugt, indem einer Zahl die Nachricht `@` mit einer weiteren Zahl als Parameter gesendet wird. Ein Rechteck wiederum wird erzeugt, indem einem Punkt die Nachricht `extent:` mit einem Punkt als Parameter zur Beschreibung der Größe eines Rechtecks gesendet wird. Auch der linke obere Eckpunkt eines Beschreibungsrechtecks wird dabei relativ zum umgebenden Fenster angegeben. Das Fenster, an das die Nachrichten versendet werden, berechnet aus diesen relativen Angaben die absoluten Koordinaten. Abbildung 3.18 zeigt das Layout eines Klasseninspektors an einem Beispiel.

In der Methode `openClassBrowserOn:` aus Abbildung 3.17 tritt schon eine spezielle Smalltalk-Terminologie (Nachrichten `model:`, `controller` und `addClassView:`) auf, die im nächsten Kapitel im Hinblick auf eine Kopplung von Anwendungs- und Visualisierungskomponenten noch detaillierter besprochen wird.

4. Teilung der Zuständigkeiten

Von vielen Autoren wird eine Trennung von Anwendungs- und Visualisierungskomponente gefordert. Das liegt u.a. darin begründet, daß die Visualisierungskomponente i.a. nicht unabhängig vom verwendeten Rechnersystem gestaltet werden kann. Doch auch zur Realisierung von programmtechnischen Prinzipien wurden verschiedene Formalismen entwickelt, die eine Isolierung von Anwendungs- und Visualisierungskomponenten ermöglichen sollen. In den nächsten Abschnitten werden in diesem Zusammenhang relevante Aspekte von einigen Visualisierungssystemen und Programmierumgebungen bzw. deren Teilsysteme zur Benutzerschnittstellen-programmierung vorgestellt. Außerdem wird betrachtet, wie bei den einzelnen Modellen „interessante Ereignisse“ definiert werden können.

Apsekte der in Kapitel 3 vorgestellten Klassen und Funktionen etc. zur Programmierung von Grundbausteinen lassen sich ebenfalls in einigen Systemen wiederfinden. Folgende Aufstellung gibt zunächst einen Überblick:

System	*betrachteter Aspekt*	*Kurzbeschreibung*
Smalltalk-80	MVC-Schema	objektorientierte Programmierumgebung
Loops (85)	Active-Values	auf Interlisp basierende Programmierumgebung, später auf Common Lisp umgestellt (Common Loops)
Balsa-II (87)	Algorithm-Events	System zur Ezeugung von Algorithmenanimationen (in PASCAL)
CLUE (89)	Callbacks	auf dem X-Windows-Quasi-Standard aufsetzendes System zur Organisation der Benutzerschnittstellenprogrammierung unter Verwendung von CLX und CLOS
Incense (83)	Artists	Inspektorsystem

4.1. Smalltalk's „Model-View-Controller“ Schema

In Smalltalk-80 [Goldberg & Robson 83] wurde eine Dreiteilung der Zuständigkeiten vorgenommen: das „Model-View-Controller“ Schema. Eine

Kurzzusammenfassung ist in [London & Duisberg 84] zu finden. Genauer und anhand von Beispielen wird dieses Schema in [Hoffmann 87] und [Krasner & Pope] dargestellt.

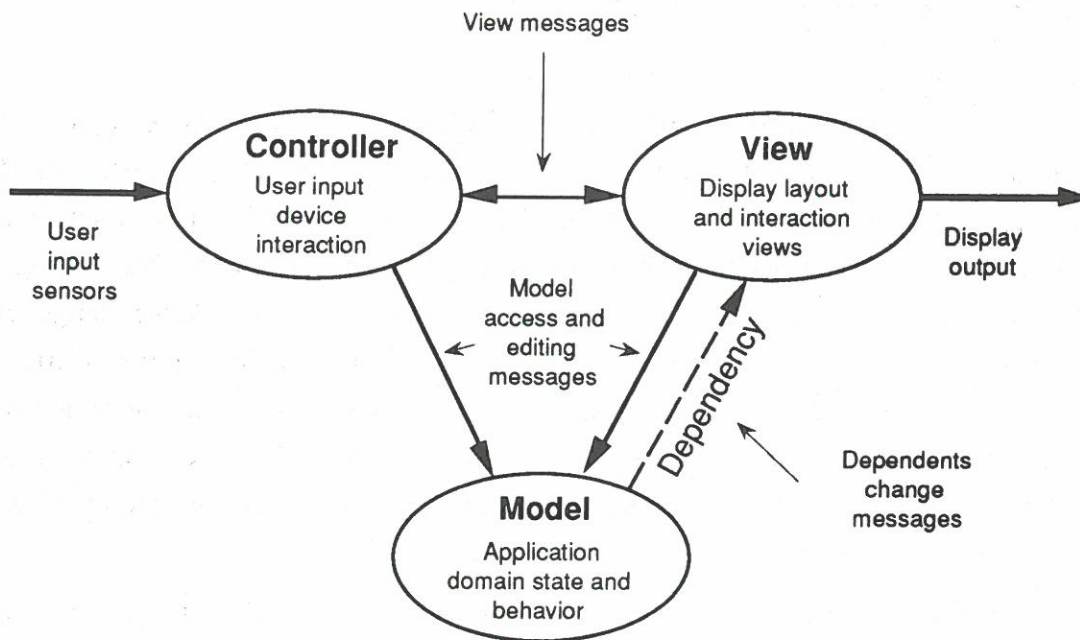


Abbildung 4.1: MVC-Kommunikationsschema (aus [Krasner & Pope]).

In einem Modell werden Leistungen einer Anwendung zusammengefaßt (Abbildung 4.1). Für eine Menge von Smalltalkobjekten der Anwendung wird dabei eine Klasse als Schnittstelle zur Visualisierungskomponente definiert. Diese Klasse legt das Modell der Anwendung fest. Eine Sicht dieses Modells ist eine Instanz der Klasse `View` (oder einer Subklasse hiervon). Es kann verschiedene solcher Sichtobjekte geben. Ein Sichtobjekt übernimmt die Verwaltung der Sichtbarkeit der Objektdarstellung (Fensteraufteilung, Rollen, Koordinatentransformationen). Eine zugeordnete Instanz der Klasse „Controller“ übernimmt die Verwaltung der Mausinteraktionen und die Einbettung in die Fensterumgebung (z.B. empfängt nur das „aktive“ Fenster Eingaben).

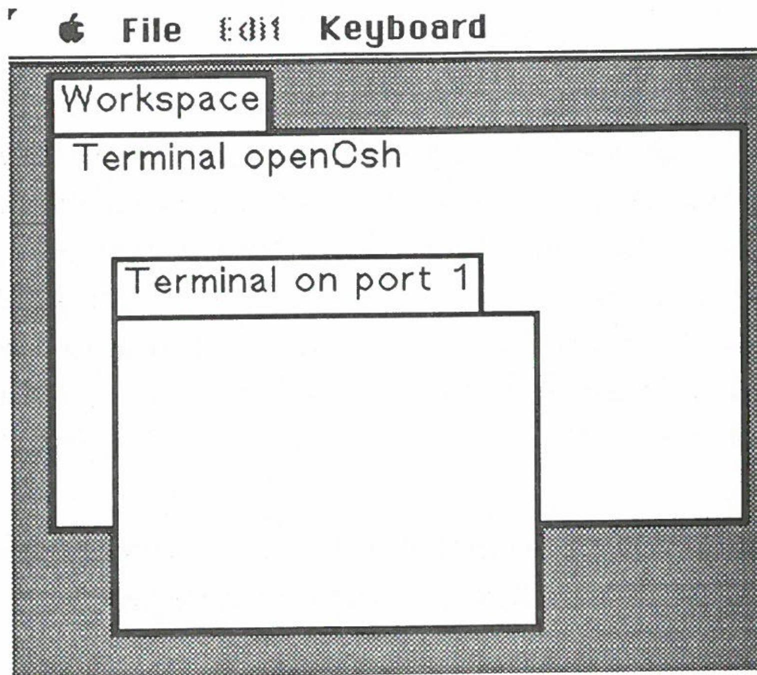


Abbildung 4.2: Ein Terminalfenster in der Smalltalk-80 Programmierumgebung.

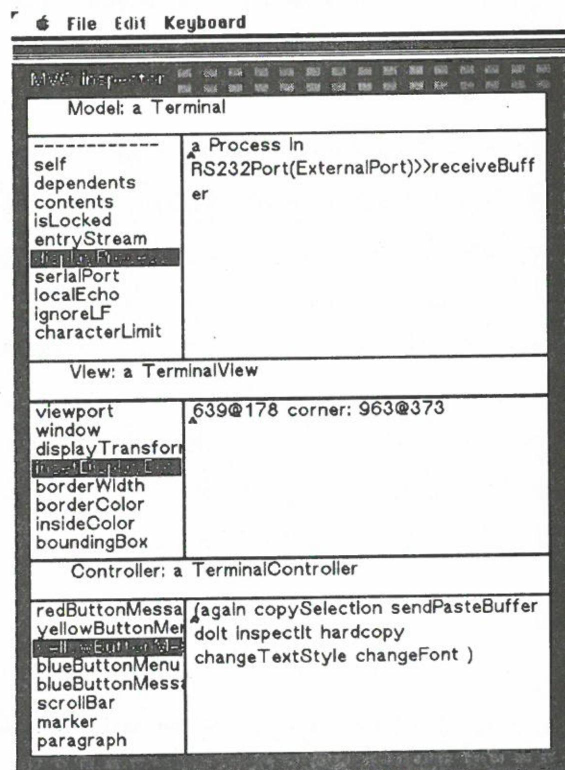


Abbildung 4.3: Spezieller Inspektor für alle Komponenten des MVC-Schemas. Dargestellt wird die Struktur des Terminalfensters aus Abbildung 4.1.

In den Abbildungen 4.2 und 4.3 wird die Architektur an einem einfachen Beispiel angedeutet. Im „Workspace"-Fenster wird ein Terminalobjekt erzeut (Nachricht `openCsh`). Als Seiteneffekt erscheint ein Terminalfenster. Dieses Terminalfensterobjekt ist die Sicht (Instanz der Klasse `View`) auf das Objekt `a Terminal`, das durch die Nachricht `openCsh` an die Klasse `Terminal` erzeugt wurde. Die Abbildung 4.3 zeigt die gesamte MVC-Struktur mit den jeweiligen Instanzvariablen in der linken Spalte. Dem Objekt `a Terminal`[1] wurde ein Sichtobjekt `a TerminalView` (Subklasse von `View`) und ein Kontrollobjekt `a TerminalController` (Subklasse von `Controller`) zugeordnet.[2]

Einem Sichtobjekt wird durch die Nachricht `model:` ein Modellobjekt zugeordnet. Diese Nachricht wird in einer geeigneten Sichtobjektmethode gesendet.

```
terminalView model: aTerminal.
```

Im Smalltalksystem ererbt jedes Objekt von der Klasse `Object` Methoden zur Verwaltung von Abhängigkeiten (siehe [Goldberg & Robson 83], Seiten 239-245). Die Abhängigkeiten werden dabei eher als implizit verstanden[3]. Durch die Nachricht `model:` wird bei dem an `a Terminal` gebundenen Modellobjekt vermerkt, daß das Objekt `a TerminalView` von diesem abhängig ist. Diese Abhängigkeit wird vom Sichtobjekt aus und nicht vom Modellobjekt aus initiiert!

Nun muß allerdings auf der Modellseite vermerkt werden, daß bei einer bestimmten Zustandsänderung die abhängigen Sichtobjekte darüber benachrichtigt werden müssen. Dieses sollte jedoch ohne explizite Kenntnis der einzelnen abhängigen Sichtobjekte – es können mehrere sein – erfolgen. Dazu kann ein Modellobjekt durch Rundsenden (broadcasting) Änderungsnachrichten an seine ihm zugeordneten Sichten (views) übermitteln. Hierzu existieren verschiedene Methoden für Nachrichten wie `broadcast:` oder `changed:`. Diese Nachrichten werden von der „Modellseite" aus an die vom Modellobjekt (implizit) abhängigen Objekte versendet. Auf der „Sichtseite" werden dann entsprechende Methoden (re-)implementiert.

---

1 Die Bezeichnung `a Terminal` deutet darauf hin, daß es sich um irgendeine nicht näher gekennzeichnete Instanz der Klasse `Terminal` handelt.
2 Auf eine Beschreibung der Klasse `Controller` wird hier verzichtet.
3 Abhängigkeiten sind in einer Klassenvariablen der Klasse `Object` vermerkt (in einem sog. `Dictionary`). Aus Gründen der Speichereffizienz kann nicht für jedes Objekt - nicht jedes Objekt wird als Modell verwendet - ein direkter Verweis auf die abhängigen Objekte in einer Instanzvariablen geführt werden.

London und Duisberg fassen z.B. ein „interessantes Ereignis" selbst als eigenständiges Objekt auf. Ein Interessantes-Ereignis-Objekt wird von dem Modellobjekt rundgesendet (nach [London Duisberg 84], Seite 65)[1]:

```
self broadcast: #update:
     with: (InterestingEvent of: #operation
                             with: value
                             on: actor).
```

Das von der Klasse `InterestingEvent` erzeugte Objekt stellt die Beschreibung des „interessanten Ereignisses" dar. Der Zustand eines Objekts – und damit auch dessen Visualisierung – wird durch die Werte seiner Instanzvariablen bestimmt. Ändert sich der Wert einer Instanzvariablen, so müssen die abhängigen Objekte darüber informiert werden:

```
self changed: #update with: (x <- newValue).
```

Die Aufgaben sind also in diesem System korrekt verteilt. Der „semantische" Teil der Abhängigkeiten wird auf der Seite des Modells festgelegt, d.h. es wird definiert, daß ein „interessantes Ereignis" stattgefunden hat und wie dieses ggf. benannt ist. Auf der Seite des Modells wird hingegen nicht festgelegt, wie auf solche Ereignisse reagiert wird und insbesondere wer auf diese Ereignisse reagieren soll. Dieses erfolgt auf der Seite der Sichten. Von dieser Seite aus werden – wie schon erwähnt – auch die Abhängigkeiten initiiert. Weiterhin werden Methoden (re-)implementiert, die festlegen, wie auf Aktualisierungshinweise, die von Seiten des Modellobjekts zunächst nur allgemein rundgesendet werden, reagiert wird.

Dennoch ist die Programmierung eines solchen Systems recht komplex. Dieses gilt besonders, da auf Seiten der Sichtobjekte auch noch die Kommunikation mit den Kontrollobjekten (controller) organisiert werden muß. Außerdem muß in den Methoden des Modellobjekts explizit vermerkt werden, daß potientielle Sichtobjekte eines Modellobjekts von „interessanten Ereignissen" in Kenntnis zu setzen sind.

4.2. Active Values in Loops

Im vorigen Abschnitt wurde geschildert, daß bei Änderung des Wertes beispielsweise einer Instanzvariablen andere Objekte darüber in Kenntnis gesetzt werden müssen. In einer anderen Sichtweise kann dieses auch als aktives Verhalten der

[1] Die Quellcodefragmente verwenden Smalltalk-Syntax. Bei mit # gekennzeichneten Bezeichnern handelt es sich um sog. Literale (Synonym: Symbole). An `self` ist innerhalb einer Smalltalk-Methode der Empfänger der Nachricht gebunden.

Instanzvariable gedeutet werden. In Anlehnung an die Terminologie des Loops-Systems [Bobrow & Stefik 83] können sie als „Active Values" bezeichnet werden.

In Loops können nicht nur Schreibzugriffe, sondern auch Lesezugriffe auf aktive Variablen „überwacht" werden, d.h. es können allgemeine Funktionen angegeben werden, die bei einem Schreib- bzw. Lesezugriff evaluiert werden. Active Values wie in Loops sind u.a. als Grundbausteine zur Realisierung von rahmenbasierten (frame-based) Wissensrepräsentationsformalismen wie z.B. FRL [Roberts und Goldstein 77] geeignet. Hier finden sich ähnliche Mechanismen zur Anbindung von Prozeduren an Einträge (slots) von Instanzen (Synonym: Instanzvariablen) durch sog. „Dämonen" (Procedural Attachment: `if-needed`, `if-removed`, `if-added`).

Mit dieser Sichtweise ergeben sich Ansatzpunkte für eine Art von untergeschobener Überwachung. Durch Funktionen auf Seiten der Sichtobjekte können an Instanzvariablen der Anwendungsobjekte geeignete Benachrichtigungsfunktionen (Dämonen) angeheftet werden.

4.3. Algorithmenereignisse in BALSA-II

Ein weiteres Modell zur Kopplung von Anwendungs- und Visualisierungskomponenten stellt Brown [Brown 88, Brown & Sedgewick 85] mit seinem BALSA-II-System zur Algorithmenanimation vor. Der Autor geht davon aus, daß man gezwungen ist, einen in einer (von ihm betrachteten) Programmiersprache wie etwa PASCAL codierten Algorithmus mit sog. Anmerkungen (annotations) zu versehen. Diese Anmerkungen (Einschübe in den Quelltext) definieren dann die „interessanten Ereignisse"[1]. Er bezeichnet die „interessanten Ereignisse" als „Programmanalogon zum Oszilloskop" (Übersetzung: R.M.). Auch ein Oszilloskop, so wie es hier verstanden wird, hat keine grundsätzliche Funktion innerhalb einer Schaltung, sondern wird von einem Techniker z.B. zur Prüfung einer Schaltung eingefügt.[2] Wie oben beschrieben, werden durch diese Anmerkungen „interessante Ereignisse" festgelegt und Aktualisierungsaufträge angestoßen.

In diesem System sind die Anwendungs- und die Visualisierungskomponente nicht getrennt. Der Autor hält eine vollständige Trennung nicht für durchführbar. Bei einem

---

[1] Brown unterscheidet zwischen Ein- und Ausgabeereignissen. Eingabeereignisse sind im Zusammenhang mit der von ihm betrachteten Algorithmenanimation von Bedeutung (verschiedene Ströme unterschiedlicher Eingabedaten). Sie seien hier einmal außen vorgelassen.

[2] Genau wie ein Oszilloskop bei zu geringem Innenwiderstand ungewollt Seiteneffekte innerhalb einer Schaltung provozieren kann, ist auch die Verwendung von Anmerkungen mit einer gewissen Fehleranfälligkeit versehen.

Sortieralgorithmus für ein Feld (Array) sei z.B. ein „interessantes Ereignis" die Vertauschung zweier Elemente. Ein Sortierprogramm könnte etwa folgende PASCAL-Zeilen enthalten:

```
...
repeat
  repeat i := i + 1 until a[i] >= v;
  repeat j := j - 1 until a[j] <= v;
  t := a[i]; a[i] := a[j]; a[j] := t;
until j <= i;
a[j] := a[i]; a[i] := a[r]; a[r] := t;
...
```

Es wird deutlich, daß die Vertauschungsoperation nicht durch eine prozedurale Abstraktion wiedergegeben ist. Es ist also hier nicht möglich, das „interessante Ereignis" *Vertauschung* etwa durch Anbindung der oben erwähnten „Anmerkungen" an eine Prozedur zu definieren (man denke losgelöst von der eingeschränkten PASCAL-Welt etwa an `:before` bzw. `:after` Methoden). Aber auch eine Prozeduralisierung des Algorithmus' brächte keine Lösung:

```
...
repeat
  repeat i := i + 1 until Compare(a[i], v, '>='1);
  repeat j := j - 1 until Compare(a[j] v, '<=');
  Exchange(a[i], a[j]);
until j <= i;
Exchange(a[i], a[j]);
Exchange(a[i], a[r]);
...
```

Das Element `a[i]` befindet sich am Ende dieses Code-Fragments an seinem endgültigem Platz[2]. Dieses läßt sich durchaus als ein „interessantes Ereignis" auffassen. Wenn also die „interessanten Ereignisse" streng an Prozeduren gekoppelt werden, so

---

1 In PASCAL können auch Prozeduren als Parameter übergeben werden. Die Übergabe des Operatornamens (als Zeichenkette) ist also nicht der eleganteste Weg. Die Intention des Autors wird dennoch deutlich.

2 Dieses geht offensichtlich nicht aus dem aus dem Zusammenhang gerissenen Programmfragment hervor.

wäre man gezwungen, eine „leere“ Prozedur wie z.B. `ElementInPlace` an die Stelle hinter die beiden letzten `Exchange` Anweisungen einzufügen. Offensichtlich ist das Ereignis „Ein Objekt befindet sich an seinem endgültigem Platz“ für dieses Sortierprogramm nicht relevant. Nichtsdestotrotz wäre es aber u.U. wünschenswert, diese Tatsache irgendwie zu visualisieren.

Hierzu schlägt er vor, den Programmcode für einen Algorithmus in geeigneter Weise mit Code-Teilen zur Generierung eines Stromes von sog. Algorithmenereignissen zu versehen.

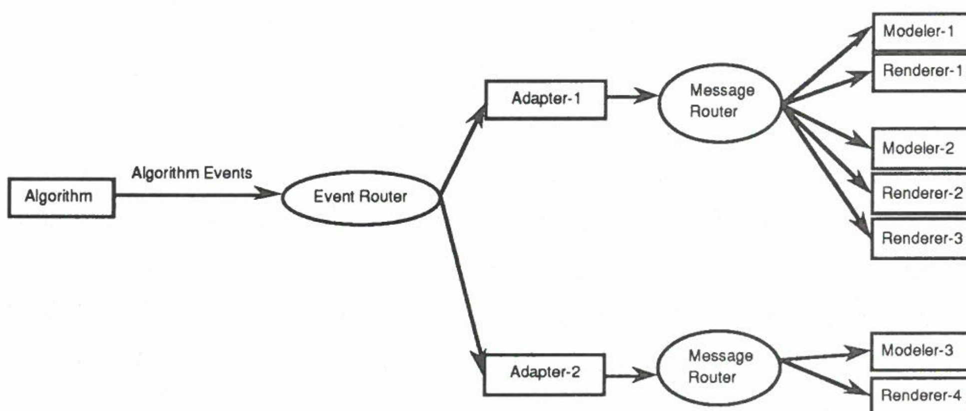


Abbildung 4.4: Verteilungschema für „Algorithmenereignisse“ (nach [Brown 88]).

Ein Algorithmenereignis wird über einen Verteiler an einen Adapter, der Umformungen und Filterungen vornehmen kann, weitergeleitet. Ein Adapter gehört zu einer Sicht (view) eines Algorithmus'. Die gesamte Sicht eines Algorithmus' besteht aus einem Adapter[1], einem Modellierer (modeler) und einer Darstellungskomponente (renderer). Ein Modellierer soll eine, für mehrere Visualisierungsanforderungen angepaßte, Schnittstelle zur Manipulation einer Visualisierungsdarstellung sein (z.B. Funktionen wie `update`, `create`, `dispose`, `start-run`, usw.). Ein Modellierer kann zur Visualisierung einer Klasse von Algorithmen verwendet werden. Für einen Strom von Algorithmenereignissen können gleichzeitig mehrere Adapter und dahinter mehrere Modellierer eingesetzt werden. Wie aus Computer-Graphik-Systemen bekannt, übernimmt eine Wiedergabekomponente (renderer) die Ausgabe. Wiederum kann ein Modell von mehreren Darstellungskomponenten verarbeitet werden. Durch einen Strom von Algorithmenereignissen läßt sich die Visualisierungskomponente von der

---

[1] Ein Adapter kann – wie aus der Abbildung 4.4 zu entnehmen – von mehreren Sichten geteilt werden.

Anwendungskomponte (Programm, das einen Algorithmus realisiert) zumindest konzeptionell trennen.

4.4. Contacts in CLUE

In dem auf eine Programmierung von Benutzerschnittstellen ausgelegten CLUE-System (Common Lisp User Interface Environment) wird ein weiteres Modell zur Kopplung von Anwendungs- und Darstellungsfunktionen realisiert [Kimbrough & LaMotte 89]. Das System baut auf der Common Lisp-X-Windows-Schnittstelle (CLX) auf [X-Windows].

Im Vergleich mit dem MVC-Paradigma liegt im CLUE-System eine Verschmelzung von `Controller` und `View` vor. Die Ankopplung der Anwendung (Modellkomponente in Smalltalk) geschieht unter Verwendung von sog. „Callbacks". Ein Darstellungsobjekt verfügt über einen Eintrag (slot) zur Verwaltung einer Assoziationsliste von „Callbacks". Ein Element dieser Liste besteht aus einem „Callback"-Namen und einer „Callback"-Funktion. Die Darstellungskompontente legt die „Callback"-Namen fest und definiert, wann und unter welchen Bedingungen die von der Anwendungskomponente festzulegenden „Callback"-Funktionen evaluiert werden. Z.B. könnte eine Menüdarstellung ein Callback-Protokoll mit einem „Callback" `:select` festlegen. Wird ein Menüelement selektiert, so wird die dem „Callback" von einer Anwendungskomponente zugewiesene Funktion evaluiert.

Betrachtet man dieses Schema genauer, so entdeckt man, daß es strukturgleich ist mit dem in dieser Arbeit verwendeten Prinzip der Kopplung durch generische Funktionen. Wird z.B. ein Menüelement ausgewählt, so evaluiert das Laufzeitsystem die generische Funktion `tv:menu-item-action`. Für spezielle Menüelemente wird eine angepaßte Methode definiert, die bei Auswahl des Menüauswahlelements evaluiert wird. In CLOS ist es möglich, Methoden auch für einzelne Objekte zu definierten. Diese Methoden entsprechen den „Callback"-Funktionen. „Callback"-Funktionen sind allerdings leichter austauschbar als Methoden. Durch „Callbacks" kann ein Kopplungschema ein wenig expliziter dargestellt werden, ansonsten erscheint mir der Ansatz nicht wesentlich von dem hier vertretenen verschieden zu sein. Die Evaluierung einer „Callback-Funktion" kann allerdings nicht z.B. durch `:around` Methoden in einen bestimmten Kontext gesetzt werden.

4.5. Artists in Incense

Die im vorigen Abschnitt angesprochenen Mechanismen bilden eine Grundlage für eine Abbildung einer Menge von Anwendungsobjekten in eine visuelle Repräsentation. Eine Abbildungsfunktion bezeichnet Myers in seinem System Incense zur Darstellung von Datenstrukturen mit einer etwas globaleren Sichtweise als „Artist“ [Myers 83][1]. Ein „Artist“ abstrahiert von den technischen Details einer Implementation wie z.B. Auslösung (Triggerung) durch Dämonen. Zur Erzeugung mehrerer Sichten können einer Menge von Anwendungsobjekten auch mehrere „Artist“-Objekte zugeordnet werden.

Nun sollte durch ein „Artist“-Objekt nicht direkt eine Zeichnung in einem Fenster generiert werden. Es ist vorteilhaft, zwischen „Artist“ und der konkreten Darstellung noch eine sog. abstrakte Darstellung einzuschieben (siehe Abbildung 4.5). Die abstrakte Darstellung entspricht dem Modell bzw. Modellierer des Balsa-Systems von Brown.

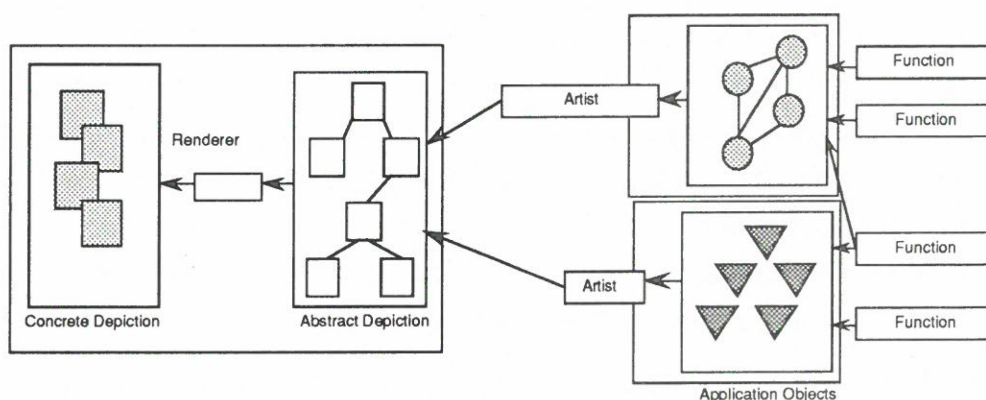


Abbildung 4.5: „Artist“-Organisationsschema [nach Young et al. 88].

Die abstrakte Darstellung (Abstract Depiction) kann insbesondere zur Organisation und Verwaltung von Benutzerinteraktionen verwendet werden (z.B. bei Verschiebung von Objekten innerhalb der konkreten Darstellung: die neue Position muß bei eventuellen Neuzeichnungen zugängig sein). Außerdem kann durch die abstrakte Darstellung nicht nur die Anwendungskomponente von der Visualisierungskomponente getrennt werden, sondern es kann auch die Visualisierungskomponente von der Anwendungskomponente entkoppelt werden. In der Abbildung 4.5 soll dieses durch die Komponente „Renderer“ symbolisiert werden. Das Kapitel 5 schildert, wie das abstrakte Modell aus Abbildung 4.5 in dieser Arbeit konkret realisiert wurde. In Abschnitt 5.6 erfolgt ein Vergleich mit den hier aufgeführten Modellen.

---

[1] Eine deutsche Übersetzung mit „Artist“ wäre mißverstandlich.

5. Ein Verarbeitungsmodell zur Erstellung von Visualisierungen

Dieses Kapitel bildet den Kern dieser Arbeit. An weiteren Beispielen wird beschrieben, wie Visualisierungen mit den in Kapitel 3 vorgestellten Grundelementen (Dialoge, Dialogelemente, Sichtbereiche, Sichtbereichselemente) aufgebaut werden können. Zusätzlich werden anhand dieser Beispiele die in dieser Arbeit verwendeten Techniken zur Kopplung von Anwendungs- und Visualisierungskomponenten erläutert. Es wird dabei untersucht, wo sich Abweichungen oder Parallelen gegenüber den im vorigen Kapitel besprochenen Modellen ergeben.

5.1. Indirekte Werte von Objekteinträgen

Die Terminologie von Visualisierungskomponenten enthält häufig Begriffe wie „Anzeigegerät" (gauge), „Skala" (dial), „Skalenwert" und „Zeiger" (pointer). Bei der Programmierung eines Anzeigegeräts (Synonym: Anzeige) sollten keine anwendungsspezifischen Informationen einfließen. Eine Basisklasse zur Implementation von Anzeigen, könnte wie folgt definiert werden.

```
(defclass gauge
  (tv:view-item)
  ((dial-value :initarg :dial-value
               :accessor dial-value))
  (:default-initargs :bordered-p t) ;; initarg of tv:view-item
  (:documentation "Gauge abstract class.
Subclasses must define special methods for drawing the dial and
the pointer, respectively."))

(defmethod tv:view-item-draw :after ((gauge gauge) view dialog)
   ;; Make a gauge visible. The generic function
   ;; tv:view-item-draw is evaluated by the gauge's view to
   ;; draw its contents.
   ;; The function is used as a dummy to change drawing
   ;; concerns to gauge terminolgy.
  (let ((position (tv:view-item-position gauge))
        (size (tv:view-item-size gauge)))
    (dial-draw gauge
               position size)
    (pointer-draw gauge
                  (dial-value gauge)
                  position size)))
```

Die Zeichenfunktionen der Bauteile der abstrakten Anzeigeklasse erzeugen einen Fehler (`subclass-responsibility` [vgl. Goldberg & Robson 83]), wenn eine Subklasse keine speziellen Methoden bereitstellt.

```
(defgeneric dial-draw (gauge
                        gauge-position gauge-size)
  (:documentation "Draws the dial of a gauge."))

(defmethod dial-draw ((gauge gauge)
                       position size)
  (subclass-responsibility 'draw-dial 'gauge))

(defgeneric pointer-draw (gauge current-pointer-value
                           gauge-position gauge-size)
  (:documentation "Draws the pointer of a gauge."))

(defmethod pointer-draw ((gauge gauge) (current-pointer-value t)
                          position size)
  (subclass-responsibility 'pointer-draw 'gauge))
```

Man beachte, daß die Zeichenfunktion für die Skala und den Zeiger[1] auch bezüglich der Aktualwerte spezialisiert werden können. In Bezug auf Anzeigen sind die Klassen- und Funktionsnamen der Sichtbereichselemente, die die Anzeige implementieren, relevant, nicht jedoch die Namen der Anwendungskomponente, die durch die Anzeige visualisiert werden soll. Eine Unterklasse von `gauge` könnte wie folgt definiert werden. Die Anzeigeinstanzen können drei diskrete, qualitative Werte oder numerische Werte darstellen (Abbildung 5.1).

```
(defclass overview-gauge
  (gauge)
  ()
  (:default-initargs :dial-value ':middle)
  (:documentation "Overview gauge used to show discrete
or numerical values."))

(defmethod dial-draw ((gauge overview-gauge)
                       gauge-position gauge-size)
   (tv:frame-arc ...))

(defmethod pointer-draw ((gauge overview-gauge)
                          (current-pointer-value number)
                          gauge-position gauge-size)
   (tv:fill-arc ...))

(defmethod pointer-draw ((gauge overview-gauge)
                          (current-pointer-value (eql ':low))
                          gauge-position gauge-size)
   "Draw 0 degrees arc."
   (tv:fill-arc ...))
```

[1] Der Begriff „Zeiger“ wird im weiteren Sinne als die Kenngröße der Anzeige interpretiert. Die Säule eines Fieberthermometers wäre in diesem Falle der „Zeiger“ des Thermometers.

```
(defmethod pointer-draw ((gauge overview-gauge)
                         (current-pointer-value (eql ':middle))
                         gauge-position gauge-size)
   "Draw 90 degrees arc."
   (tv:fill-arc ...))

(defmethod pointer-draw ((gauge overview-gauge)
                         (current-pointer-value (eql ':high))
                         gauge-position gauge-size)
   "Draw 180 degrees arc."
   (tv:fill-arc ...))
```

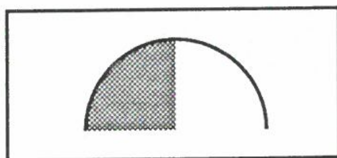


Abbildung 5.1: Skizze einer Anzeige der Klasse `overview-gauge` (siehe Text).

Mit dieser Anzeige könnte nun beispielsweise ein Objekt visualisiert werden, das innerhalb eines Expertensystems eine qualitative Einschätzung modelliert.

```
(defclass estimation
   (...)
   ((estimation-value :initarg :init-estimation
                      :accessor current-estimation)))
```

Beide Komponenten, Anzeigen und Schätzungen, verwenden eine verschiedene Terminologie und sind gemäß den Überlegungen aus Kapitel 4 programmtechnisch zu trennen.

In Smalltalk-80 wird im MVC-Schema ein Schätzer als Modell aufgefaßt. Dieses Modell wird einer als Sicht (view) realisierten Anzeige zugeordnet. Ein Kontrollobjekt (controller) übernimmt die Kontrolle der Interaktion. Wird der angezeigte Wert interaktiv geändert, so erfolgt eine Benachrichtigung des Modells durch ein festzulegendes Nachrichtenprotokoll. Das Modell muß seinerseits Aktualisierungsnachrichten an seine Sichten weiterleiten. Wie im vorigen Kapitel angedeutet, führt ein Modell keinen expliziten Verweis auf seine zugeordneten Views. Mit dem MVC-Schema wurde ein eigenes Instrumentarium geschaffen, die Benutzerschnittstellenprogrammierung zu organisieren. Reichen für einfache Visualisierungen nicht auch einfachere Mittel?

Betrachtet man sich das obige Beispiel genauer, so erkennt man, daß der Eintrag `dial-value` (aktueller Skalenwert) einer Anzeige zu jedem Zeitpunkt genau den Wert des Eintrags `estimation-value` des zu visualisierenden Schätzerobjekts enthalten muß. Dieses Verhalten kann erreicht werden, indem die Eintragslesefunktion (slot reader function) `dial-value` den Eintrag `estimation-value` des Schätzerobjekts

referenziert. `dial-value` soll in diesem Fall also als indirekter Eintrag (indirect slot) fungieren. In dem in dieser Arbeit erstellten System können solche indirekten Slotwerte durch Verwendung der neuen Metaklasse `pcl:indirect-slots-class` realisiert werden. Die Definition der Anzeigen wird dahingehend abgeändert.

```
(defclass gauge
  (tv:view-item)
  ((dial-value :initarg :dial-value
              :accessor dial-value))
  (:metaclass  pcl:indirect-slots-class)
  (:default-initargs :bordered-p t) ;; initarg of tv:view-item
  (:documentation "Gauge abstract class.
Subclasses must define special methods for drawing the dial and
the pointer."))

(defclass overview-gauge
  (gauge)
  ()
  (:metaclass  pcl:indirect-slots-class)
  (:default-initargs :dial-value ':middle)
  (:documentation "Overview gauge used to show discrete
or numerical values."))
```

Indirekte Einträge werden in einer an die „Active Values" des Loops-Systems angelehnten Form angegeben.[1] Ein Ausdruck `` #`(<object> <reader>) ``, der einen indirekten Eintrag definiert, besteht aus einem referenzierten Objekt und einer Zugriffsfunktion, die auf dieses indirekte Objekt angewendet wird.[2] Diese Funktion sei als indirekte Lesefunktion bezeichnet.

```
(defvar *estimate-1*
  (make-instance 'estimation
                 :init-estimation ':low))

(defvar *estimate-1-visualizer*
  (make-instance 'overview-gauge
          :dial-value #`(*estimate-1*  current-estimation)
        ...))
```

Die angegebene Zugriffsfunktion ist in diesem Fall die Eintragslesefunktion `current-estimation` des Eintrags `estimation-value` des darzustellenden Objekts `*estimate-1*`. Durch Verwendung der Metaklasse `pcl:indirect-slots-class` evaluiert sich der Ausdruck `(dial-value *estimate-1-visualizer*)` zu dem Ausdruck `:low`. Ändert sich jedoch der Eintragswert des Schätzerobjekts auf z.B. `:high`, so führt eine erneute Evaluierung des obigen

1 Implementierungstechnisch handelt es sich um Readmacros.

2 Die Zugriffsfunktion wird in einer Form angegeben, die der CommonLisp-Spezialform `function` entspricht. In diesem Sinne kann das Readmacro `` #`(object fcn) `` auch als Erweiterung des Readmacros `#'`*fcn*, das für `(function fcn)` steht, angesehen werden.

Ausdrucks zu dem „aktuellen“ Wert `:high`. Dieses kann auch als verzögerte Evaluierung (lazy evaluation, delayed evaluation) des Ausdrucks `#`(*estimate-1* current-estimation)` interpretiert werden [Abelson & Sussman 85]. Damit ist die Anzeige `*estimate-1-visualizer*` (indirekt) an das Schätzerobjekt gekoppelt, ohne daß irgendeine Methode der Visualisierungskomponente geändert werden müßte. Das Visualisierungsobjekt ist für beliebige Zwecke verwendbar. Man beachte: der indirekte Wert des angesprochenen Eintrags ist nicht zwingend. In anderen Einsatzgebieten oder während der Entwicklungsphase (der Anzeige) kann auch ein direkter, „normaler“ Wert verwendet werden.

In der bisherigen Diskussion wurde stillschweigend vorausgesetzt, daß der Wertebereich der indirekten Lesefunktion gleich der von der Anzeige darstellbaren Wertmenge {`:middle, :low, :high`} ist. Dieses muß nicht unbedingt der Fall sein, wie das nachfolgende Beispiel aus einer anderen Domäne zeigt. Auch hier kann diegleiche Anzeige verwendet werden, um eine Überblicksvisualisierung von Schulnoten zu erzeugen.

```
(defclass mark
   (...)
   ((estimation-value :initarg :init-mark
                      :accessor current-mark
                      :type (member :sehr-gut
                                    :gut
                                    :befriedigend
                                    :ausreichend
                                    :mangelhaft
                                    :ungenügend)))
   (:metaclass pcl:indirect-slots-class)
   (:documentation "Mark of a term's report."))
```

Das Codefragment soll andeuten, daß der Eintrag `estimation-value` in diesem Beispiel sechs verschiedene Werte annehmen kann.[1] Eine indirekte Zugriffsfunktion muß eine Konvertierung und Filterung vornehmen. Die Funktion liegt damit in der „Grauzone“ zwischen Anwendung und Visualisierung.

```
(defun non-linear-mark-converter (mark)
  "nicht linearer Konvertierer von Noten in Anzeigewerte."
  (ecase (mark-value mark)
    (:sehr-gut ':high)
    ((:gut :befriedigend :ausreichend) ':middle)
    ((:mangelhaft :ungenügend) ':low)))

(defvar *mark-1*
  (make-intance 'mark :init-mark ':befriedigend))
```

[1] Für die Elemente der Eintragswertemenge wurden deutsche Bezeichner verwendet, da es kein jeweiliges englisches Pendant gibt.

```
(defvar *mark-visualizer-1*
  (make-instance 'overview-gauge
   :dial-value #`(*mark-1* non-linear-mark-converter)
   ...))
```

Zur Änderung der Einschätzung könnte die Anzeige durch Mausinteraktion verändert werden. Ohne die Details zu betrachten, läßt sich sagen, daß ein Schreibzugriff auf einen Eintrag eines Objektes, der mit einem indirekten Wert belegt ist, sich auf das indirekte Objekt auswirken sollte. Hierzu kann eine indirekte Schreibfunktion angegeben werden. Die Definition eines indirekten Wertes wird in einer erweiterten Form verwendet: `#`(`*<object>* *<reader>* *<writer>*`).`[1] Offensichtlich reicht die diskrete Darstellungsgenauigkeit der Anzeige nicht aus, doch dieses ist ein anderes Problem. Es wird angenommen, daß der Generalisierungseffekt erwünscht ist, wenn nicht, könnten numerische Werte verwendet werden.

```
(defun linear-mark-writer (new-value gauge)
  (setf (current-mark *mark-1*)
        (ecase new-value
          (:high ':sehr-gut)
          (:middle ':befriedigend)
          (:low ':mangelhaft))))

(defvar *mark-visualizer-1*
  (make-instance 'overview-gauge
   :dial-value #`(*mark-1* non-linear-mark-converter
                           linear-mark-writer)
   ...))
```

Die Funktion `linear-mark-writer` ist noch ein wenig unmodular, da die dynamisch gebundene Variable `*mark-1*` referenziert wird. Es ist wünschenswert, ein über einen indirekten Objekteintrag referenziertes Objekt bestimmen zu können. Hierzu dient die Funktion `pcl:indirect-object`, die auf ein (Visualisierungs-)Objekt und den Namens eines Eintrags dieses Objekts angewendet wird. Sie liefert das indirekt referenzierte Objekt, sofern eines eingetragen ist, `nil` sonst. Man beachte, daß diese „Rückverfolgung" dynamisch erfolgt: wird ein anderer indirekter Wert eingetragen, so liefert `pcl:indirect-object` auch ein anderes indirektes Objekt!

Damit ändert sich die Definition der Funktion `linear-mark-writer` wie folgt. Es ist sichergestellt, daß `pcl:indirect-object` innerhalb dieser Funktion nicht-`nil` liefert, da sie erst evaluiert wird, wenn ein indirekter Eintrag vorliegt.

```
(defun linear-mark-writer (new-value gauge)
  (setf (current-mark
          (pcl:indirect-object gauge 'dial-value))
        (ecase new-value
```

[1] Die Angabe einer indirekten Schreibfunktion ist optional.

```
            (:high ':sehr-gut)
            (:middle ':befriedigend)
            (:low ':mangelhaft))))
```

In diesem Beispiel wird der Eintrag `estimation-value` des Schätzerobjekts direkt gesetzt. Es kann jedoch ohne weiteres noch eine Kommunikationszwischenstufe in Anwendungs- oder Visualisierungsterminologie realisiert werden (vgl. Abschnitt 4.1). Ein Beispiel in Anwendungsterminologie:[1]

```
(defgeneric re-estimation (mark estimation)
  (:documentation "Neueinschätzung der Benotung"))

(defun linear-mark-writer (new-value gauge)
  (re-estimation
        (pcl:indirect-object gauge 'dial-value)
        (ecase new-value
          (:high ':sehr-gut)
          (:middle ':befriedigend)
          (:low ':mangelhaft))))
```

Die Anwendung muß für die generische Funktion `re-estimation` eine entsprechende Methode bereitstellen.

Die Vorteile einer Realisierung einer Anzeige durch Sichtbereichselemente liegt in der leichten Kombinierbarkeit mit anderen Objekten innerhalb einer Darstellung. Eine Anzeige muß nicht unbedingt allein in einem Fenster dargestellt werden. Wird dieses jedoch gewünscht, so wird die Anzeige mit einer Rahmenbox im Sichtbereich angeordnet. Der Sichtbereich wiederum kann mit einer weiteren Rahmenbox im Dialogfenster fixiert werden. Durch Verwendung anderer Anordnungen können sowohl weitere Sichtbereichselemente hinzukommen als auch weitere Dialogelemente gezeigt werden. Ist der Sichtbereich einer Anzeige rollbar, so kann die Anzeige ohne Programmieraufwand innerhalb ihres Sichtbereichs gerollt werden. Durch Hinzumischung der Klasse `tv:moveable-view-item` (siehe Abschnitt 3.5.3) kann eine Anzeige innerhalb ihres Sichtbereichs mit der Maus verschoben werden.

Durch indirekte Werte von Einträgen erfolgt eine einfach zu steuernde Entkopplung der Visualisierungs- von der Anwendungskomponente. Gezeigt wurde dieses an zwei Visualisierungsbeispielen für unterschiedliche Anwendungen (Schätzungen und Benotungen), welche die gleiche Anzeige verwenden können. Der nächste Abschnitt diskutiert, wie die umgekehrte Richtung, also eine Aktualisierung einer Visualisierung z.B. bei Änderung eines indirekt referenzierten Eintrags einer Anwendungskomponente,

---

[1] In dieser Arbeit werden generische Funktionen in Visualisierungsterminologie als Kommunikationsprotokoll verwendet. Vergleiche z.B. die Funktion `tv:dialog-item-action` in Abschnitt 3.2. Spezielle Visualisierungen im MVC-Schema verwenden häufig eine Anwendungsterminologie.

organisiert wird. Ein Beispiel ist die Änderung des Eintrags `estimation-value` eines Schätzerobjekts durch einen schreibenden Eintragszugriff seitens der Anwendung. Die Anzeige muß den Zeiger aktualisieren. Eine Möglichkeit, dieses zu organisieren, wäre die Implementierung einer Anzeige als Prozeß. Der Anzeigeprozeß überwacht, ob sich der Eintrag geändert hat (polling). Dieses läßt sich jedoch nur in Mehrprozeßbetriebssystemen realisieren. Für das Macintosh-System muß ein anderer Weg gefunden werden.

In den nächsten Beispielen werden ebenfalls Anzeigen verwendet, die jedoch etwas aufwendiger sind und zwei Zeiger benötigen. Anzeigen sollten in einer Standard-Visualisierungsbibliothek untergebracht werden. Die Erstellung einer vollständigen Anzeigenbibliothek überschreitet den Rahmen dieser Arbeit. Eine Ausarbeitung über verschiedene Anzeigentypen und -klassen enthält [Stenger 86]. Bauteile, die in dem in dieser Arbeit vorgestellten System realisiert sind, werden aus dem Modul `vislib` exportiert.

5.2. Dämonen für Eintragszugriffe

Im vorigen Abschnitt wurde erläutert, wie Visualisierungen durch Verwendung von indirekten Eintragswerten an Anwendungen gekoppelt werden. Es muß z.B. eine Anzeige aktualisiert werden, wenn bekannt ist, daß sich der über einen indirekten Eintrag referenzierte Eintragswert eines Anwendungsobjektes geändert hat. Die Änderung dieses Eintrags erfolgt über Eintragsschreibzugriffe. Solche Schreibzugriffe müssen also an die Visualisierungsobjekte übermittelt werden. Hier ist ein allgemeineres Konzept erkennbar. Jedem Eintrag kann eine Menge von „Dämonenfunktionen" zugeordnet werden. Ändert sich der „überwachte" Eintragswert, so „evaluieren sich" die Dämonenfunktionen. Die etwas ungewöhnliche Ausdrucksweise, daß sich eine Funktion selbst evaluiert, soll darauf hindeuten, daß es sich nach außen hin um ein nicht-imperatives Konzept handelt. Daher kommt auch der Begriff „Dämon". Dämonen sind nicht zu verwechseln mit `:after` Methoden des CLOS-Systems und werden auch nicht durch solche realisiert. `:after` Methoden werden i.a. schon durch die Anwendung verwendet! Dämonen für Einträge werden durch Verwendung der neuen Metaklasse `pcl:demon-slots-class` unterstützt. Eine Dämonenzuordnung ist durch folgende Funktion möglich.[1]

```
(pcl:add-slot-if-modified-demon
     <object>
     <slot-name>
     <demon-function>
     &optional (<demon-id> (gensym)))
```

[1] Dämonen können auch für alle Instanzen einer Klasse definiert werden (siehe Anhang B).

Der Parameter `<object>` bestimmt das (Anwendungs-)Objekt, dessen Eintrag `<slot-name>` mit einer Dämonenfunktion `<demon-function>` ausgestattet wird. Als vierter Parameter kann noch ein Lisp-Objekt als Kennung angegeben werden. Verschiedene Dämonen eines Eintrags müssen auch verschiedene Kennungen haben.[1] Die Funktion `<demon-function>` wird mit vier Aktualparametern evaluiert: dem Objekt, dessen Eintrag überschrieben wurde, dem Namens dieses Eintrags sowie dem alten und dem neuen Eintragswert.[2]

Um die Verwendung des skizzierten Ansatzes im Zusammenhang mit Visualisierungen zu testen, wird eine *vorgefertigte* Anwendung mit einer Visualisierung versehen. Als Beispiel dient ein Programm aus [Winston & Horn 89, Kapitel 23]. Es handelt sich um ein (sehr einfaches) System zur Modellierung von Börsenbeziehungen und -einflüssen unter Verwendung von Einschränkungen (constraints).[3] Es geht darum, einen günstigen Moment zum Ankauf von Aktien abzuschätzen. Eine Einschätzung wird durch ein Unsicherheitsintervall in den Grenzen von 0 bis 1 modelliert (siehe Abbildung 5.2). Eine Einschätzung wird auch als Assertion bezeichnet und durch eine Klasse `assertion` (s.u.) modelliert.

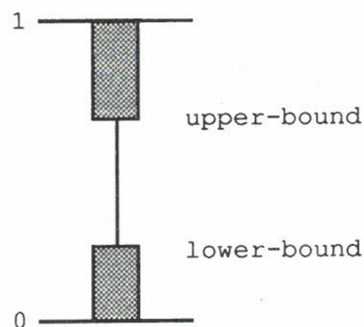


Abbildung 5.2: Obere und untere Schranke der Einschätzung (aus [Winston & Horn 89]).

Assertionen von verschiedenen Börsenteilnehmern stehen in Relation zueinander. Diese Relationen (dargestellt durch Subklassen einer Klasse `constraint`) werden durch Einschränkungserfüllung aufrecht erhalten. Für eine genaue Definition der Verknüpfungsregeln für Einschätzungen wird auf Anhang C oder die Originalliteratur

---

1 Die Kennung wird u.a. dazu verwendet, Dämonen wieder zu löschen (siehe Anhang B). Wird keine Kennung angegeben, so wird eine eindeutige Kennung automatisch erzeugt.

2 Eventuell können als Dämonenfunktionen auch generische Funktionen verwendet werden.

3 Innerhalb des Buches [Winston & Horn 89] dient das Beispiel als Anwendungsbeispiel zur Lehre des Verhaltens von Einschränkungen (constraints) bzw. deren Aufrechterhaltung.

verwiesen. Dieses ist hier nicht der entscheidende Punkt. Es kommt in diesem Beispiel nur darauf an, zu untersuchen, ob Dämonen eine ausreichende Kopplung der Anwendungs- an die Visualisierungskomponente darstellen bzw. welche weiterreichenden Konzepte benötigt werden. Das Beispiel wurde fast vollständig ohne Änderungen übernommen. Die Klasse `assertion` wurde mit der Metaklasse `pcl:demon-slots-class` versehen, so daß die zur Visualisierung relevanten Einträge `lower-bound` und `upper-bound` mit Dämonenfunktionen ausgestattet werden können.

```
(defclass assertion
  ()
  ((name :accessor assertion-name :initarg :name)
   (lower-bound :accessor assertion-lower-bound :initform 0)
   (upper-bound :accessor assertion-upper-bound :initform 1)
   (constraints :accessor assertion-constraints :initform nil))
  (:metaclass  pcl:demon-slots-class))
```

Ein Beispielnetz sieht wie folgt aus.

```
(defun build-example-net ()
  (let ((assertions
         (list (make-instance 'assertion :name 'broker1)
               (make-instance 'assertion :name 'broker2)
               (make-instance 'assertion :name 'broker-opinion)
               (make-instance 'assertion :name 'mystic1)
               (make-instance 'assertion :name 'mystic2)
               (make-instance 'assertion :name 'mystic-opinion)
               (make-instance 'assertion :name 'your-opinion)))
        (constraints
         (list (make-instance 'or-box :name 'broker-constraint)
               (make-instance 'or-box :name 'mystic-constraint)
               (make-instance 'or-box :name 'your-constraint))))
    ... ; "Verdrahtung"
    nil))
```

Das durch diese Funktion erstellte Beispielnetzwerk habe die Struktur aus Abbildung 5.3.

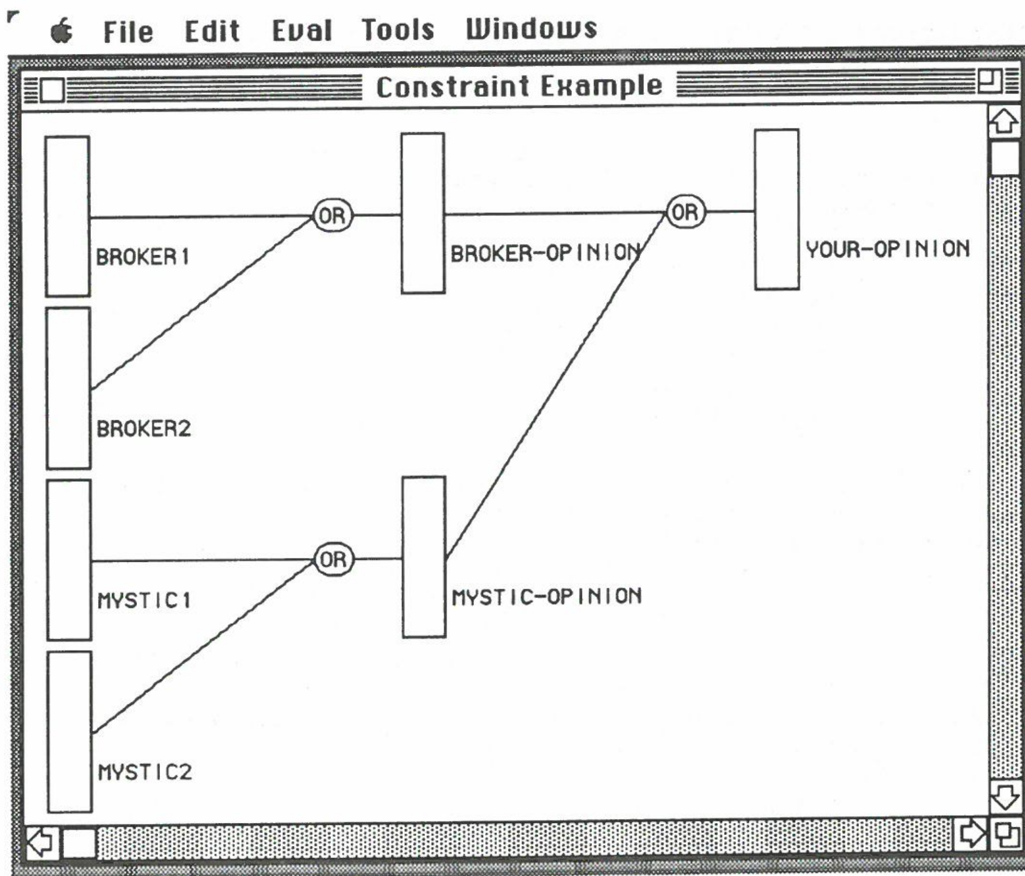


Abbildung 5.3: Einschränkungsnetz dargestellt als DAG.

Die Rechtecke sind die zur Visualisierung der Einschätzungen verwendeten Anzeigen. Dabei sei die 0 am unteren Anzeigenrand und die 1 oben angeordnet (vgl. Abbildung 5.2). Zur Visualisierung einer Liste von Börsenbeteiligten eines speziellen Einschränkungsnetzes dient die Funktion `stock-exchange-connections-visualization`. Nach inzwischen bewährtem Strickmuster wird eine Visualisierung innerhalb eines Sichtbereichs erzeugt. Hierzu läßt sich ebenfalls ein DAG-Layoutmuster verwenden (siehe hierzu Kapitel 3.6.4).

```
(defvar *max-connection-depth* 9
  "Maximale Anzeigetiefe der Börsenverbindungen.")

(defun tautology (ignore)
  "Used as an expansion predicate."
  t)

(defun connection-visualizer ()
  "Edge of a DAG taken from a view-item library and positioned
by references during DAG layout."
  (make-instance 'vislib:line-view-item))
```

```
(defclass stock-exchange-dialog
  (layout:layouted-dialog) ; vordefinierte Dialogklasse
                           ; mit Layoutverwaltung
  ())

(defun make-stock-exchange-dialog (layout)
  (layout:make-layouted-dialog
              :class 'stock-exchange-dialog
              :window-size #@(500 330)
              :window-type ':document-with-zoom
              :window-title "Constraint Example"
              :window-font '(9 "Monaco")
              :default-button nil
              :layout layout))

(defun stock-exchange-connections-visualization (participants)
  "Opens a window and shows the connnections of
the given participants in a scrollable view."
  (let ((stock-exchange-view (layout:make-layouted-view
                               :scroll-bars ':both
                               :bordered-p nil
                               :auto-scrolling t))))

    (make-stock-exchange-dialog (:fbox () stock-exchange-view))
    (setf (layout:layout stock-exchange-view)
          (:vbox ()
                 10
                 (:hbox ()
                        10
                         (:gbox
                         (:annotation
                          ;; Beschriftung von Assertionen
                          ;; durch Generierung weiterer
                          ;; Sichtbereichselememte
                          ;; durch die folgende generische Funktion
                          #'vislib:name-annotation
                           (:dag participants
                                 #'stock-exchange-wizard
                                 *max-connection-depth*
                                 #'tautology
                                 #'application-visualization-coupler
                                 #'connection-visualizer
                                 #'vislib:western-reference
                                 #'vislib:eastern-reference)))))))
```

Eine generische Nachfolgerfunktion ermöglicht die Expansion des Graphen bzgl. unterschiedlicher Knotenobjektklassen. Die Objekte der Klasse `assertion` und `constraint` sind jeweils durch wechselseitige Referenzen miteinander verbunden. Der Anordnungsalgorithmus des `:dag` Musters sorgt dafür, daß zyklische „Nachfolger" unterdrückt werden. Es werden also nicht alle vorhandenen Nachfolgerbeziehungen, die `stock-exchange-wizard` liefert auch durch Graphkanten visualisiert. Da in diesem Beispiel Zyklen nur durch wechselseitige Referenzen zwischen zwei verbundenen Nachbarn im Graphen entstehen, ist dieses hier nicht problematisch, da nicht-gerichtete Kanten verwendet werden. Dieses ist zugegebenermaßen nur bei solchen direkten Zyklen

möglich. In anderen Fällen wird ein Anordnungsalgorithmus benötigt, der auch indirekte Zyklen verarbeiten kann.

```
(defgeneric stock-exchange-wizard (participant)
  (:documentation "Function considered as an agent that
is a stock exchange expert. Evaluating this function
for a special participant returns the participant's
stock exchange connection."))

(defmethod stock-exchange-wizard ((participant assertion))
  "Wizard's information about connections of assertion objects."
  (assertion-constraints participant))

(defmethod stock-exchange-wizard ((participant constraint))
  "Wizard's information about connections of constraint objects."
  (list (constraint-output participant)))

(defmethod stock-exchange-wizard ((participant ternary-constraint))
  "Wizard's information about connections
of ternary constraint objects."
  (list (constraint-output participant)
        (constraint-input-a participant)
        (constraint-input-b participant)))

(defmethod stock-exchange-wizard ((participant binary-constraint))
  "Wizard's information about connections
of binary constraint objects."
  (list (constraint-output participant)
        (constraint-input participant)))
```

Das Aussehen der Knoten ist abhängig von der Objektklasse des Knotens. Für einen Assertionsknoten wird eine Anzeige gewählt, während für einen Einschränkungsobjektknoten ein gerundetes Rechteck mit entsprechender Inschrift verwendet wird. Man beachte die verteilte Repräsentation durch generische Funktionen.

```
(defgeneric appearance (participant)
  (:documentation "This function provides appropriate
appearences for stock exchange participants.
Appearances are represented as view-items"))

(defmethod appearance ((participant or-box))
  "Appearance of or-box constraint objects.
They are represented by OR in a box with rounded rectangles."
  (vislib:make-label "OR"))

(defmethod appearance ((participant and-box))
  "Appearance of and-box constraint objects.
They are represented by AND in a box with rounded rectangles."
  (vislib:make-label "AND"))

(defmethod appearance ((participant assertion))
  "Appearance of assertion objects.
Assertions are represented by two-level-gauges."
  (vislib:make-two-level-gauge
          #`(participant  assertion-lower-bound)
          #`(participant  assertion-upper-bound)))
```

Die Funktion `vislib:make-two-level-gauge` soll eine Anzeige (implementiert als Sichtbereichselement) erzeugen. Es können zwei (indirekte) Werte aus dem Bereich [0.0, 1.0] angezeigt werden.[1] Die indirekten Zugriffsfunktionen `assertion-lower-bound` und `assertion-upper-bound` sind die Lesefunktionen der Einträge `lower-bound` bzw. `upper-bound` der Assertionsobjekte (s.o.). Die Verwendung von Anzeigen zur Visualisierung von Assertionsknoten ist jetzt der entscheidende Punkt. Eine Anzeige zeigt die beiden Einschätzungsniveaus der Assertionsobjekte der Anwendung an. Ändern sich diese Werte, so muß die Anzeige aktualisiert werden. Dieses wird durch Eintragsdämonen initiiert.

```
(defgeneric application-visualization-coupler (object button))

(defmethod application-visualization-coupler (object button)
  (appearance object))

(defmethod application-visualization-coupler ((participant assertion)
button)
   (let ((assertion-gauge (appearance participant)))
      (pcl:add-slot-if-modified-demon
      participant
      'lower-bound
      #'(lambda (assertion-obj name-of-modified-slot
                                old-value new-value)
           (vislib:gauge-update assertion-gauge)))
      (pcl:add-slot-if-modified-demon
      participant
      'upper-bound
      #'(lambda (assertion-obj name-of-modified-slot
                                old-value new-value)
           (vislib:gauge-update assertion-gauge)))
     assertion-gauge))
```

Die Anzeige `assertion-gauge` wird aktualisiert, wenn sich die Eintragswerte `lower-bound` oder `upper-bound` ändern. Die Benachrichtigung erfolgt durch Evaluierung der Funktion `vislib:gauge-update` innerhalb der Eintragszugriffsdämonen. Die Dämonenfunktionen der Anwendungsobjekte stellen eine Verbindung zu dem Visualisierungsobjekt `assertion-gauge` über Abschlüsse (closures) der Lambda-Ausdrücke der Dämonenfunktionen her.[2]

Wie sieht nun ein Beispiel aus? Nachdem zunächst die Funktion

---

[1] In dem Beispiel werden die indirekten Werte als Parameter übergeben. Es wird nicht unmittelbar deutlich, daß sie zur Initialisierung von Anzeigeeinträgen verwendet werden. Dieses muß in einer Dokumentation der Bibliotheksfunktion herausgestellt sein!

[2] Eine Dämonenfunktion hat vier Formalparameter. Durch Verwendung von generischen Dämonenfunktionen könnte eine Benachrichtigung auch durch Methoden für z.B verschiedene Klassen von neuen Werten (4. Parameter `new-value`), d.h. datengetrieben erfolgen.

```
(build-example-net)
```

evaluiert wird, erhält man durch Evaluierung von

```
(stock-exchange-connections-visualization
                broker1 broker2 mystic1 mystic2)
```

das Dialogfenster aus Abbildung 5.3. Die Ausgangsituation ist also gekennzeichnet durch völlige Unsicherheit. Alle Beteiligten haben eine Einschätzung von `lower-bound` gleich 0 und `upper-bound` gleich 1. Angenommen, der eine Börsenmakler `broker1` und das Objekt `mystic2` ändern ihre Einschätzung.

```
(initiate-propagation broker1 0.25 0.75)
(initiate-propagation mystic2 0.85 0.85)
```

Es wird jeweils eine Einschränkungspropagierung gestartet. Abbildung 5.4 zeigt, welche Objekte dadurch in ihrer Einschätzung beeinflußt wurden.

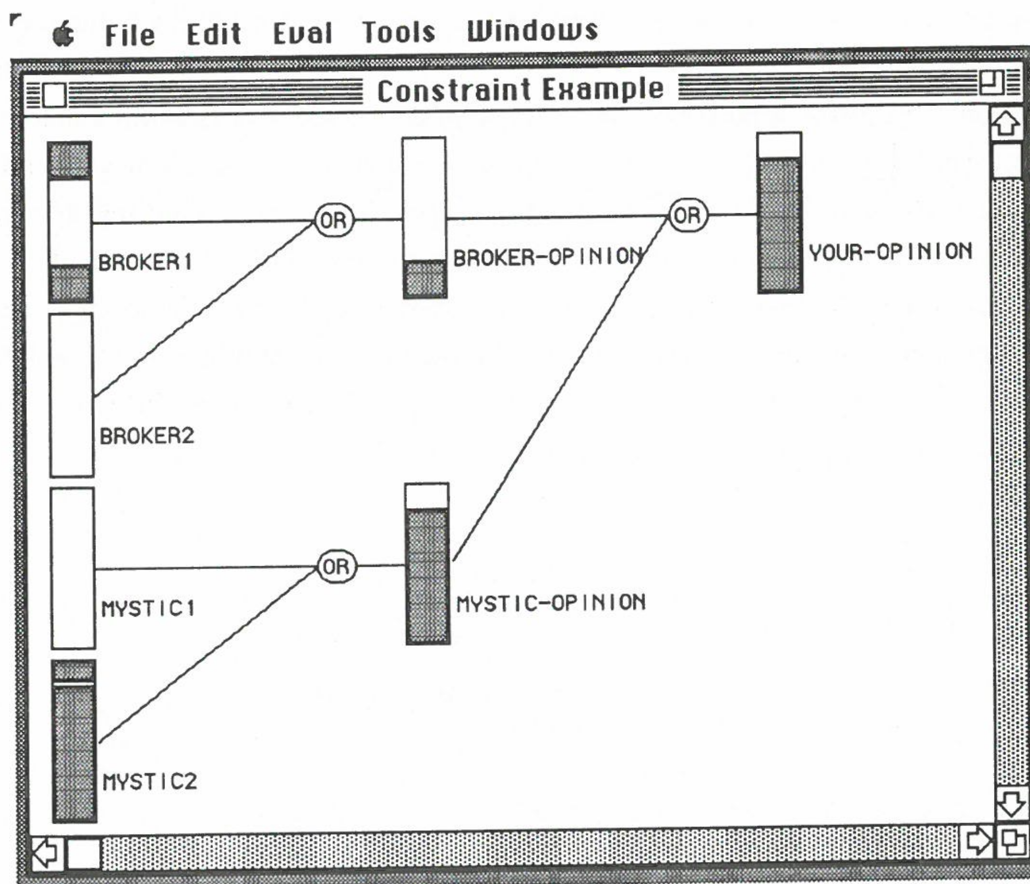


Abbildung 5.4: Zustand des Einschränkungsnetzes nach der Propagierung.

Eine Änderung der Einschätzung bedeutet die Änderung der Werte der Einträge `lower-bound` und `upper-bound` einiger Assertionsobjekte. Das wiederum heißt, daß die Eintragsdämonen aktiviert werden und dadurch die Anzeigeobjekte durch die Funktion `vislib:gauge-update` „benachrichtigt" werden.

Wenn eine Visualisierung nur von den Eintragswerten von Anwendungsobjekten abhängt, sind Dämonen ein ausreichendes Mittel, die Werteänderungen zu registrieren. Statt wie in diesem Beispiel über eine Funktion `vislib:gauge-update` eine sofortige Aktualisierung einer Anzeige zu erzwingen, könnte ebenso zunächst ein „Ereignis" erzeugt und in eine Warteschlange oder eine Agenda eingetragen werden. Dämonenfunktionen bilden eine untere Stufe zur Ankopplung der Visualisierungskomponente, die genügend Flexibilität bietet, geeignete weiterführende Mechanismen zu realisieren.

5.3. Dämonen für Methodenevaluationen

Ein für eine Visualisierung „interessantes Ereignis" kann auch einfach durch eine Evaluierung einer bestimmten Methode definiert sein. Ähnlich wie die Eintragsschreibzugriffe sollten auch die Methoden des CLOS-Systems mit Dämonen versehen werden können. N.b.: `:after` Methoden können hierfür nicht verwendet werden, da diese schon durch die Anwendung (für eigene Zwecke) definiert sein können. Folgendes Konstrukt erlaubt es, Methoden mit Dämonenfunktionen zu versehen, die *vor* der eigentlichen Methode oder *danach* evaluiert werden. Die Methodendämonen werden mit den gleichen Parametern evaluiert wie die eigentliche Methode. Die Lambda-Ausdrücke der Dämonen können – wie im obigen Beispiel geschildert – zur Visualisierungsankopplung verwendet werden.

```
(defclass application
  ()
  ()
  (:documentation "stands for any application class."))

(defmethod application-operation ((x application))
  (print "application-operation: primary")
  nil)

(defmethod application-operation :before ((x application))
  (print "application-operation: before")
  nil)

(defmethod application-operation :after ((x application))
  (print "application-operation: after")
  nil)
```

```
(pcl:add-method-demon
    (pcl::get-method1 #'application-operation
                     '() ; Keine Qualifizierer also
                         ; die Primärmethode
                      (list (find-class 'application)))
                     ;; Berechnung des Methodenobjekts

                     :after ; alternativ :before
                     ;; Die nachfolgende Funktion wird nach
                     ;; der obigen Methode evaluiert.

                     #'(lambda (x)
                         (format t
                          "~%application-operation: ~
                           after primary for ~S"
                          x)))

? (setf appl-instance-1 (make-instance 'application))
#<APPLICATION 2792524>
? (application-operation appl-instance-1)
"application-operation: before"
"application-operation: primary"
application-operation: after primary for #<APPLICATION 2792524>
"application-operation: after"
NIL
```

Methodendämonen werden in derselben Reihenfolge evaluiert wie sie mit `pcl:add-method-demon` definiert wurden. Das gewünschte Verhalten konnte durch geringfügige Änderungen der CLOS-Implementation erreicht werden (Hinzufügung eines Eintrags für alle Methodenobjekte, veränderte Funktion zur Berechnung der kombinierten, d.h. aus `:after`, `:before` und jeweiligen Dämonenfunktionen zusammengesetzten, Methode. Inwieweit eine Änderung der CLOS-Implementation durch Verwendung der Form `define-method-combination` [Keene 89] vermieden werden kann, konnte noch nicht getestet werden, da diese Form in der für diese Arbeit vorliegenden Fassung des CLOS-Systems (Mai 89) noch nicht implementiert ist. Die Methodendämonen stellen also zunächst nur eine Studie dar, sind evt. aber nicht in allen CLOS-Systemen so zu realisieren.

Eintragszugriffsdämonen ließen sich evt. ersetzen durch Methodendämonen für die Zugriffsmethoden. Eintragsdämonen haben aber ihre Existenzberechtigung, da nicht unbedingt für jeden Eintrag eine Zugriffsmethode definiert sein muß. Methodendämonen sind nicht unbedingt zu einer nachträglichen „geheimen Überwachung" von Methodenevaluationen konzipiert. Sie stellen ein Konstruktionsmittel im Sinne einer strukturierten Programmiertechnik dar, um eine Visualisierung von der Anwendung zu

---

[1] Methoden können aus einer generischen Funktion durch `pcl::get-method` ermittelt werden. Die Parameter von `pcl::get-method` sind: die generische Funktion der gesuchten Methode, eine Liste der Qualifizierer (`:before`, `:after`, ...) und eine Liste der Klassen der spezialisierten Argumente. Da das Metaobjekt-Protokoll von CLOS sich noch in der Entwicklung befindet, ist das Symbol noch nicht aus dem Modul `pcl` exportiert.

trennen. Als Methodendämon kann z.B. eine generische Funktion verwendet werden; sie wird mit den gleichen Parametern evaluiert wie die eigentliche Methoden (s.o.). Diese Funktion könnte als „Kopplungskontrakt" zwischen Anwendung und Visualisierung vorher vereinbart worden sein. Die Visualisierungskomponente könnte geeignete Methoden für bestimmte Parameter bereitstellen. Man erkennt, daß hier ein Konstruktionsmittel eingeführt wurde, daß Erweiterungen und Abstraktionen zuläßt.

5.4. Elementarereignisse

Betrachtet man das Börsenbeispiel aus Abschnitt 5.2, so stellt man fest, daß die gemeinsame Evaluierung der beiden Ausdrücke

```
(initiate-propagation broker1 0.25 0.75)
(initiate-propagation mystic2 0.85 0.85)
```

eventuell als eine Einheit bzgl. der Visualisierung betrachtet werden sollte. Sie bilden ein elementares Ereignis. Der Zwischenzustand der Propagierung nach Evaluierung der ersten Funktion wäre dann nicht darzustellen. Hier wird ein Konstruktionsmittel benötigt, um beide Ausdrücke als elementar zu klassifizieren. Folgende einfache Form leistet dieses:

```
(tv:as-elementary-event
  (initiate-propagation broker1 0.25 0.75)
  (initiate-propagation mystic2 0.85 0.85))
```

Die Form `tv:as-elementary-event` bewirkt, daß in allen Sichtbereichen die Zeichenoperationen nicht direkt ausgeführt, sondern nur vermerkt werden. Alle Zeichenoperation werden „in einem Rutsch" am Ende des Rumpfes dieser Form ausgeführt.[1] Deklarationen von elementaren Ereignissen sind durch das Verwaltungssystem für Sichtbereiche realisiert.

Auch Methoden der Anwendungskomponente können als Elementaroperation deklariert werden. Ein Beispiel hierfür wäre eine Methode zum Rücksetzen des Einschränkungsnetzes im Börsenbeispiel.

```
(defmethod reset-example-net ()
  (setf (assertion-lower-bound broker1) 0)
  (setf (assertion-upper-bound broker1) 1)
  ...
```

[1] Die gleichen Ausgabetechniken werden z.B. bei den Verschiebungen von Sichtbereichselementen verwendet. Es wird zunächst ermittelt und vermerkt, welche Objekte zu löschen und welche neu zu zeichnen sind. Erst „am Ende" erfolgt die Löschung und dann die Zeichnung aller bewegten Sichtbereichselemente. Dadurch kann ein unnötiges Zeichnen einzelner Sichtbereichselemente vermieden werden.

```
  (setf (assertion-lower-bound your-opinion) 0)
  (setf (assertion-upper-bound your-opinion) 1))
```

Diese Methode kann mit folgender Form als Elementarmethode definiert werden:

```
(tv:defelementary-event reset-example-net ())
```

Die Zeichenoperationen aller Sichtbereiche werden bis zum Ende des Methodenrumpfes verzögert. Es entsteht der Eindruck einer elementaren Operation. Das Makro `tv:defelementary-event` hat folgende allgemeine Form.

`tv:defelementary-event` **{** *qualifiers* **}*** (**{** *specializers* **}***)

Die Form ist identisch mit dem Kopf von `defgeneric`. Die Implementation der Form verwendet die im vorigen Kapitel besprochenen Methodendämonen.

5.5. Zeitführung

In dem Beispiel zur Vererbungshierarchievisualisierung in Kapitel 3.6.4 wurde angesprochen, daß Kontroll- oder Fokussierungsmögichkeiten für eine Visualisierung von erheblicher Wichtigkeit sind. Kontrollmöglichkeiten werden auch in Bezug auf eine Zeitführung benötigt. Innerhalb des Börsenbeispiels ist es z.B. während der Programmentwicklungsphase interessant, daß Verhalten der Einschränkungs-propagierung zu verfolgen. Dies ist also genau das Gegenteil zur Definition von Elementarereignissen. Nach jedem Setzen eines Eintrags `lower-bound` oder `upper-bound` eines Assertionsobjektes soll die Propagierung gestoppt werden, bis eine Schaltfläche „Proceed" betätigt wird. Eine Schaltfläche kann ohne Schwierigkeiten in das Layout des Dialoges integriert werden.

```
(defun stock-exchange-connections-visualization (participants)
  "Opens a window and shows the connnections of
the given participants in a scrollable view."
  (let ((stock-exchange-view (layout:make-layouted-view
                              :scroll-bars ':both
                              :bordered-p nil
                              :auto-scrolling t))
        (proceed-button (tv:make-dialog-item
                              :class 'proceed-button
                              :dialog-item-text "Proceed"
                              :dialog-item-enabled-p nil)))

    (make-stock-exchange-dialog (:vbox ()
                                       (:fbox () stock-exchange-view)
                                       5
                                         (:hbox (:height 1/16)
                                                5
                                                 (:fbox ()
                                                   proceed-button)
                                                20)
                                      6))
    ...))
```

Es tritt hier also genau der Fall auf, der als Motivation zu einer deklarativen Layoutbeschreibung in vorigen Kapiteln geschildert wurde: es stellt sich nachträglich heraus, daß noch ein weiteres Dialogelement benötigt wird (Abbildung 5.5).

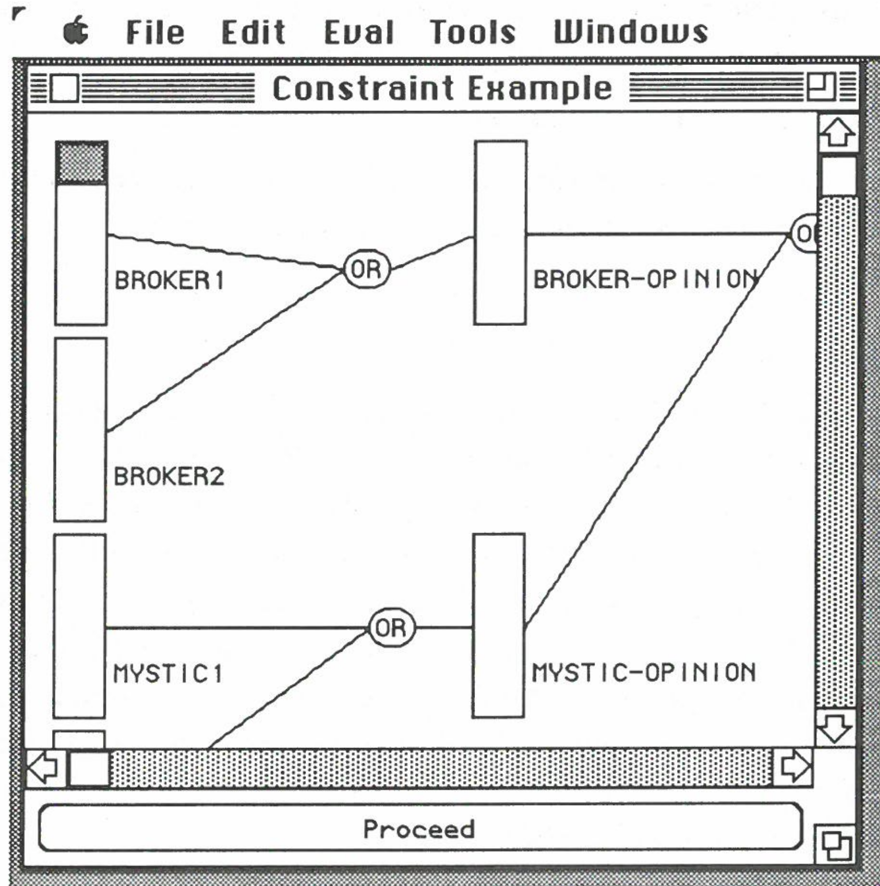


Abbildung 5.5: Einschränkungsnetz mit Schaltfläche zur Überwachung der Propagierung.

Eine Zeitführung kann durch Eintragszugriffsdämonen integriert werden. In der Methode `application-visualization-coupler` werden weitere Dämonen an die Einträge gebunden. Die Funktion `pcl:slot-demon-object-notifier` erzeugt eine Funktion, die einem Objekt `button` die Nachricht `wait-for-proceed` sendet.[1]

[1] Die Konzepte „Senden einer Nachricht"und „Evaluieren einer generischen Funktion" sind äquivalent, wenn für eine generische Funktion nur Methoden mit nur einem spezialisierten Argument definiert sind. Ist dieses der Fall, können die obigen Ausdrücke synonym verwendet werden.

```
(defmethod application-visualization-coupler
          ((participant assertion) button)
  (let ((assertion-gauge (appearance participant)))
       (pcl:add-slot-if-modified-demon
       participant
       'lower-bound
         (pcl:slot-demon-object-notifier  #'wait-for-proceed
                                          button))
       (pcl:add-slot-if-modified-demon
       participant
       'upper-bound
         (pcl:slot-demon-object-notifier  #'wait-for-proceed
                                          button))
    (pcl:add-slot-if-modified-demon
     participant
     'lower-bound
     #'(lambda (assertion-obj name-of-modified-slot
                              old-value new-value)
         (vislib:gauge-update assertion-gauge)))
    (pcl:add-slot-if-modified-demon
     participant
     'upper-bound
     #'(lambda (assertion-obj name-of-modified-slot
                              old-value new-value)
         (vislib:gauge-update assertion-gauge)))
    assertion-gauge))
```

Man definiert eine Klasse zur Beschreibung der zur Kontrolle verwendeten Schaltfläche. Es wird eine Methode zur Behandlung der Nachricht `wait-for-proceed`, die vom Dämon (s.o.) gesendet wird, sowie eine Methode zur Behandlung von Mausklicks in der Schaltfläche benötigt. Man beachte: die generische Funktion `tv:dialog-item-action` wird vom Allegro-Laufzeitsystem asynchron evaluiert, wenn auf ein Dialogelement geklickt wird.[1]

```
(defclass proceed-button
  (tv:button-dialog-item)
  ())
```

---

[1] Leider kann die Methode `wait-for-proceed` nicht ereignisorientiert programmiert werden. Hierzu wäre ein „Stackgroup"-Konstrukt nötig. Der gesamte Evaluierungskeller muß „eingefroren" werden. Die Evaluierung kann nach Betätigung der Schaltfläche im gleichen Zustand fortgesetzt werden. Ein solches Konstrukt stellt die Programmierumgebung nicht zur Verfügung. Da die Anwendung unangetastet bleiben soll, kann es auch nicht ohne weiteres durch sog. „Continuations" [Abelson & Sussman 85] simuliert werden. Obwohl hierdurch zwar der Evaluierungszustand an der Stelle der Erzeugung der „Continuation" festgehalten und gespeichert werden kann – es kann also nach Betätigung der Schaltfläche mit der Evaluierung fortgefahren werden –, so kann dennoch nicht ohne Änderung der Anwendungsfunktion `initiate-propagation` die Evaluierung der Funktion (temporär) abgebrochen werden. Hierzu wäre eine weitere „Continuation" oder ein „Catch" notwendig. In CommonLisp lassen sich (mit vertretbarem Programmieraufwand) nur „Continuations" erzeugen, die durch Sprünge innerhalb der Blockstruktur nachgebildet werden können. Es macht wenig Sinn, eine solche „Continuation"-Nachbildung in eine Datenstruktur zu speichern, da man wegen der dynamischen Lebensdauer der Blöcke nicht von „außen" in einen Block „springen" kann (vgl. [Hennessy 89]).

```
(defmethod wait-for-proceed ((button proceed-button) &rest ignore)
  "Methode für die vom Dämon versendete Nachricht."
  (unless tv:*elementary-event-p*
    (tv:dialog-item-enable button)
    (let ((dialog (tv:own-dialog button)))
      (loop
        (if (stock-exchange-proceed-p dialog)
          (progn
            (setf (stock-exchange-proceed-p dialog) nil)
            (return))))))))

(defmethod tv:dialog-item-action :after ((button proceed-button))
  "Methode zur Behandlung von Mausklicks auf
'Proceed'-Schaltflaechen."
  (tv:dialog-item-disable button)
  (setf (stock-exchange-proceed-p (tv:own-dialog button)) t))
```

5.6. Vergleich und Zusammenfassung

Indirekte Werte von Objekteinträgen erinnern in Bezug auf die indirekten Erfragefunktionen an `if-needed` Facetten in rahmenorientierten Wissensrepräsentationssprachen [Roberts & Goldstein 77, Bobrow & Winograd 77]. Auch hier können Funktionen angegeben werden, die evaluiert werden und deren Ergebnis dann verwendet wird. Im Gegensatz zu der indirekten Referenz werden in vielen rahmenorientierten KI-Systemen die durch `if-needed` Facetten ermittelten Werte dann aber evt. im Objekteintrag gespeichert. Sie bekommen so den Charakter einer Funktion zur Ermittlung eines Standardwertes. Dieses Verhalten ist im Zusammenhang mit Visualisierungen nicht erwünscht. Bei einem weiteren indirekten Lesezugriff soll der evt. neue Wert ermittelt werden.

Aufträge zu einer Aktualisierung einer Visualisierung lassen sich durch Eintrags- oder Methodendämonen erstellen. Hier wird eine völlig andere Sichtweise als im BALSA-II System von Brown vertreten (vgl. Abschnitt 4.3). Ereignisse gehen nicht von einem Algorithmus aus, sondern von den einzelnen Objekten selbst. Hier wird also eine objektorientierte, dezentrale Sicht unterstützt. Das Argument, daß sich nicht alle für eine Visualisierung relevanten Informationen durch eine prozedurale Abstraktion im Programm widerspiegeln, trifft dennoch zu (vgl. Abschnitt 4.3, Prozedur *ElementInPlace*). Andererseits sollten die auftretenden prozeduralen Abstraktionen auch zur Ankopplung der Visualisierungskomponente ausgenutzt werden. Änderungen des Programmtextes zur Einbettung einer dedizierten Visualisierungsfunktion sind immer noch möglich.

Im MVC-Schema wären Anzeigen als Sichtobjekte (Instanzen der Klasse `View`) zu realisieren. Die Berechnung der indirekten Objekte (durch die Funktion

`pcl:indirect-object`) ist vergleichbar mit der Nachricht `model`, die ein Sichtobjekt im Smalltalk-80-MVC-Schema an sich selbst sendet, um sein Modell zu bestimmen. Im Gegensatz hierzu kann bei Verwendung von indirekten Einträgen nicht von *dem* Modell gesprochen werden, da verschiedene indirekte Objekte existieren können. Anders ausgedrückt: eine Menge von Anwendungsobjekten muß nicht als *ein* Modell an eine Visualisierung gekoppelt werden. Bei Verwendung von indirekten Einträgen erfolgt keine explizite, statische Zuordnung von einem Modell zu einem Sichtobjekt, sondern eine implizite, dynamische von evt. mehreren Anwendungsobjekten (Modelle) zu einem Visualisierungsobjekt (view).

Sichtbereiche sind mit der abstrakten Depiktion des „Artist"-Modells aus Kapitel 4.5 vergleichbar, deren Elemente mit den in dieser Arbeit vorgestellten Sichtbereichselementen korrespondieren. Die Verwaltung der Sichtbereiche übernimmt die Koordination der Abbildung (Komponente Renderer). In Kapitel 3 wurde an mehreren Beispielen geschildert, wie man Sichtbereichselemente verwendet und anpaßt, um sie für bestimmte Visualisierungsaufgaben einzusetzen. Die Aufgabe des „Artists", also die Überführung einer Menge von Anwendungsobjekten in eine visuelle Repräsentation zusammen mit der Aktualisierung der Anwendung, wurde an mehrere Objekte delegiert. Durch indirekte Eintragsreferenzen können (ggf. mehrere) Anwendungsobjekte an ein Visualisierungsobjekt gekoppelt werden. Aktualisierungsinformationen werden durch Eintrags- und/oder Methodendämonen an die Visualisierungsobjekte übermittelt. Die Anordnung von Visualisierungskomponenten wird deklarativ durch Layoutangaben beschrieben. Ein Objekt „Artist", das alle diese Aufgaben zentralisiert, existiert nicht.

Nun ist die Erstellung von Visualisierungen – auch mit den hier vorgestellten Werkzeugen – nicht unbedingt trivial. Im nächsten Abschnitt wird ein Beispiel für eine konzeptionelle Visualisierung vorgestellt, das eine Motivation zur Erstellung einer Visualisierung liefern soll.

5.7. Motivation zur Verwendung von Visualisierungen

Wie schon in Kapitel 1 erwähnt, treten bei größeren Anwendungen sehr große Datenmengen auf. Visualisierungen können verwendet werden, eine Interpretation (Auswertung) dieser Daten zu erleichtern. Hierzu sind besonders konzeptionelle, problemnahe Visualiserungen geeignet, da sowohl das Wesentliche betont als auch evt. die Datenmenge reduziert werden kann. Dieses sei an einem kurzen Beispiel aus der Bildverarbeitung betrachtet [Ballard & Brown 82, Seite 296 ff.]. In einem Bild seien die

Kanten eines (nicht durchsichtigen) Würfels in einer festgelegten Perspektive dargestellt (s. Abbildung 5.6). Aus Teilinformationen über Kanten- und Ecktypen können Informationen über die möglichen Typen von anderen Kanten erschlossen werden. Programmtechnisch wird das Auswerten von Teilbeschriftungen durch einen Einschränkungserfüllungsalgorithmus realisiert (Waltz-Algorithmus, edge labeling). Eine strukturelle Visualisierung könnte ein hierarchisches Einschränkungsnetz in Graphform zeigen.

Die Typen werden durch eine Beschriftung der Kanten codiert. Man unterscheidet zwischen konkaven und konvexen Kanten sowie Verdeckungskanten, d.h. Kanten auf deren einen Seite sich der Würfel, auf deren anderen Seite sich der Hintergrund befindet. Konkave Kanten entstehen z.B. wenn der Würfel an eine Wand angrenzt oder andere Körper einer Szene an den Würfel angrenzen. Es existieren folgende Beschriftungen:

+	:	konkave Kante
–	:	konvexe Kante
Pfeil	:	Außenkante, wenn man in Pfeilrichtung schaut, liegt rechts des Pfeils der Körper.

Es wird hier nicht detailliert auf den Algorithmus eingegangen. Eine Auflistung des Programmcodes befindet sich in Anhang C. Wichtig ist nur, daß die Kanten im Einschränkungssystem als gerichtet repräsentiert sind. Es gibt also zwei Pfeil- bzw. Kantentypen:

IN	:	Ist eine Kante PQ mit IN beschriftet, so zeigt der Pfeil von P nach Q.
OUT	:	Ist eine Kante PQ mit OUT beschriftet, so zeigt der Pfeil von Q nach P.

In der visuellen Darstellung sollte diese Unterscheidung jedoch nicht erscheinen. Die Visualisierung leistet damit schon eine Datenreduktion.

Abbildung 5.6 zeigt eine Darstellung eines Würfels. Man erkennt die verschiedenen Ecktypen des Würfels. Es handelt sich um eine konzeptionelle Visualisierung, denn das Konzept der Ecke spielt für den Algorithmus keine Rolle, wohl aber für die Visualisierung. Die „Ecken" im Algorithmus sind die Einschränkungsobjekte. Nur ein Teil der Einschränkungsobjekte wird gezeigt. Das (hierarchische) Einschänkungsnetz besteht aus weiteren, nicht gezeigten Einschränkungsobjekten, da die Kanten jeweils in zwei Richtungen repräsentiert sind. Dieses ist für die Visualisierung der Kantenbeschriftung aber nicht relevant.

Die Erzeugung der Visualisierung ist etwas aufwendiger als die der vorigen Beispiele und beträgt ca. fünf Seiten Code. Es muß ein Layoutmuster für einen Würfel und eine Zeichenfunktion für die Ecken und Kanten definiert werden (siehe Anhang C).

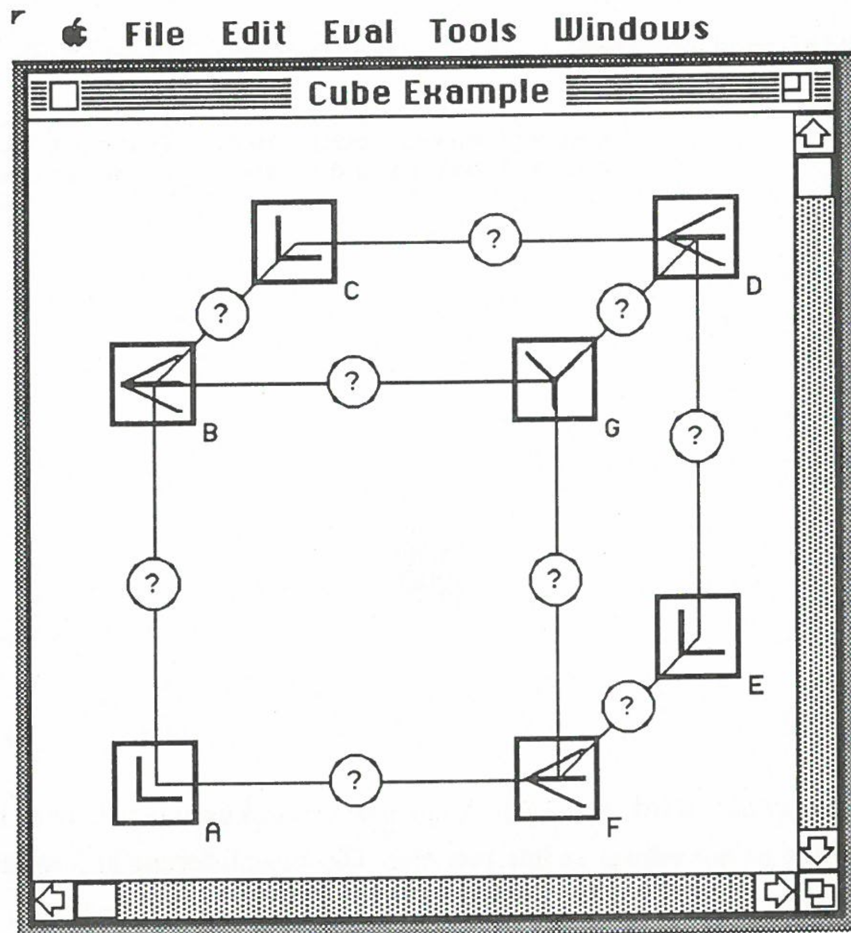


Abbildung 5.6: Konzeptuelle Visualisierung des Einschränkungsnetzes zur Kantenbeschriftung. Die Piktogramme an den Ecken zeigen den Eckentyp (L-Ecke, K-Ecke, und Y-Ecke). Das Netz befindet sich im Grundzustand.

Abbildung 5.6 zeigt die Beschriftung der Würfelkanten (innerhalb eines Kreises) im nicht initialisierten Einschränkungsnetz. Zur Implementation des Einschränkungsnetzes wurde das System „Consat" der „Babylon KI-Werkbank" verwendet [Christaller et al. 89]. Die Beschriftung ? steht für den Wert `unconstrained`. Die Visualisierungsobjekte sind mit den in den vorigen Abschnitten dieses Kapitels vorgestellten Techniken an die Anwendungsobjekte im Einschränkungsnetz gekoppelt.

Für eine Teilbeschriftung kann über eine Einschränkungspropagierung die Menge aller gemäß der Form eines Würfels konsistenten Kantenbeschriftungen ermittelt werden. Das Einschränkungssystem verwendet die Methode `:satisfy`, die als elementares Ereignis deklariert wird.

```
? (tv:defelementary-event :satisfy (consat::constraint-base t))

? (satisfy cube :with a-f = (:one-of out)  f-e = (:one-of out)
                      e-d = (:one-of out)  d-c = (:one-of out)
                      c-b = (:one-of out)  b-a = (:one-of out)))
((A-B IN)
 (B-A OUT)
 (B-C IN)
 (C-B OUT)
 (C-D IN)
 (D-C OUT)
 (D-E IN)
 (E-D OUT)
 (E-F IN)
 (F-E OUT)
 (F-A IN)
 (A-F OUT)
 (B-G +)
 (G-B +)
 (D-G +)
 (G-D +)
 (F-G +)
 (G-F +))

?
```

Das Ergebnis wird als Lisp-Ausdruck zurückgeliefert.[1] Die symbolische Repräsentation ist nur schwer zu interpretieren. Die Visualisierung in Abbildung 5.7 zeigt die Beschriftung einer einzelnen Kante sofort. Anstelle von Pfeilen (s.o.) wurden hier allerdings Halbschalen jeweils in Richtung des Körpers vorgesehen. Es kann sofort die Seite des Körpers ermittelt werden.

[1] `p-q` bezeichnet eine Kante von der Ecke P zur Ecke Q.

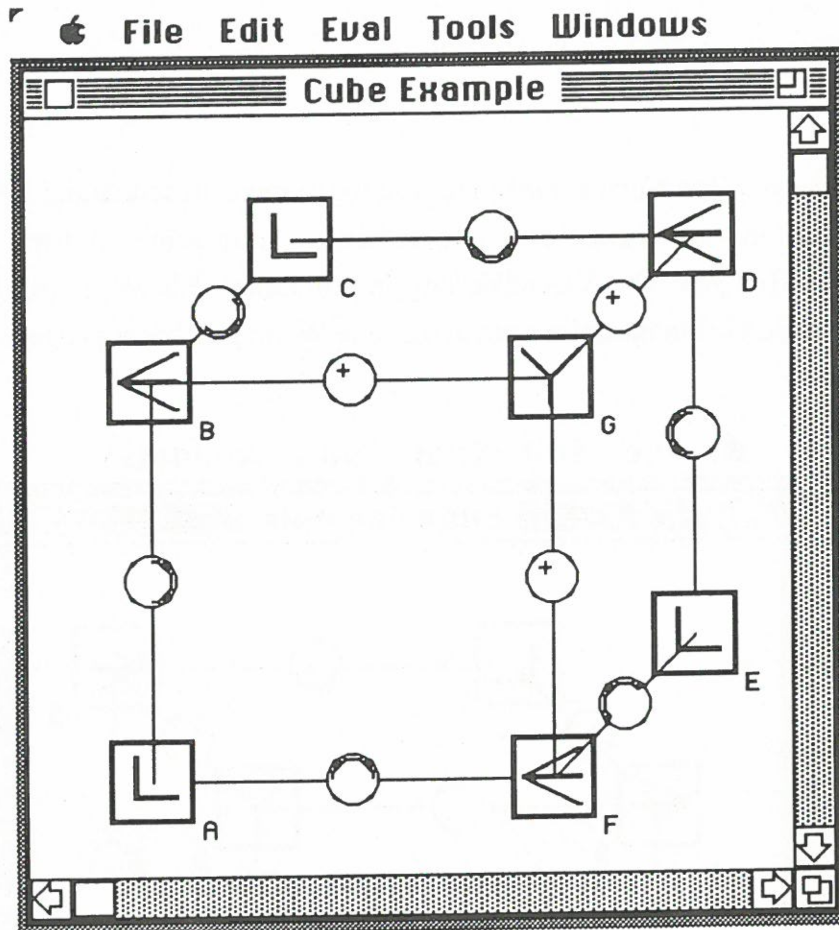


Abbildung 5.7: Zustand des Einschränkungsnetzes zur Kantenbeschriftung nach der Propagierung mit einer eindeutigen Lösung.

In diesem Beispiel gibt es für jede Kante genau eine Lösung. Dieses muß nicht der Fall sein, wie das nächste Beispiel zeigt.

```
? (tv:as-elementary-event (restore-state))
...
? (satisfy cube :with a-f = (:one-of out))

((A-B IN -)
 (B-A OUT -)
 (B-C - IN)
 (C-B - OUT)
 (C-D IN -)
 (D-C OUT -)
 (D-E - IN)
 (E-D - OUT)
 (E-F IN)
 (F-E OUT)
 (F-A IN)
 (A-F OUT)
 (B-G +)
 (G-B +)
```

```
(D-G +)
(G-D +)
(F-G +)
(G-F +))

?
```

Nach Rücksetzen des Netzes wird eine unterbestimmte Beschriftung propagiert. Im obigen Ergebnisausdruck erscheinen zwei mögliche Werte für einige Kantenbeschriftungen. Die Visualisierung in Abbildung 5.8 zeigt ein klareres Bild. Innerhalb der Beschriftungskreise werden mehrere Werte gleichzeitig angezeigt.

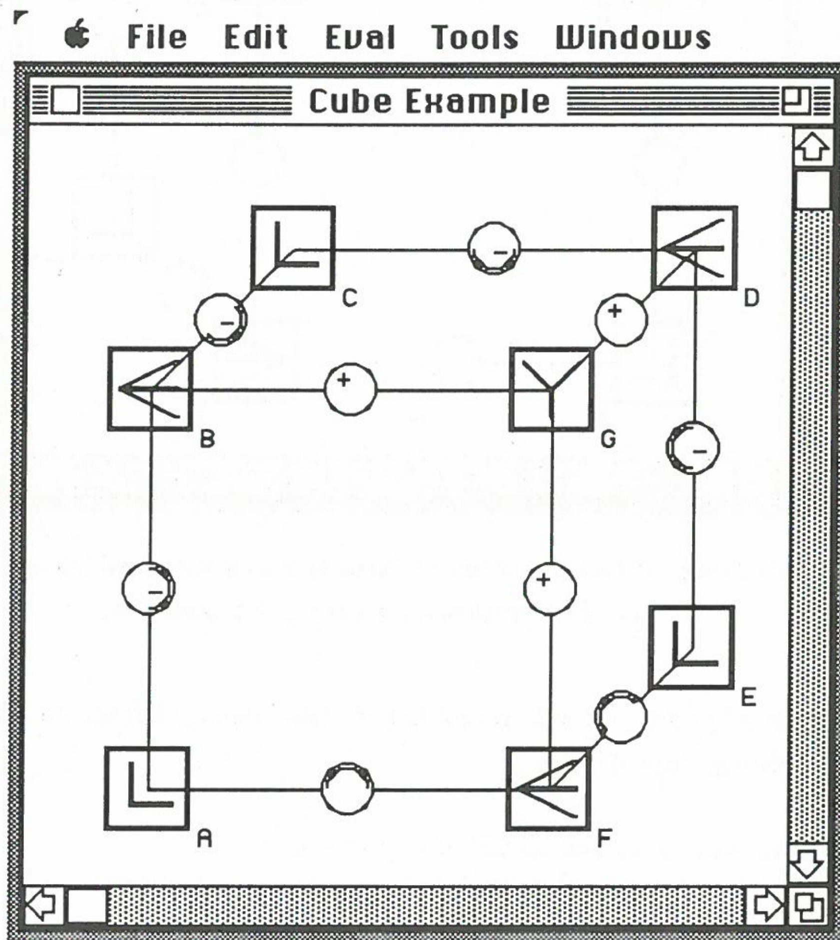


Abbildung 5.8: Zustand des Einschränkungsnetzes zur Kantenbeschriftung nach der Propagierung mit einer mehrdeutigen Lösung.

Das Beispiel zeigt, daß eine visuelle Darstellung helfen kann, symbolische Strukturen schneller zu interpretieren. Es hängt allerdings von der Anwendung ab, ob es sich lohnt, eine Visualisierung zu erstellen. Mit den hier vorgestellten Werkzeugen sollte dieses erheblich leichter sein.

6. Visuelle Sprachen - ein Ansatz zur Formalisierung

6.1. Visuelle Programmiersysteme

Einen Überblick über visuelle Programmiersprachen und -systeme bietet [Myers 86a, Myers 88]. Er definiert den Begriff „visuelles Programmiersystem" als ein System, das es erlaubt, „ein Programm in zwei- oder mehrdimensionaler Weise zu spezifizieren" (Übers. R.M.)[1]. Er sieht die Vorteile u.a. bei einer Anwendung für Nicht-Programmierer, die „kleinere" Anpassungen von bestehenden Systemen dadurch selbst vornehmen können. Insbesondere sieht er Möglichkeiten einer Einbettung von Programmiertechniken, die an Beispielen orientiert sind oder Programme anhand von Beispielen erzeugen (Programming with Example bzw. Programming by Example). Eine visuelles Programmiersystem zur Erzeugung von Benutzerschnittstellen [Myers 86b], das von ihm erstellt wurde, setzt diese Ideen in die Tat um. Aufgrund der Mehrdeutigkeit von visuellen „Ausdrücken" ist es jedoch erforderlich, eine Dialogkomponente zu integrieren, die den Benutzer (in nicht visueller Form, also textuell) über jede aus den Beispielzeichnungen berechnete Inferenz befragt und sich somit „absichert". Benutzerschnittstellen lassen sich dennoch recht leicht erzeugen. Allerdings handelt es sich um ein System, das speziell für diesen Bereich ausgelegt ist, aber genau die Konzepte der Anwendungsdomäne auch visuell repräsentieren kann.

Auf der anderen Seite des Spektrums liegen Systeme wie Prograph [Matwin & Pietrzykowski 85]. Hier erfolgt eine 1:1-Abbildung der Konstrukte klassischer Programmiersprachen in graphische Formen.[2] Der Kontrollfluß wird durch Datenflußdarstellungen angezeigt. Es erfolgt eine Abstraktionsbildung durch Hierarchisierung. Datenflußdarstellungen sind wegen des Wirrwarrs von Linien aber nicht für alle Bereiche geeignet. Außerdem wird hier ein Nachteil vieler visuellen Programmierysteme deutlich: sie verbrauchen relativ viel Bildschirmplatz. Weiterhin sind die meisten Systeme in ihren Darstellungsformen auf vorgegebene Konstrukte fixiert.

Diesem Manko versucht das GARDEN-System von Reiss [Reiss 87a] zu begegnen. Auf der Grundlage des Objektsystems GARDEN [Reiss 87b] können visuelle

---

1 Dabei werden konventionelle Programmtexte nicht als zweidimensional betrachtet, da sie von Compilern oder Interpretern als eindimensionaler Datenstrom (Stream) verarbeitet werden.

2 Für das Prograph-System wurde inzwischen Erweiterungen entwickelt zur visuellen Darstellung von Objekten. Die Visualisierung bleibt aber von struktureller Natur.

Darstellungen erstellt und integriert werden. Diese graphischen Darstellungen können dann editiert und „evaluiert“ werden. Evaluieren einer visuellen Darstellung bedeutet dabei Evaluieren des Objekts, für das diese Darstellung steht. Dabei geht er von einer 1:1-Abbildung von Objekt und Darstellung aus. Zur Erstellung einer Objektdarstellung sind verschiedene graphische Editoren vorgesehen. Die Grundidee der Abbildung von Objekt auf Visualisierung beruht auf ähnlichen Ideen wie Myers' System Peridot: auf der Grundlage von Beispielen (stylized examples), die ein Benutzer durch Interaktion graphisch aus vorgegebenen Basisformen definiert, wird eine Interaktionskomponente (z.B. ein Menü) entworfen. Geometrische Transformationen (Translation, Drehung, Skalierung) werden dann vom System gemäß dem aktuellem Verwendungskontext selbsttätig durchgeführt. Es sollen formularorientierte und graphorientierte Grundlayouts von aus Basiskomponenten zusammengesetzten Objekten unterstützt werden.

Graphische Editierfunktionen werden auf Zustandsänderungen der zugrundeliegenden Objekte abgebildet. Der Autor verschweigt jedoch nicht, daß für Nicht-Standardverhalten das gewünschte Verhalten „von Hand“ programmiert werden muß. Bis 1987 wurde eine Prototypimplementation erstellt.

Einige einfachere Systeme orientieren sich auch direkt an bekannten Visualisierungen von abstrakten Maschinen, wie etwa endliche Automaten [Jacob 85]. Zur Programmierung im Großen scheinen mir diese Systeme allerdings ungeeignet zu sein. In diesem Bereich ist es wichtig, das Zusammenspiel von Teilmodulen zu visualisieren. Ein Beispiel hierzu wäre Pegasys [Moriconi & Hare 85]. Doch diese Systeme würde man wieder in den Bereich der Unterstützungssysteme und nicht in den Bereich der visuellen Programmiersysteme einordnen.

6.2 Visuelle Sprachen

Eine ähnliche Einteilung in lexikalische und syntaktische Komponenten etc. wie bei formalen oder natürlichsprachlichen Systemen kann auch bei visuellen Sprachen erfolgen. [Selker & Koved 88] definieren als Pendant zum Alphabet die Menge der räumlichen, die (Bildschirm-)Oberfläche betreffenden sowie auch zeitlichen Techniken, die in einer visuellen Sprache verwendet werden, um eine Bedeutung zu vermitteln. Die Bestandteile dieser Menge sind Piktogramme, Symbole (z.B. das Zeichen „a“) oder Oberflächeneigenschaften wie Sättigung und Intensität (von Farben) sowie auch Textur. Chang [Chang 87] spricht auf der Alphabetebene (iconic system) von einer Menge von „verwandten“ (related) Piktogrammen. Auf der Wortebene können komplexe Piktogramme aus einfacheren zusammengesetzt werden.

Konzepte können in natürlicher Sprache durch bekannte oder neukonstruierte Worte, Umschreibungen, oder Entwicklung neuer Worte aus alten beschrieben werden. Eine Entwicklung von (zusammengesetzten) Piktogrammen kann als Erweiterung dieser Möglichkeiten angesehen werden. Der Abstraktionsgrad eines Piktogrammes hängt von seiner Verwendung ab. Während Piktogramme für Objekte sich konkreter auf das denotierte Objekt beziehen, sind Prozeßpiktogramme (vergleichbar mit Verben) meist abstrakter [Korfhage & Korfhage 87]. Das Fehlen visueller Verben wird von [Korfhage & Korfhage 87] als ein Punkt betrachtet, wo in der Ausdrucksstärke Differenzen auftreten.

Ein Satz wird von Chang als räumliche Anordnung von Piktogrammen angesehen. Eine (visuelle) Sprache wird als eine Menge von Sätzen, die bzgl. einer vogegebenen Syntax und Semantik definiert sind, definiert. Die syntaktische Analyse eines Satzes (spatial parsing [Lakin 87]) kann nach [Selker & Koved 88] folgende syntaktische Kategorien unterscheiden:

- positionale Syntax
    - relativ
        - sequentiell
        - metrisch
        - gemäß einer Orientierung
    - interagierend
        - eingebettet
        - sich überschneidend
    - bezeichnet
        - verbunden
        - benannt
- Größe
- zeitliche Abfolge
- algorithmische Beschreibung (z.B. Einschränkungen)
- regelorientierte Beschreibung (z.B. Zeilenumbruch)

Die beiden letzten Punkte zeigen, daß der Kreis hier sehr weit gezogen werden muß, um alle „Effekte" zu beschreiben. Der Bereich der Visuellen Sprachen und Visuellen Programmiersprachen ist noch nicht ausreichend erforscht. Vorgestellte Systeme sind nur für eingeschränkte Aufgaben einsetzbar.

6.3 Generelle Problematik in visuellen Programmsystemen für allgemeine Anwendungen

Von einigen Autoren wird die Meinung vertreten, daß durch eine Verwendung einer Visuellen Programmierung die Programmvisualisierung als Nebenprodukt des Programmentwicklungsprozesses abfällt. Ohne jetzt in die Details eines Systems einzudringen, sei hier ein Beispiel aus [Hsia & Ambler 88] diskutiert. Der erste Schritt zur Erstellung eines Programmes sei die Festlegung von graphischen Datenstrukturen.[1] Als Beispiel wird ein Programm vorgestellt, das einen Algorithmus zur Simulation des Problems der Stabilen Heirat realisiert. Durch Kopieren von Grundfiguren könnten die in Abbildung 6.1 aufgeführten oberen Präferenzlisten erstellt werden. Abläufe des Algorithmus werden durch Manipulationen der graphischen Darstellung und Interaktion mit Dialogen zur Integration von textuellen Darstellungen (z.B. Bedingungen) implementiert. Ein Objekt „Frau" wird durch einen Kreis realisiert. Die Attribute des Objektes „Frau" (verheiratet vs. ledig, geschieden vs. nicht geschieden) werden in Beziehung gesetzt mit den graphischen Attributen der Darstellung (siehe Abbildung). Auf diese Attribute wird innerhalb von textuell dargestellten Bedingungen Bezug genommen (Attribut „contour" = „fat").

[1] Die Probleme, die sich aus einer zweidimensionalen Darstellung ergeben, seien hier einmal vernachlässigt.

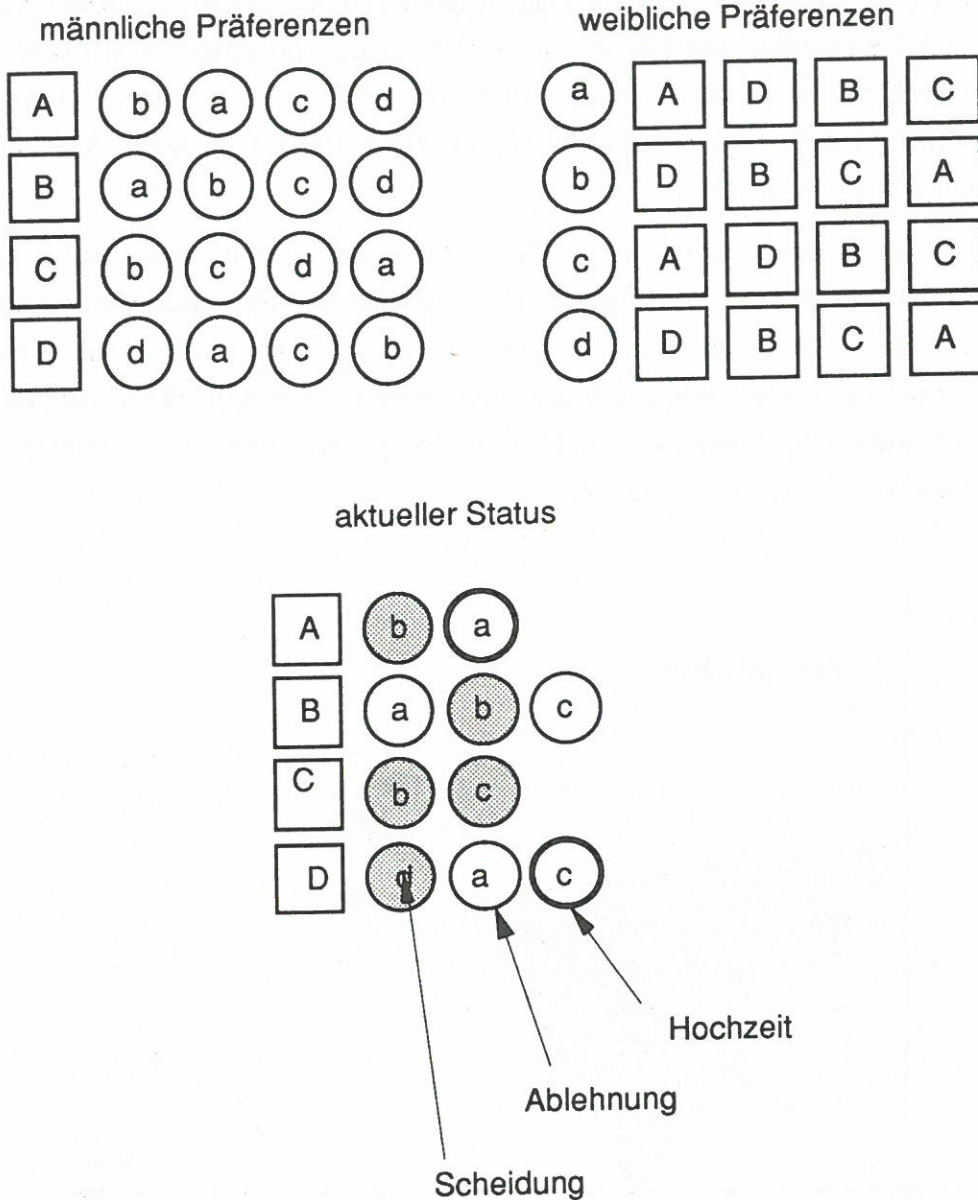


Abbildung 6.1: Schnappschuß der Zuordnung im Algorithmus „Stabile Heirat" (nach [Hsia & Ambler 88]).

Hier zeigt sich eine konzeptionelle Schwäche von vielen visuellen Programmiersystemen. Domänenspezifische Konzepte wie z.B. „Männlich", „Verheiratet-Sein" und „Geschieden-Sein" werden durch Attribute von graphischen Objekten codiert (Kontur, Textur, Zeichenstiftattribute). Da die Codierung willkürlich ist, kann nicht von einer problemorientierten Programmierung im klassischen Sinne gesprochen werden. Das Grundproblem der visuellen Programmierung ist, daß die syntaktischen Einheiten sich nicht so einfach editieren lassen wie bei textueller Programmierung. Es wird also notwendig sein, vorgefertigte Formen zu verwenden, die

das Konzept, für das sie stehen, nicht adäquat repräsentieren. Dieses gilt insbesondere, da bei einer Programmierung allgemeiner Anwendungen im größeren Stil sehr viele syntaktische Einheiten benötigt werden. Die bisherigen Werkzeuge zur Erstellung von Piktogrammen i.w.S. sind nicht ausreichend. Hier müssen bessere Modelle und (Interaktions-)Techniken entwickelt werden.

Diese Überlegungen wurde von Shu [Shu 87] in einer mehrdimensionalen Einteilung klassifiziert (Abbildung 6.2). Folgende Dimensionen zur Einteilung von visuellen Sprachen werden vorgeschlagen: Adäquatheit der Visualisierung (Visual Extent), Sprachniveau, d.h. Adäquatheit der Ausdrucksformen für Konzepte (Language Level), und Sprachanwendungsbreite (Scope), also die Adäquatheit, Objekte in verschiedenen Anwendungsbereichen zu repräsentieren.

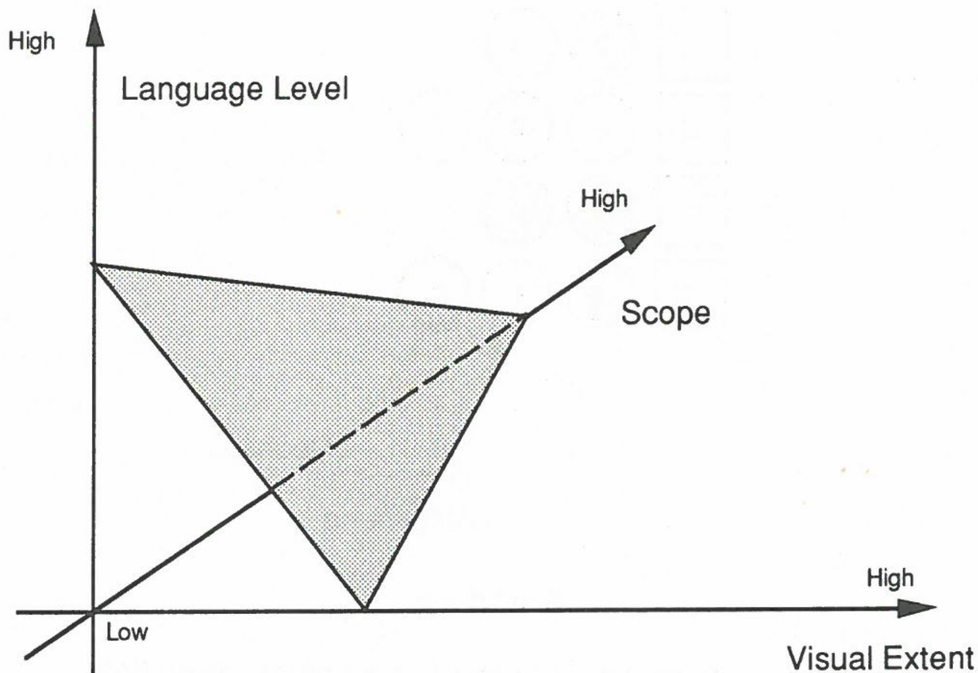


Abbildung 6.2: Dimensionen von Visuellen Programmiersprachen

Es scheint der Fall zu sein, daß Maximierung zweier Dimensionen eine Minimierung der anderen Dimension zur Folge hat. Die meisten Abwägungen sehen bei visuellen Darstellungen einen Verzicht auf einen umfassenden Gültigkeitsbereich vor, so daß eine visuelle Programmierung nur für dedizierte Systeme möglich ist oder durch textuelle Ergänzungen versehen werden muß. Eine Gewinnung einer (konzeptuellen) Visualisierung aus einer Visuellen Programmierung, wie sie von [Hsia & Ambler 88] vorgeschlagen wird, muß sich ebenfalls an diesen Abwägungen orientieren.

7. Zusammenfassung, Erweiterungen und Einordnung in die Forschungslandschaft

Visualisierungen, die sich an den Konzepten der Anwendungsdomänen orientieren, müssen „von Hand“ erstellt werden. Eine Darstellung muß bei Überschreiten einer bestimmten „Größe“ einhergehen mit einer Integration von Kontroll- und Fokussierungsmöglichkeiten (Kapitel 3.6.4). Weiterhin muß eine Interaktion mit den dargestellten Objekten unterstützt werden. Hieraus folgt, daß ein Visualisierungssystem mit einem System zur Erstellung von Benutzerschnittstellen zu vergleichen ist. Kapitel 2.6 enthielt eine Zusammenstellung der Anforderungen an ein System zur Erstellung von Benutzerschnittstellen (siehe auch: [Herczeg 89]).

In dieser Arbeit werden Bausteine zur Erstellung von Visualisierungen sowie allgemeine Konstruktionsprinzipien zur Kopplung von Anwendungs- und Visualisierungskomponenten vorgestellt und mit bestehenden Systemen und Modellen verglichen (Kapitel 4 und 5.6). Weiterhin werden Beschreibungsmöglichkeiten für die Anordnung der Bausteine präsentiert. Folgende Aufstellung faßt noch einmal die wesentlichen Punkte zusammen:

1. Ein Kernpunkt liegt in der Erstellung eines *universellen Systems zur Erstellung und Verwaltung beliebiger zweidimensionaler graphischer Objekte.* Graphische Objekte werden (prozedural) durch eine Zeichenfunktion definiert. Wann und unter welchen Bedingung 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 einem 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).
2. Die Eigenschaften von *selbstdefinierten Sichtbereichselementen* kann man durch Verwendung vordefinierter (Super-)Klassen geeignet zusammenstellen. Es existieren

Klassen, die Sichtbereichselemente maussensitiv, beweglich oder markierbar machen. Das Verwaltungssystem evaluiert bei Bedarf generische Zeichen- und Löschfunktionen für aktuell sichtbare Elemente. In prototypischen Anwendungen ist die Angabe einer Zeichenmethode schon ausreichend. Eine spezielle Löschfunktion wurde in Abschnitt 3.6.3 vorgestellt. Durch die objektorientierte Architektur der Sichtbereichsverwaltung lassen sich auch noch andere Verwaltungsfunktionen erweitern (z.B. Aussehen eines Elements bei einer Verschiebung) und auf spezielle Sichtbereichselemente zuschneiden. Diese Grundidee wurde in die Architektur der Common Lisp Programmierumgebung Allegro eingebettet. Sichtbereiche stellen in ihrer Funktionalität vergleichbare Komponenten wie Schaltflächen, editierbare Textfelder oder Tabellen und Matrizen dar. Letztere Komponenten werden durch die Macintosh „Toolbox“ unterstützt.

3. Bei einer Verwendung der von der Common Lisp Umgebung zur Verfügung gestellten, objektorientierten Erweiterung (ObjectLisp) wäre nur eine inhomogene Architektur möglich gewesen. Da nun die Sichtbereichsfunktionen in CLOS erstellt werden sollten, mußte auch eine *CLOS-Schnittstelle zur „Toolbox“* geschaffen werden. So entstand ein Schnittstelle „aus einem Guß“, die sich auf dem Baukastenkastenniveau (siehe Kapitel 2.6) befindet, aber im Bereich der Sichtbereiche wesentliche Erweiterungen enthält. Die entwickelten Konzepte lassen sich direkt zur Programmierung von Benutzerschnittstellen einsetzen, wobei der entscheidende Punkt darin liegt, daß sich die in dieser Arbeit vorgestellten objektorientierten Grundkomponenten erweitern und für dedizierte Anwendung anpassen lassen.

4. Eine weitere wesentliche Komponente dieser Arbeit bilden die *Beschreibungsformen für Anordnungen*. Positions- und Größenangaben von Bauelementen lassen sich deklarativ durch Layoutschemata bzgl. einer rechteckigen Darstellungsfläche festlegen. Dieses gilt für die beiden Ebenen der Dialog- und der Sichtbereichselemente. Für Sichtbereichselemente lassen sich auch lokale Anordnungsbeziehungen durch Referenzpunkte bestimmen. Selbstdefinierte Layoutbeschreibungen lassen sich nahtlos integrieren. Ein wesentliches Merkmal der Layoutalgorithmen ist, daß sie sich nicht nur für vordefinierte Dialog- oder Szenenelemente eignen. Durch Verwendung eines (abstrakten) Protokolls mit generischen Funktionen lassen sich die Anordnungsalgorithmen für beliebige Lisp-Objekte verwenden, sofern hierfür geeignete Methoden definiert werden.

5. Das Haupteinsatzgebiet der in dieser Arbeit vorgestellten Konzepte liegt in der Programmvisualisierung (vgl. Kapitel 2). Insbesondere für dieses Einsatzgebiet

wurden die *Kopplungswerkzeuge* entwickelt. Wie in der Arbeit ausgeführt, können sie aber auch eingesetzt werden, um eine strukturierte Ankopplung einer Benutzerschnittstelle zu ermöglichen.

Im Bereich der Erstellung von Benutzerschnittstellen leisten diese Werkzeuge gute Dienste. Man vergleiche die Variabilität des für CLOS erstellten Klassen- und Objektinspektors (Abbildungen 3.6 und 3.7) mit einem vergleichbaren Dialog des Goldworks-Systems (Abbildung 2.6). Durch Verwendung von Layoutbeschreibungen erhält man die Anpassung der Darstellung bei Vergrößerung eines Fensters geschenkt. Bezüge zu vergleichbaren Ansätzen stellte Kapitel 3.6.5 her.

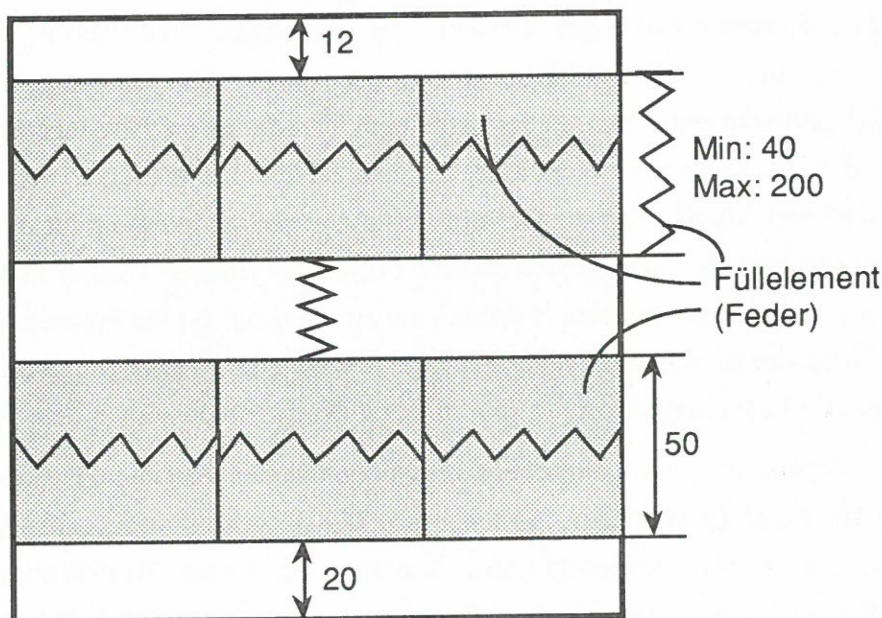


Abbildung 7.1: Grobe Skizze einer visuellen Definition von Anordnungschemata. Boxen sind durch Rechtecke repräsentiert, Füllelemente durch stilisierte Federn.

Für den Bereich der Visuellen Programmierung (Kapitel 6) böte sich eine Anwendung der in dieser Arbeit vorgestellten Bausteine u.a. in der Erstellung von visuellen Layoutbeschreibungen an. Systeme dieser Art existieren als Resource Editoren oder „Interface Builder“. Als Erweiterungen müßte eine visuelle Repräsentation des Federmodells entwickelt werden. Abbildung 7.1 zeigt eine Skizze. Zu beachten ist jedoch, daß Terme zur Definition von Maximal- und Minimalausdehnungen i.a. nicht die einfache Gestalt aus Abbildung 7.1 haben, sondern oft durch recht aufwendige Berechnungen ermittelt werden. Nichtsdestotrotz könnten durch Verwendung

vordefinierter Bausteine inhärent visuelle Teilaspekte auch visuell repräsentiert werden. Die in dieser Arbeit erstellten Systeme bilden eine Grundlage zur Erstellung von visuellen (Teil-)Programmsystemen.

Im Bereich der Zeichenfunktionen könnte man sog. graphische Kontexte, wie sie aus dem CLX-Standard [X-Windows] bekannt sind, einführen. Aus dem CLUE-System (Common Lisp User Interface Environment) könnte man eine Initialisierung von Interaktionsobjekten (Dialogfenster, Schaltflächen, ...) durch sog. Resourcen übernehmen. Anwendungs-„Programme" wären durch Verwendung von Resource-Editoren durch die Benutzer (in gewissem Rahmen) adaptierbar.

Eine Bereitstellung weiterer zweidimensionaler Layoutbeschreibungen zusammen mit einer Erweiterung der Bibliothek für Anzeigegeräte könnte ohne Änderung des bestehenden Systems integriert werden. Als wichtigste Erweiterung wäre eine Beschreibung der vertikalen Überlagerungsstruktur der (zweidimensionalen) Sichtbereichelemente zu nennen. In der bisherigen Version ist die Überlagerungsstruktur nicht determiniert. Es ist noch nicht klar, wie eine Beschreibung hierfür aussehen kann. Als Möglichkeiten könnte man die Definition einer partiellen Ordnung für die Elemente einer Ebene vorsehen. Ebenen sollten wie Folien überlagert werden können. Eine Integration von dreidimensionalen Beschreibungen – sowohl für die Formen als auch für die Anordnung der dreidimensionalen Objekte – wäre eine Erweiterung von größerem Umfang. Hier müßten komplexere Beschreibungsformen entwickelt werden.

Durch Verwendung von Layoutbeschreibungen tritt der Vorgang des Anordnens der Elemente (in einer Box) in den Hintergrund. Die Beschreibungen sind deklarativ. Inwieweit die Anordnungsschemata als Primitive zur Repräsentation von Anordnungwissen 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. welche Erweiterungen nötig sind.

Literatur

Abelson & Sussman 85: *Structure and Interpretation of Computer Programs*, H. Abelson, G. J. Sussman, MIT Press, 1985.

Anderson 88: *Kognitive Psychologie - Eine Einführung*, J. R. Anderson, Spektrum-der-Wissenschaft-Verlagsgesellschaft, 1988.

Ballard & Brown 82: *Computer Vision*, D. H. Ballard, C. M. Brown, Prentice Hall Inc., Englewood Cliffs, New Jersey.

Bergmann & Gerlach 87: *Quirk - Implementierung einer TBox zur Repräsentation begrifflichen Wissens*, H. Bergmann, M. Gerlach, Universität Hamburg, WISBER Memo 11, Juni 1987.

Blaschek et al. 87: *Einführung in die Programmierung mit Modula-2*, G. Blaschek, G. Pomberger, F. Ritzinger, Springer Verlag, 1987.

Bobrow & Stefik 83: *The LOOPS Manual*, D. G. Bobrow, M. Stefik, Xerox Corporation, Dezember 1983.

Bobrow & Kiczales 88: *The Common Lisp Object System Metaobject Kernel*, D. G. Bobrow, G. Kiczales, Preprint, erschienen in: 1988 ACM Conference on LISP and Functional Programming.

Bobrow & Winograd 77: *A Knowledge Representation Language*, D. G. Bobrow, T. Winograd, Cognitive Science, 1, 1977, pp. 3-45.

Borning 81: *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, A. Borning, ACM Trans. Programming Languages and Systems, 3, 4, Oct. 1981, pp. 353-387.

Bothner 88: *Efficiently Combining Logical Constraints with Functions*, P. M. A. Bothner, Dissertation, Stanford University, Department of Computer Science, Report No. STAN-CS-88,1238.

Brachman & Schmolze 85: *An Overview of the KL-ONE Knowledge Representation System*, R. J. Brachman, J. G. Schmolze, Cognitive Science, 9, 1985, pp. 171-216.

Bromsley & Lamson 87: *Lisp Lore: a Guide to Programming the Lisp-Machine*, H. Bromsley, R. Lamson, 2. Ausgabe, Kluwer, 1987.

Brown et al. 85: *Program Visualization: Graphical Support for Software Development*, G. P. Brown, R. T. Carling, C. F. Herot, D. A. Kramlich, P. Souza, IEEE Computer, 18, 8, August 1985, pp. 27-35.

Brown 88: *Algorithm Animation*, M. H. Brown, ACM Distinguished Dissertations Series, MIT Press, 1988.

Brown & Sedgewick 84: *A System for Algorithm Animation*, M. H. Brown, R. Sedgewick, Computer Graphics, 18, 3, July 1984, pp. 177-186.

Brown & Sedgewick 85: *Techniques for Algorithm Animation*, M. H. Brown, R. Sedgewick, IEEE-Software, 2, 1, Januar 1985, pp. 28-39.

Cattaneo et al. 86: *Iconlisp: an Example of a Visual Programming Language*, G. Cattaneo, A. Guercio, S. Levialdi, G. Tortora, in: Visual Languages Conference Proceedings 1986, pp. 22-25.

Chang et al. 86: *Visual Languages*, S. K. Chang, T. Ichikawa, P. A. Ligomides (Hrsg.), Plenum Press, New York, London, 1986.

Chang 87: *Visual Languages: A Tutorial and Survey*, S. K. Chang, IEEE Sofware, Januar 1987, pp. 29-39.

Christaller et al. 89: *Die KI-Werkbank Babylon*, T. C. Christaller, F. diPrimio, A. Voss (Hrsg.), Addison-Wesley, 1989.

Dodani et al. 89: *Seperation of Powers*, M. H. Dodani, C. H. Hughes, J. M. Moshell, BYTE, März, 1989.

Drescher: *ObjectLISP User Manual*, G. Drescher, LMI, 1000 Massachussets Avenue, Cambridge, MA 02138.

Eisenstadt & Brayshaw 88: *The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming*, M. Eisenstadt, M. Brayshaw, Journal of Logical Programming, 5, 1988, pp. 277-342.

Glinert & Tanimoto 84: *Pict: An Interactive Graphical Programming Environment*, E. P. Glinert, S. T. Tanimoto, IEEE Computer, 17, 11, November 1984, pp. 7-25.

Goldberg & Robson 83: *Smalltalk-80 - The Language and its Implementation*, A. Goldberg, D. Robson, Addison Wesley, 1983.

Goos 80: *The Programming Language ADA - Reference Manual*, G. Goos, J. Hartmanis (Hrsg.), Springer Verlag, Reihe Lecture Notes in Computer Science, 1980.

Haarslev & Möller 88a: *Visualisierung und Animation in der experimentellen Bildauswertung*, V. Haarslev, R. Möller, in: Proceedings, GI/ÖCG-Fachgespräch Visualisierungstechniken und Algorithmen, Wien, 26.-27. September 1988, W. Barth (Hrsg.), Informatik Fachberichte Nr. 182, Springer Verlag, Berlin, 1988, pp. 213-223.

Haarslev & Möller 88b: *Eine graphische Umgebung zur experimentellen Bildverarbeitung*, V. Haarslev, R. Möller, in: Proceedings, 10. DAGM-Symposium Zürich, 27.-29. September 1988, H. Bunke, O. Kübler, P. Stucki (Hrsg.), Mustererkennung, Informatik Fachberichte Nr. 180, Springer Verlag, Berlin, 1988, pp. 319-325.

Haarslev & Möller 88c: *Visualization of Experimental Systems*, V. Haarslev, R. Möller, in: Proceedings, 1988 IEEE Workshop on Visual Languages, Pittsburgh/PA, Oct. 10.-12. 1988, IEEE Computer Society Press, 1988, pp. 175-182.

Haarslev & Möller 89: *VIPEX: Visual Programming of Experimental Systems*, V. Haarslev, R. Möller, erscheint in: Visual Languages and Visual Programming, S. K. Chang (Hrsg.), Plenum Press, New York and London, 1989.

Habel 87: *Repräsentation räumlichen Wissens*, C. Habel, Fachbereich Informatik, November 1987, Mitteilung Nr. 153, FBI-HH-M-153/87.

Haeberli 88: *ConMan: a Visual Programming Language for Interactive Graphics*, P. E. Haeberli, ACM SigGraph Computer Graphics, 22, 4, August 1988, pp. 103-111.

Hayes & Baran 89: *A Guide to GUIs*, F. Hayes, N. Baran, BYTE, Juli 1989, pp. 250-257.

Hennessy 89: *COMMON LISP*, W. L. Hennessy, McGraw-Hill, 1989.

Herczeg 86: *INFORM-Manual: Clusters Version 1.9*, M. Herczeg, Universität Stuttgart, Institut für Informatik, Forschungsgruppe INFORM, Verbundprojekt WISDOM.

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

Hoffmann 87: *SMALLTALK verstehen und anwenden*, H. J. Hoffmann, Hanser Verlag, 1987.

Hsia & Ambler 88: *Programming Through Pictorial Transformations*, Y.-T. Hsia, A. L. Ambler, in: Proceedings 1988 International Conference on Computer Languages, 3.-9. Oktober 1988, Miami Beach, Florida, Computer Society Press.

Jacob 85: *A State Transition Diagram Language for Visual Programming*, R. J. Jacob, IEEE Computer, 18, 8, August 1985, pp. 51-59.

Keene 89: *Object-Oriented Programming in CLOS - A Programmer's Guide to CLOS*, S. Keene, Addison-Wesley, 1989.

Kimbrough & LaMotte 89: *Common Lisp User Interface Environment*, K. Kimbrough, O. LaMotte, Preprint, Texas Instruments Inc., July 1989.

Kindermann & Quantz 88: *Graphik-orientierte Wissensrepräsentation*, C. Kindermann, J. Quantz, in: *Modellierung graphischer Dialoge für interaktive Entwicklungsprozesse*, C. Hoffman, J. Röhrich (Hrsg.), GMD-Studien Nr. 151, August 88, pp. 102-132.

Knuth 79: *TEX and Metafont – New Directions in Typesetting*, D. E. Knuth, Digital Press, 1979.

Krasner & Pope: *A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80*, G. E. Krasner, S. T. Pope, ParcPlace Systems, Smalltalk Manual, Palo Alto, Georg Heeg, Dortmund.

Korfhage & Korfhage 87 *Criteria for Visual Languages*, M. A. Korfhage, R. R. Korfhage, in: [Chang et al. 87], pp. 207-231.

Lakin 86: *Spatial Parsing for Visual Languages*, F. Lakin, in: [Chang et al. 87], pp. 35-85.

Lakin 87: *Visual Grammars for Visual Languages*, F. Lakin, AAAI '87 Proceedings, Bd.2, pp. 683-688, 1987.

Langenscheidt 60: *Taschenwörterbuch der englischen und deutschen Sprache*, Langenscheidt Verlagsbuchhandlung, 20. Auflage 1960.

Lieberman 85: *There's More to Menu Systems Than Meets the Screen*, H. Lieberman, ACM SigGraph, Proceedings, 19, 3, 1985, pp. 181-189.

Linton et al. 89: *Composing User Interfaces with InterViews*, M. A. Linton, J. M. Vlissides, P. R. Calder, IEEE Computer, 22, 2, Februar 1989, pp. 8-22.

London & Duisberg 84: *Animating Programs using Smalltalk*, Tech. Report, CR-84-30, Computer Research Lab., Tektronix, Inc., Beaverton, OR, Dec. 1984, auch in: IEEE Computer, 18, 8, August 1985, pp. 61-71.

Matwin & Pietrzykowski 85: *Prograph: A Preliminary Report*, S. Matwin, T. Pietrzykowski, Computer Languages, 10, 2, 1985, pp. 91-126.

McCormick et al 87: *Special Issue on Visualization in Scientific Computing*, B. H. McCormick, T. A. DeFanti, M. D. Brown (Hrsg.), Computer Graphics, 21, 6, November 1987.

Möller 88: *Gestaltung und Implementation einer graphischen Dialogschnittstelle auf einer Lisp-Maschine nach dem Vorbild eines datenfluß- und objektorientierten Bildfolgenanalysesystems*, R. Möller, Universität Hamburg, Fachbereich Informatik, Mitteilung Nr. 163/88, FBI-HH-M-163/88.

Moriconi & Hare 85: *Visualizing Program Designs Through PegaSys*, M. Moriconi, D. F. Hare, IEEE Computer, 18, 8, August 1985, pp. 72-85.

Mundie 88: *Interacting with the Tiny and the Immense*, C. Mundie, BYTE, April, 1989, pp. 279-288.

Myers 80: *Displaying Data Structures for Interactive Debugging*, B. A. Myers, Tech. Report CSL-80-7, Xerox Palo Alto Research Center, Palo Alto, CA, Juni 1980.

Myers 83: *Incense: A System for Displaying Data Structures*, B. A. Myers, Computer Graphics, 7, July 1983, pp. 115-125.

Myers 86a: *Visual Programming, Programming by Example and Program Visualization: A Taxonomy*, B. A. Myers, in: CHI'86 Proceedings, Computer-Human Interaction, 1986, pp. 59-66.

Myers 86b: *Creating Highly-Interactive and Graphical User Interfaces by Demonstration*, B. A. Myers, W. Buxton, ACM-SIGGRAPH Computer Graphics, 20 , 4, 1986, pp. 249-258.

Myers 88: *The State of the Art in Visual Programming and Program Visualization*, B. A. Myers, Carnegie Mellon University, Computer Science Department, Februar 1988, CMU-CS-88-114.

Myers 89a: *User-Interface Tools: Introduction and Survey*, B. A. Myers, IEEE Software, Januar, 1989, pp. 15-23.

Myers 89b: *Encapsulating Interactive Behaviors*, B. A. Myers, in: Proceedings, Human Factors in Computing Systems, K. Bice, C. Lewis (Hrsg.), Austin, Texas, 30. April - 4. May, 1989, pp. 319-324.

Nelson 85: *Juno, a Constraint-based Graphics System*, G. Nelson, in: ACM SigGraph, Proceedings, 19, 3, 1985, pp. 235-243.

Neumann & Novak 84: *Szenenbeschreibung und Imagination in NAOS*, B. Neumann, H.-J. Novak, Universität Hamburg, Fachbereich Informatik, Mitteilung Nr. 123.

Newbery 88: *EDGE: An Extendible Directed Graph Editor*, F. J. Newbery, Universität Karlsruhe, Fakultät für Informatik, Interner Bericht Nr. 8/88.

Sterling & Shapiro 88: *Prolog - Fortgeschrittene Programmiertechniken*, Addison-Wesley, 1988.

Pribbenow 88: *Verträglichkeitsprüfungen für die Verarbeitung räumlichen Wissens*, S. Pribbenow, in: Proceedings GWAI 88, W. Höppner (Hrsg.), Springer-Verlag, 1988, pp. 226-235.

Raeder 85: *A Survey of Current Graphical Programming Techniques*, G. Raeder, IEEE Computer, 18, 8, August 1985, pp. 11-25.

Reiss 86: *Displaying Program and Data Structures*, S. P. Reiss, Tech. Report CS-86-19, Brown University, Providence, RI, April 1986.

Reiss 87a: *Working in the Garden Environment for Conceptual Programming*, S. P. Reiss, IEEE Software, November 1987, pp. 16-27.

Reiss 87b: *An Object Oriented Framework for Conceptual Programming*, S. P. Reiss, in: *Research Directions in Object-Oriented-Programming*, B. Shriver, P. Wegner (Hrsg.), MIT Press 1987, Computer Systems Series, pp. 189-218.

Rickert 86: *Werkzeuge und Systeme zur Unterstützung des Erwerbs und der objektorientierten Modellierung von Wissen*, W. F. J. Rickert, Dissertation, Universität Stuttgart, Institut für Informatik, 1986.

Roberts & Goldstein 77: *The FRL Manual*, R. E. Roberts, I. P. Goldstein, AI Memo 409 Edition, MIT Lab., 1977.

Rowe et al. 86: *A Browser for Directed Graphs*, L. A. Rowe, M. Davis, E. Messinger, C. Meyer, C. Spirakis, A. Tuan, Report No. UCB/CSD 86/292, April 86, Computer Science Division (EECS) University of California, Berkeley, California 94720.

Schmucker 86: *Object-oriented Programming for the Macintosh*, K. Schmucker, Hayden Book Company, 1986.

Selker & Koved 88: *Elements of Visual Language*, T. Selker, L. Koved, Research Report, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, N. Y. 10598, RC 13929 (#62613) 8/17/88, 1988.

Shu 87: *Visual Prgogamming Languages, A Perspective and a Dimensional Analysis*, N. C. Shu, in: [Chang et al. 87], pp. 11-34.

Smith et al. 82: *Designing the Star User Interface*, D. C. Smith, C. Irby, R. Kimball, B. Verplank, BYTE, No. 4, 1982, pp. 242-282.

Smith 86: *The Alternate Reality Kit - An Integrated Environment for Creating Interactive Simulations*, R. B. Smith, in: Conference Proceedings Visual Languages 1986, pp. 99-106.

Steele 84: *Common Lisp - The Language*, G.L. Steele jr., Digital Press, 1984.

Stenger 86: *VISIVAL – Graphische Visualisierung numerischer Daten durch Anzeigeinstrumente auf dem Bildschirm*, Universität Stuttgart, Forschungsgruppe INFORM, Verbundprojekt WISDOM, FB-INF-87-07, Dezember 1986.

Stoyan 88: *Programmiermethoden der Künstlichen Intelligenz*, H. Stoyan, Band 1, Studienreihe Informatik, Springer-Verlag, 1988.

Stoyan & Görz 84: *LISP - Eine Einführung in die Programmierung*, Springer Verlag, 1984.

Strobel 89: *TRED: ein Trajektorieneditor mit dreidimensionaler Visualisierungskomponente*, K. Strobel, Diplomarbeit, Universität Hamburg, Januar 1989.

Stroustrup 87: *The C++ Programming Language*, B. Stroustrup, Addison-Wesley, 1987.

Symbolics 88a: Handbücher zur Symbolics-Programmierumgebung, Symbolics Inc., 1988.

Symbolics 88b: Handbücher zur Symbolics-Programmierumgebung, 7A Programming the User Interface – Concepts, Symbolics Inc., 1988.

Vander Zanden 89: *Constraint Grammars - A New Model for Specifying Graphical Applications*, B. T. Vander Zanden, in: CHI'89, Human Factors in Computing Systems, K. Bice, C. Lewis (Hrsg.), 1989, pp. 325-330.

Wahrig 80: *Deutsches Wörterbuch*, G. Wahrig, Mosaik Verlag, 1980.

Winston & Horn 89: *LISP*, P. H. Winston, B. K. P. Horn, 3. Ausgabe, Addison-Wesley, 1989.

Wright et al. 88: *pluribus: A Visual Programming Environment for Education and Research*, S. Wright, W. Feuerzeig, J. Richards, in: 1988 IEEE Workshop on Languages for Automation, Symbiotic and Intelligent Robotics, Univ. of Maryland, College Park, Maryland, Aug. 29-31, 1988, IEEE Soc. Press, pp. 122-127.

X3J13: *Common Lisp Object System Specification*, X3J13 Document 88-002R, D. Bobrow et al., Association for Computing Machinery, ISBN 0-89791-289-6, June 1988.

X-Windows: *CLX - Common Lisp Language X Interface*, Preprint, Texas Instruments Inc., 1989.

Young et al. 88: *Software Environment Architectures and User Interface Facilities*, M. Young, R. N. Taylor, D. B. Troup, IEEE Transactions on Software Engineering, 14, 6, Juni 1988, pp. 697-707.

Anhang A

Allegro Common Lisp und ObjectLisp

Das erstellte CLOS-System zur Schnittstellenprogrammierung verwendet in einigen Teilen das ObjectLisp-System als Implementierungsgrundlage. Dabei ist entscheidend, daß die Semantik des ObjectLisp-Systems aufgrund einiger Schwächen gerade nicht durch CLOS-Metaklassen simuliert werden sollte. Das Ziel der Integration von CLOS in das Allegro-System ist es, die gebotene Funktionalität der ObjectLisp-Klassen zu erhalten, diese ggf. zu erweitern sowie hierauf aufbauend höhere Abstraktionsebenen zu errichten.

Es ist nun nicht so ohne weiteres möglich, die ObjectLisp-Objekte einfach über Bord zu werfen und von Grund auf, d.h. unter Verwendung der angebotenen Toolbox-Schnittstelle, eine Klassenhierarchie zu konzipieren. Das liegt daran, daß die ObjectLisp-Objekte mit dem Laufzeitsystem der Allegro-Implementierung verzahnt sind. Zur Illustration sei ein einfaches Beispiel aus der Fensterverwaltung genannt. Soll z.B. das Lisp-System verlassen werden, so wird für jedes noch offene Fenster die generische Funktion `ccl:window-close` evaluiert. Für jedes selbstdefinierte Fensterobjekt kann diese Funktion evt. spezialisiert sein. Da diese Funktion durch das Laufzeitsystem ausgewertet wird, kann niemals verhindert werden, daß eine zum ObjectLisp-System gehörige Funktion evaluiert wird, da kein Zugang zur Implementierung des Laufzeitsystems besteht. Wie läßt sich erreichen, daß auch für vom Laufzeitsystem evaluierte generische Funktionen Spezialisierungen in CLOS erstellt werden können?

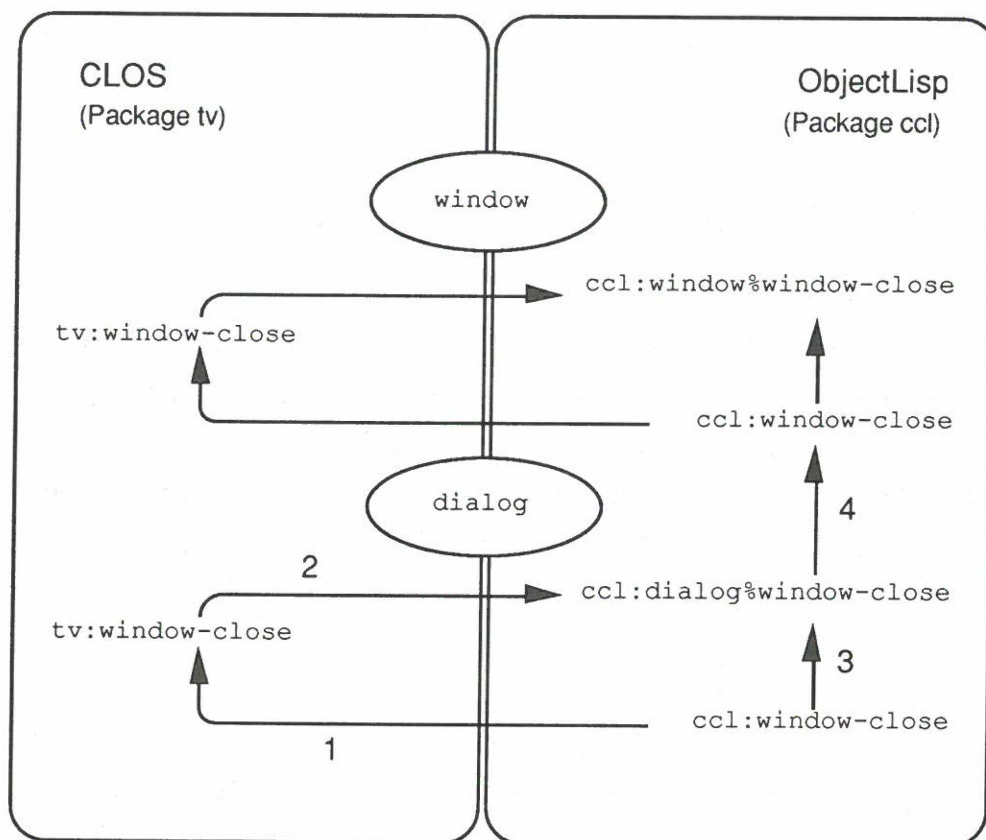


Abbildung A.1: Kombination von Methoden.

Das generelle Prinzip der Einflechtung der CLOS-Methoden in das ObjectLisp-System läßt sich an einem Beispiel veranschaulichen. Für jede exportierte ObjectLisp-Klasse wurde ein CLOS-Äquivalent geschaffen. Instanzen der CLOS-Klassen sind dabei als Hüllen über die dahinterstehenden ObjectLisp-Instanzen aufzufassen. Diese wiederum werden jeweils mit einem Verweis auf ihr CLOS-Pendant versehen. Ebenso ist für jede (exportierte) ObjectLisp-Methode eine entsprechende CLOS-Variante vorgesehen. In Abbildung A.1 sei dies für einen kleinen Ausschnitt der Klassenhierarchie gezeigt.

Die Funktion `ccl:window-close` ist sowohl für die Klasse `window` als auch für deren Spezialisierung `dialog` definiert, wobei zu beachten ist, daß die Methode der Klasse `dialog` die übergeordnete evt. zusätzlich evaluieren kann. Jede vorherige Methode einer ObjectLisp-Klasse wurde mit einem Präfix umbenannt, der sich aus dem Klassennamen ergibt. Im Beispiel aus Abbildung 1 wurde also aus dem ursprünglichen `ccl:window-close` ein `ccl:window%window-close` bzw. `ccl:dialog%window-close`. Nun wurde jede ObjectLisp-Methode folgendermaßen neu definiert:

– es wird überprüft, ob es sich bei dem Objekt, für das die Methode evaluiert wurde, um eine Instanz handelt, die als Implementierung einer CLOS-Instanz dient.

– Ist dieses der Fall, so wird für die zugehörige CLOS-Instanz die korrespondierende CLOS-Methode ausgewertet (`tv:window-close`, Pfeil 1 in Abbildung 1).

– Der Körper der CLOS-Methode dereferenziert wiederum die ObjectLisp-Instanz und ruft die eigentliche Methode des ObjectLisp-Systems mit dem Klassennamen als Präfix auf (Pfeil 2).

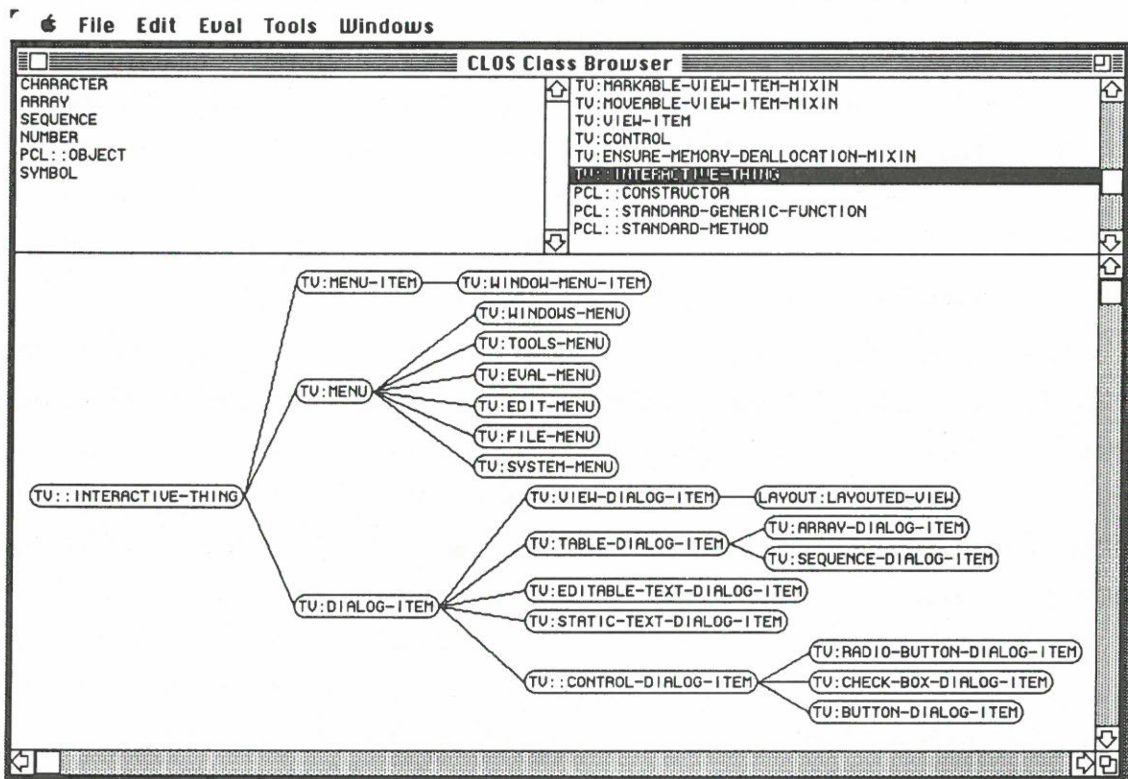
– Wenn es sich nicht um ein Objekt mit entsprechendem CLOS-Repräsentanten handelt, so evaluiert z.B. `ccl:window-close` der Klasse `dialog` ohne „Umweg" (Pfeil 3) die eigentliche Methode `ccl:dialog%window-close` (welche u.U. `ccl:window-close` der Klasse `window` evaluiert – Pfeil 4).

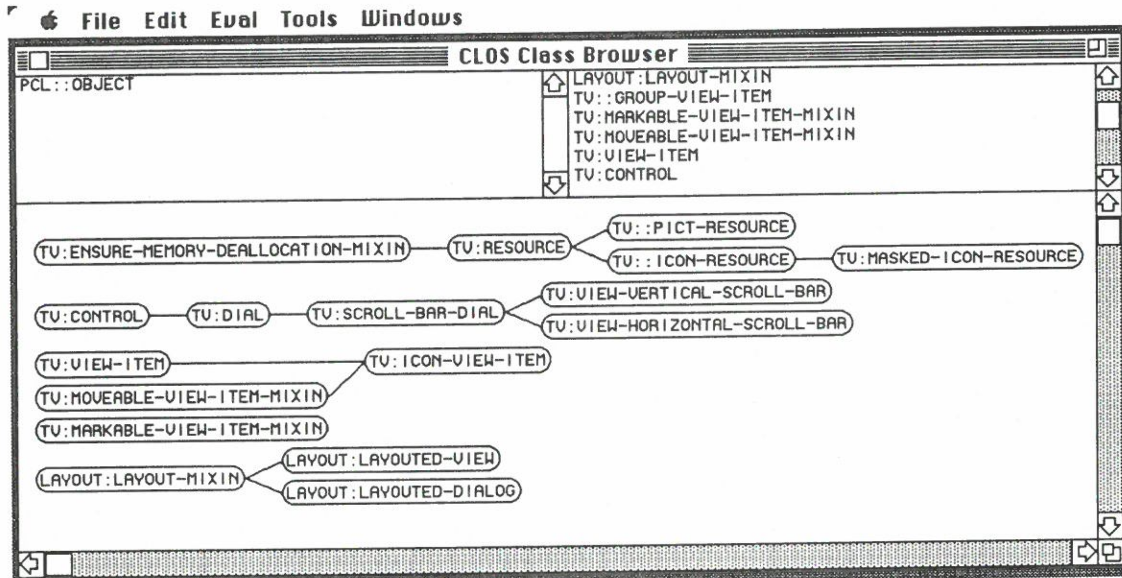
Wie schon erwähnt muß immer damit gerechnet werden, daß die in der Vererbungshierarchie nächsthöherliegende Methode direkt „aufgerufen" wird. Nimmt man nun an, dieses sei hier der Fall, so würde also durch `ccl:dialog%window-close` die Methode `ccl:window-close` der Klasse `window` evaluiert. Falls in dieser Situation noch einmal den „Umweg" über die entsprechende CLOS-Methode erfolgte, so würden ggf. definierte Dämonenmethoden der Methode `tv:window-close` ein zweites Mal evaluiert! Dieses Verhalten ist nicht erwünscht. Es darf nur einmal zur CLOS-Seite „gewechselt" werden. Es muß zur Laufzeit entschieden werden, ob die entsprechende Methode `tv:window-close` evaluiert wird oder nicht.

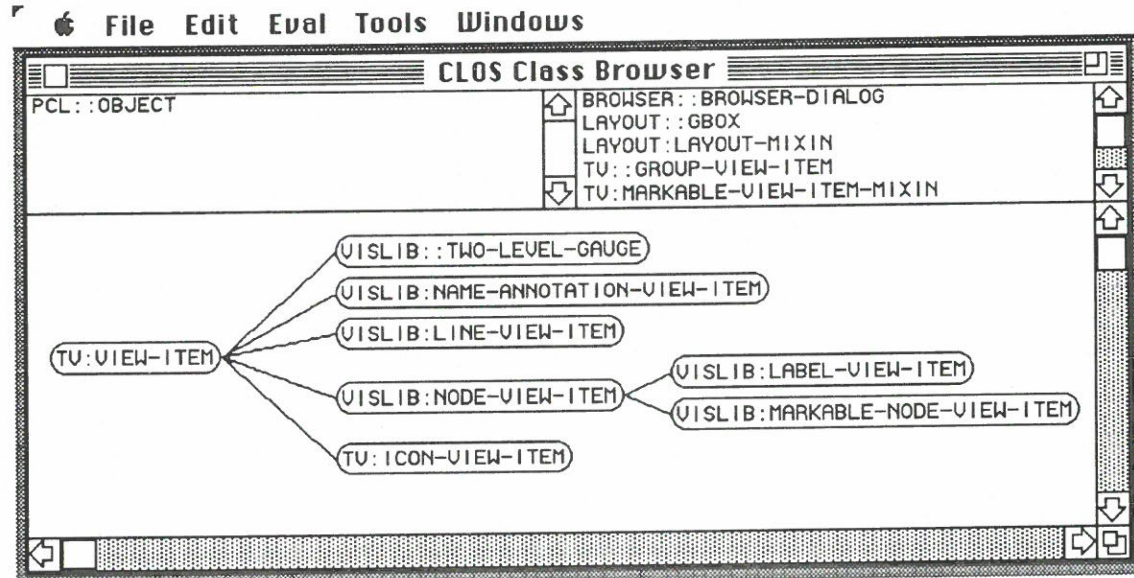
Es sollte deutlich werden, daß die hier erwähnten CLOS-Methoden aus der CLOS-Sicht nur Hüllen über den eigentlichen ObjectLisp-Methoden sind, während sie aus der ObjectLisp-Sicht in die Evaluierung der Methoden einer generischen Funktion jeweils an geeigneter Stelle eingeflochten werden. Wie schon angedeutet, können für die ehemaligen ObjectLisp-Methoden nun ohne weiteres Dämonen in CLOS definiert werden. Insofern stellt also die Umstellung auf CLOS-Syntax eine echte Erweiterung der Möglichkeiten des bestehenden Systems dar.

Anhang B

Überblick über die Klassenhierarchie




File Edit Eval Tools Windows
CLOS Class Browser
PCL::OBJECT
LAYOUT:LAYOUT-MIXIN
TV::GROUP-VIEW-ITEM
TV:MARKABLE-VIEW-ITEM-MIXIN
TV:MOVEABLE-VIEW-ITEM-MIXIN
TV:VIEW-ITEM
TV:CONTROL
TV:ENSURE-MEMORY-DEALLOCATION-MIXIN
TV:RESOURCE
TV::PICT-RESOURCE
TV::ICON-RESOURCE
TV:MASKED-ICON-RESOURCE
TV:CONTROL
TV:DIAL
TV:SCROLL-BAR-DIAL
TV:VIEW-VERTICAL-SCROLL-BAR
TV:VIEW-HORIZONTAL-SCROLL-BAR
TV:VIEW-ITEM
TV:ICON-VIEW-ITEM
TV:MOVEABLE-VIEW-ITEM-MIXIN
TV:MARKABLE-VIEW-ITEM-MIXIN
LAYOUT:LAYOUTED-VIEW
LAYOUT:LAYOUT-MIXIN
LAYOUT:LAYOUTED-DIALOG


File Edit Eval Tools Windows
CLOS Class Browser
PCL::OBJECT
BROWSER::BROWSER-DIALOG
LAYOUT::GBOX
LAYOUT:LAYOUT-MIXIN
TV::GROUP-VIEW-ITEM
TV:MARKABLE-VIEW-ITEM-MIXIN
VISLIB::TWO-LEVEL-GAUGE
VISLIB:NAME-ANNOTATION-VIEW-ITEM
VISLIB:LINE-VIEW-ITEM
TV:VIEW-ITEM
VISLIB:LABEL-VIEW-ITEM
VISLIB:NODE-VIEW-ITEM
VISLIB:MARKABLE-NODE-VIEW-ITEM
TV:ICON-VIEW-ITEM

Anhang C

Auflistung des Quellcodes einiger Beispiele

C.1. Quellcode des Dialogs zur Visualisierung einer Vererbungshierarchie aus Kapitel 3.6.4.

```
(defun class-name-as-string (class)
  (format nil "~S" (class-name class)))

(defclass class-table
  (tv:sequence-dialog-item)
  ((graph-view :accessor graph-view)
   (partner-table :accessor partner-table)
   (double-click-shift-direction
          :accessor double-click-shift-direction
          :initarg :double-click-shift-direction))
  (:documentation "Tabelle zur Darstellung von CLOS-Klassen."))


(defun make-class-table (initial-class-sequence shift-direction)
  "Erzeugung einer Klassentabelle mit best. Voreinstellungen."
  (tv:make-dialog-item
   :class 'class-table
   :table-sequence initial-class-sequence
   :double-click-shift-direction shift-direction
   :table-hscrollp nil
   :selection-type ':disjoint))

(defmethod tv:cell-contents ((item class-table)
                             h &optional (v nil))
  "Filterfuntion zur Bestimmung des Aussehens eines Tabelleneintrags.
In der Tabelle sind Klassenobjekte enthalten. Es sollen
jedoch nur deren Namen erscheinen."
  (let ((contents (call-next-method)))
    (if (not (equal contents ""))
      (format nil "~S" (class-name contents))
      contents)))


(defun class-browser (&optional (roots (list (find-class 't))))
  (macrolet ((tables (table1 table2)
                    `(:hbox ()
                            (:fbox () ,table1) 1
                            (:fbox () ,table2))))

    (let* ((supers-table (tv:make-dialog-item
                          :class 'class-table
                          :table-sequence roots
                          :selection-type ':disjoint
                          :double-click-shift-direction ':right
                          :table-hscrollp nil))
           (subs-table (tv:make-dialog-item
                          :class 'class-table
                          :table-sequence '()
                          :double-click-shift-direction ':left
                          :selection-type ':disjoint
```

```
                          :table-hscrollp nil))
           (class-browser-scene (layout:make-layouted-view
                                  :scroll-bars ':both
                                  :bordered-p nil
                                  :hsb-line-increment 30
                                  :vsb-line-increment 30
                                  :hsb-left-offset -1
                                  :hsb-right-offset -1
                                  :vsb-top-offset -1
                                  :vsb-bottom-offset -1
                                  :auto-scrolling t)))
      (setf (partner-table supers-table)
            subs-table)
      (setf (partner-table subs-table)
            supers-table)
      (setf (graph-view supers-table)
            class-browser-scene)
      (setf (graph-view subs-table)
            class-browser-scene)
      (layout:make-layouted-dialog
        :window-size #@(500 330)
        :window-type ':document-with-zoom
        :window-title "CLOS Class Browser"
        :window-font '(9 "Monaco")
        :default-button nil
        :layout (:vbox ()
                 (:hbox (:height 1/4)
                  (:vbox ()
                   (tables supers-table
                           subs-table))
                   1)
                  1
                  (:fbox () class-browser-scene))))))


(defmethod tv:dialog-item-action :after ((item class-table))
  "Behandlung der Mausinteraktion
Einfachklick = Anzeige der Super- bzw. Subklassen
               in der Partnertabelle
Klick mit Wahltaste = Anzeige der Subklassen in Graphform
Doppelklick = Anzeige der Sub- bzw. Superklassen mit
Verschiebung der Tabelle nach links bzw. rechts."
  (tv:busy)
  (let ((selected-classes
         (mapcar #'(lambda (index)
                     (elt (tv:table-sequence item)
                          (tv:cell-to-index item index)))
                 (tv:selected-cells item))))
    (unless (null selected-classes)
      (dolist (cell (tv:selected-cells (partner-table item)))
              (tv:cell-deselect (partner-table item) cell))
      (cond ((tv:control-key-p)
             (unless (tv:double-click-p)
               (if (= (length selected-classes) 1)
                 (inspect (first selected-classes))
                 (inspect selected-classes))))
            ((tv:option-key-p)
             (unless (tv:double-click-p)
               (show-graph (graph-view item) selected-classes)))
            ((tv:double-click-p)
             (dolist (cell (tv:selected-cells item))
               (tv:cell-deselect item cell))
             (setf (tv:table-sequence (partner-table item))
                   (tv:table-sequence item))
             (setf (tv:table-sequence item)
                   (funcall (ecase (double-click-shift-direction item)
                              (:left #'pcl::class-direct-subclasses)
```

```
                                        (:right #'pcl::class-local-supers))
                                 (first selected-classes))))
            (t (setf (tv:table-sequence (partner-table item))
                     (if (> (length selected-classes) 1)
                       nil
                       (funcall (ecase (double-click-shift-direction item)
                                        (:right #'pcl::class-direct-subclasses)
                                        (:left #'pcl::class-local-supers))
                                 (first selected-classes))))))))
  (tv:idle))

(defvar *hierarchy-depth* 9
  "Tiefe eines DAGs zur Visualisierung einer Vererbungshierarchie.")

(defun make-line-view-item () (make-instance 'vislib:line-view-item))

(defun show-graph (view selected-classes)
  (setf (layout:layout view)
        (:vbox
         ()
         10      ; 10 Pixel oberer Abstand
         (:hbox
          ()
          10     ; 10 Pixel linker Abstand
          (:gbox
           (:dag selected-classes
                 #'pcl::class-direct-subclasses
                 *hierarchy-depth*
                 #'(lambda (class) t)
                 #'(lambda (class)
                     (vislib:make-label
                      (class-name-as-string class)))
                 #'make-line-view-item
                 #'vislib:western-reference
                 #'vislib:eastern-reference))))))

(setf *class-browser-menu-item* (tv:make-menu-item :menu-item-title "Class
Browser"))

(defmethod tv:menu-item-action ((command (eql *class-browser-menu-item*)))
  (class-browser))

(tv:add-menu-items tv:*tools-menu*
                   *class-browser-menu-item*)
```

C.2. Quellcode des Beispiels mit dem Einschränkungsnetz aus Kapitel 5.2.

C.2.1. Anwendungskomponente

```
(defclass assertion
  ()
  ((name :accessor assertion-name :initarg :name)
   (lower-bound :accessor assertion-lower-bound :initform 0)
   (upper-bound :accessor assertion-upper-bound :initform 1)
   (constraints :accessor assertion-constraints :initform nil))
  (:metaclass pcl::demon-slots-class)) ; Änderung der Metaklasse

(defclass constraint
```

```
   ()
   ((name :accessor constraint-name :initarg :name)
    (output :accessor constraint-output)))

(defclass binary-constraint
  (constraint)
  ((input :accessor constraint-input)))

(defclass ternary-constraint
  (constraint)
  ((input-a :accessor constraint-input-a)
   (input-b :accessor constraint-input-b)))

(defmethod connect2 ((c constraint)
                     (i assertion)
                     (o assertion))
  (setf (constraint-input c) i)
  (setf (constraint-output c) o)
  (push c (assertion-constraints i))
  (push c (assertion-constraints o)))

(defmethod connect3 ((c constraint)
                     (a assertion)
                     (b assertion)
                     (o assertion))
  (setf (constraint-input-a c) a)
  (setf (constraint-input-b c) b)
  (setf (constraint-output c) o)
  (push c (assertion-constraints a))
  (push c (assertion-constraints b))
  (push c (assertion-constraints o)))

(defclass not-box
  (binary-constraint)
  ())

(defclass or-box
  (ternary-constraint)
  ())

(defmethod propagate-via-box ((constraint or-box))
  (let* ((a (constraint-input-a constraint))
         (b (constraint-input-b constraint))
         (o (constraint-output constraint))
         (la (assertion-lower-bound a))
         (ua (assertion-upper-bound a))
         (lb (assertion-lower-bound b))
         (ub (assertion-upper-bound b))
         (lo (assertion-lower-bound o))
         (uo (assertion-upper-bound o)))
    (propagate-via-assertion o constraint (max la lb) (+ ua ub))
    (propagate-via-assertion a constraint (- lo ub) uo)
    (propagate-via-assertion b constraint (- lo ua) uo)))


(defclass and-box
  (ternary-constraint)
  ())

(defmethod propagate-via-box ((constraint and-box))
  (let* ((a (constraint-input-a constraint))
         (b (constraint-input-b constraint))
         (o (constraint-output constraint))
         (la (assertion-lower-bound a))
         (ua (assertion-upper-bound a))
         (lb (assertion-lower-bound b))
```

```
      (ub (assertion-upper-bound b))
      (lo (assertion-lower-bound o))
      (uo (assertion-upper-bound o)))
  (propagate-via-assertion o constraint (+ la lb -1) (min ua ub))
  (propagate-via-assertion a constraint 0 (+ 1 (- ua lb)))
  (propagate-via-assertion b constraint 0 (+ 1 (- uo la)))))


(defmethod propagate-via-assertion ((assertion assertion)
                                    (source constraint)
                                    lower
                                    upper)
  (let* ((old-upper (assertion-upper-bound assertion))
         (old-lower (assertion-lower-bound assertion))
         (new-upper (max 0 (min old-upper upper)))
         (new-lower (min 1 (max old-lower lower))))
    (unless (= old-upper new-upper)
      (setf (assertion-upper-bound assertion) new-upper))
    (unless (= old-lower new-lower)
      (setf (assertion-lower-bound assertion) new-lower))
    (when (or (/= old-lower new-lower) (/= old-upper new-upper))
      (format t "~%Constraint ~a has modified ~a's values:"
              (constraint-name source)
              (assertion-name assertion))
      (format t "~%[~4,2f, ~4,2f] --> [~4,2f, ~4,2f]"
              old-lower old-upper
              new-lower new-upper)
      (dolist (constraint (assertion-constraints assertion))
        (propagate-via-box constraint)))))


(defmethod initiate-propagation ((assertion assertion)
                                 lower
                                 upper)
  (setf (assertion-upper-bound assertion) upper)
  (setf (assertion-lower-bound assertion) lower)
  (format t "~%You have started propagation from ~a with values:"
          (assertion-name assertion))
  (format t "~%[~4,2f, ~4,2f]"
          lower upper)
  (dolist (constraint (assertion-constraints assertion))
    (propagate-via-box constraint)))



(defmethod reset-example-net ()
  (setf (assertion-lower-bound broker1) 0)
  (setf (assertion-upper-bound broker1) 1)
  (setf (assertion-lower-bound broker2) 0)
  (setf (assertion-upper-bound broker2) 1)
  (setf (assertion-lower-bound broker-opinion) 0)
  (setf (assertion-upper-bound broker-opinion) 1)
  (setf (assertion-lower-bound mystic1) 0)
  (setf (assertion-upper-bound mystic1) 1)
  (setf (assertion-lower-bound mystic2) 0)
  (setf (assertion-upper-bound mystic2) 1)
  (setf (assertion-lower-bound mystic-opinion) 0)
  (setf (assertion-upper-bound mystic-opinion) 1)
  (setf (assertion-lower-bound your-opinion) 0)
  (setf (assertion-upper-bound your-opinion) 1))
```

C.2.2. Visualisierungskomponente

```
(defgeneric stock-exchange-wizard (participant)
;  (:documentation "Function considered as an agent that
;is a stock exchange expert. Evaluating this function
;for a special participant returns the participant's
;stock exchange connection.")
)

(defmethod stock-exchange-wizard ((participant assertion))
  "Wizard's information about connections of assertion objects."
  (assertion-constraints participant))

(defmethod stock-exchange-wizard ((participant constraint))
  "Wizard's information about connections of constraint objects."
  (list (constraint-output participant)))

(defmethod stock-exchange-wizard ((participant ternary-constraint))
  "Wizard's information about connections of ternary constraint objects."
  (list (constraint-output participant)
        (constraint-input-a participant)
        (constraint-input-b participant)))

(defmethod stock-exchange-wizard ((participant binary-constraint))
  "Wizard's information about connections of binary constraint objects."
  (list (constraint-output participant)
        (constraint-input participant)))

(defgeneric appearance (participant)
;  (:documentation "This function provides appropriate
;appeareces for stock exchange participants.
;Appearances are represented as view-items")
)

(defmethod appearance ((participant or-box))
  "Appearance of or-box constraint objects.
They are represented by OR in a box with rounded rectangles."
  (vislib:make-label "OR"))

(defmethod appearance ((participant and-box))
  "Appearance of and-box constraint objects.
They are represented by AND in a box with rounded rectangles."
  (vislib:make-label "AND"))

(defmethod appearance ((participant assertion))
  "Appearance of assertion objects.
Assertions are represented by two-level-gauges."
  (vislib:make-two-level-gauge
   #`(participant assertion-lower-bound)
   #`(participant assertion-upper-bound)))


(defun assertion-name-as-string (assertion)
  (symbol-name (assertion-name assertion)))

(defmethod vislib:name-annotation (item (obj assertion) ignore-1 ignore-2 ignore-3)
  (let ((position (tv:view-item-position item))
        (size (tv:view-item-size item)))
    (list (vislib:make-name-annotation-view-item #`(obj assertion-name-as-string)
                                                  item
                                                  (+ (point-h position)
                                                     (point-h size))
                                                  (+ (point-v position)
                                                     (point-v size))))))
```

Anhang C. Auflistung des Quellcodes einiger Beispiele

```
(defvar *max-connection-depth* 9
  "Maximale Anzeigetiefe der Börsenverbindungen.")

(defun tautology (ignore)
  "Used as an expansion predicate."
  t)

(defun connection-visualizer ()
  "Edge of a DAG taken from a view-item library positioned
by references during DAG layout."
  (make-instance 'vislib:line-view-item))

(defclass proceed-button
  (tv:button-dialog-item)
  ())

(defclass stock-exchange-dialog
  (layout:layouted-dialog)
  ((proceed-p :initform nil :accessor stock-exchange-proceed-p)))

(defun make-stock-exchange-dialog (layout)
  (tv:make-layouted-dialog
               :class 'stock-exchange-dialog
               :window-size #@(500 330)
               :window-type ':document-with-zoom
               :window-title "Constraint Example"
               :window-font '(9 "Monaco")
               :default-button nil
               :layout layout))


(defmethod wait-for-proceed ((button proceed-button) &rest ignore)
  (unless tv:*elementary-event-p*
    (tv:dialog-item-enable button)
    (let ((dialog (tv:own-dialog button)))
      (loop
        (if (stock-exchange-proceed-p dialog)
          (progn
            (setf (stock-exchange-proceed-p dialog) nil)
            (return)))))))

(defmethod tv:dialog-item-action :after ((button proceed-button))
  (tv:dialog-item-disable button)
  (setf (stock-exchange-proceed-p (tv:own-dialog button)) t))



(defgeneric application-visualization-coupler (object button))

(defmethod application-visualization-coupler (object button)
  (appearance object))

(defmethod application-visualization-coupler ((participant assertion) button)
  (let ((assertion-gauge (appearance participant)))
    (pcl:add-slot-if-modified-demon
     participant
     'lower-bound
     (pcl:slot-demon-object-notifier #'wait-for-proceed button))
    (pcl:add-slot-if-modified-demon
     participant
     'upper-bound
     (pcl:slot-demon-object-notifier #'wait-for-proceed button))
    (pcl:add-slot-if-modified-demon
     participant
     'lower-bound
```

```
     #'(lambda (assertion-obj name-of-modified-slot
                             old-value new-value)
          (vislib:gauge-update assertion-gauge)))
    (pcl:add-slot-if-modified-demon
     participant
     'upper-bound
     #'(lambda (assertion-obj name-of-modified-slot
                             old-value new-value)
          (vislib:gauge-update assertion-gauge)))
    assertion-gauge))
```

```
(defun stock-exchange-connections-visualization (participants)
  "Opens a window and shows the connnections of
the given participants in a scrollable view."
  (let ((stock-exchange-view (layout:make-layouted-view
                               :scroll-bars ':both
                               :bordered-p nil
                               :auto-scrolling t))
        (proceed-button (tv:make-dialog-item :class 'proceed-button
                                             :dialog-item-text "Proceed"
                                             :dialog-item-enabled-p nil)))

    (make-stock-exchange-dialog (:vbox ()
                                       (:fbox () stock-exchange-view)
                                       5
                                       (:hbox (:height 1/16)
                                              5
                                              (:fbox () proceed-button)
                                              20)
                                       6))
    (setf (layout:layout stock-exchange-view)
          (:vbox ()
                 10
                 (:hbox ()
                        10
                        (:gbox
                         (:annotation
                          ;; Beschriftung von Assertionen
                          ;; durch Generierung weiterer
                          ;; Sichtbereichselememte
                          ;; durch die folgende generische Funktion
                          #'vislib:name-annotation
                          (:dag participants
                                #'stock-exchange-wizard
                                *max-connection-depth*
                                #'tautology
                                #'(lambda (object)
                                    (application-visualization-coupler
                                     object proceed-button))
                                #'connection-visualizer
                                #'vislib:western-reference
                                #'vislib:eastern-reference))))))
    nil))
```

C.3. Quellcode des Beispiels der Kantenbeschriftung mithilfe eines Einschränkungsnetzes

C.3.1. Anwendungskomponente

Dieses Beispielprogramm entstammt einer frühen Version des Babylon-Systems für Symbolics-Rechner.

```
;;; -*- Mode: LISP; Package: "CONSAT" -*-

(in-package 'consat)

(defconstraint l-junction
  (:type primitive)
  (:interface  l r)
  (:relation (:tuple (out in))
             (:tuple (in out))
             (:tuple (in +))
             (:tuple (+ out))
             (:tuple (- in))
             (:tuple (out -))))

(defconstraint t-junction
  (:type primitive)
  (:interface  l m r)
  (:relation (:tuple (out - in))
             (:tuple (out + in))
             (:tuple (out in in))
             (:tuple (out out in))))

(defconstraint k-junction
  (:type primitive)
  (:interface  l m r)
  (:relation (:tuple (- + -))
             (:tuple (in + out))
             (:tuple (+ - +))))

(defconstraint y-junction
  (:type primitive)
  (:interface  l m r)
  (:relation (:tuple (+ + +))
             (:tuple (- - -))
             (:tuple (- out in))
             (:tuple (in - out))
             (:tuple (out in -))))

(defconstraint invers
  (:type primitive)
  (:interface l-r r-l)
  (:relation (:tuple (+ +))
             (:tuple (- -))
             (:tuple (in out))
             (:tuple (out in))))



(defconstraint cube
  (:type compound)
  (:interface a-b b-a
              b-c c-b
```

```
             c-d d-c
             d-e e-d
             e-f f-e
             f-a a-f
             b-g g-b
             d-g g-d
             f-g g-f)
   (:constraint-expressions
    (invers a-b b-a)
    (invers b-c c-b)
    (invers c-d d-c)
    (invers d-e e-d)
    (invers e-f f-e)
    (invers f-a a-f)
    (invers b-g g-b)
    (invers d-g g-d)
    (invers f-g g-f)
    (l-junction a-f a-b)
    (l-junction c-b c-d)
    (l-junction e-d e-f)
    (k-junction b-c b-g b-a)
    (k-junction d-e d-g d-c)
    (k-junction f-a f-g f-e)
    (y-junction g-b g-d g-f)))
```

C.3.2. Visualisierungskomponente

```
(defconstant value-circle-height 20)

(defmacro :cube (&rest cube-args)
  `(list ':cube ,@cube-args))

(defmethod layout:layout-spec-p-using-key ((key (eql ':cube)))
  t)

(defmethod layout:parse-layout-spec-using-key ((key (eql ':cube)) cube-layout-spec)
  (let ((length-edges (second cube-layout-spec))
        (edge-item-creation-fcn (third cube-layout-spec))
        (vertext-creation-fcn (fourth cube-layout-spec)))
    (let* ((delta (round (* 0.25 length-edges (sqrt 2))))
           (delta+length (+ delta length-edges)))
      (let ((c-b (funcall edge-item-creation-fcn 'c-b))
            (d-c (funcall edge-item-creation-fcn 'd-c))
            (g-d (funcall edge-item-creation-fcn 'g-d))
            (b-g (funcall edge-item-creation-fcn 'b-g))
            (b-a (funcall edge-item-creation-fcn 'b-a))
            (f-g (funcall edge-item-creation-fcn 'f-g))
            (e-d (funcall edge-item-creation-fcn 'e-d))
            (a-f (funcall edge-item-creation-fcn 'a-f))
            (f-e (funcall edge-item-creation-fcn 'f-e))
            (a (funcall vertext-creation-fcn
                        'a
                        0 delta+length))
            (b (funcall vertext-creation-fcn
                        'b
                        0 delta))
            (c (funcall vertext-creation-fcn
                        'c
                        delta 0))
            (d (funcall vertext-creation-fcn
                        'd
                        delta+length 0))
            (e (funcall vertext-creation-fcn
                        'e
```

Anhang C. Auflistung des Quellcodes einiger Beispiele

```
(defmethod tv:view-item-draw :after ((item edge-view-item) view dialog)
  "Kanten werden durch Linien dargestellt. Die Beschriftung (Labeling) der
   Kante erfolgt innerhalb eines kleinen Kreises in der Mitte der Linie.
   Zur Vereinfachung wird davon ausgegangen, daß der Font des Fenster 'dialog'
   auf (Monaco 9) gesetzt ist. Bei den Beschriftungen wird von den aus
   der Literatur bekannten etwas abgewichen (vgl. Waltz, Ballard & Brown, etc.)
   Beschriftungen: unconstrained : ? (unconstrained ist ein technischer Wert
                                      der zugrunde liegenden Constraint-Systems
Consat)
                      Pfeil : (rechts vom Pfeil ist der Körper, links davon der
                               Hintergrund)
                              Je nach Kantenrichtung im Constraint-Netz durch die
                              Symbole consat::in oder consat::out implementiert.
                              Beide Wert sind hier dargestellt durch
                              einen 180-Grad-Bogen in Richtung des Körpers"

  (flet ((draw-180-degrees-arc
          (dialog p1 p2 m  size direction)
          ; p1 und p2 geben die (gerichtete) Kante an.
          ; m ist der Mittelpunkt der Kante, size die Größe des Beschriftungskreises
          ; auf der Kante.
          (tv:move-to dialog m)
          (let* ((vect (tv:subtract-points p2 p1)) ; Richtungsvektor der Kante
                 (angle ; Winkel der Drehwinkelanschlags von der x-Achse
                        ; (math. Drehrichtung) in Grad (Bereich -180 bis 180)
                  (/ (* 180 (if (zerop (point-h vect)) ; nicht durch 0 teilen
                                (if (> (point-v p2)
                                       (point-v p1))
                                  (- (/ pi 2))
                                  (/ pi 2))
                                (atan (- (point-v vect)) (point-h vect))))
                     pi)))

            ;; Normierung auf positive Drehwinkel
            (if (< angle 0)
              (setf angle (+ angle 360)))

            ;; Anpassung an Quickdraw Drehanschlag und Winkel (siehe Inside Mac)
            ;; Der Drehanschlag zeigt hier in Richtung der kartesische y-Achse.
            (setf angle (+ 90 (- 360 angle)))

            (tv:frame-arc dialog
                          (+ (round angle)
                             (if (eq direction 'consat::out)
                               0
                               180))
                          180
                          (tv:add-points m (tv:make-point (- 2 size)
                                                          (- 2 size))) ; etwas
kleiner
                          (tv:add-points m (tv:make-point (- size 2)
                                                          (- size 2)))))))

    (let* ((references (tv:references-of-this-item item))
           (p1 (tv:reference-position (first references)))
           (p2 (tv:reference-position (second references)))
           (half-circle-height (round (/ value-circle-height 2)))
           (m (tv:add-points (scalar-product (tv:subtract-points p2 p1)
                                             0.5)
                             p1))
           ;; Quickdraw verlangt die Angabe eines Rechtecks zum Zeichnen
           ;; eines Kreises bzw. Kreisbogens.
           (circle-left-top (tv:subtract-points m
                                                (tv:make-point half-circle-height
                                                               half-circle-height)))
           (circle-right-bottom (tv:add-points m
                                               (tv:make-point half-circle-height
```

```
                   delta+length length-edges))
             (f (funcall vertext-creation-fcn
                   'f
                   length-edges delta+length))
             (g (funcall vertext-creation-fcn
                   'g
                   length-edges delta)))
        (layout:parse-layout-spec (vislib:middle-reference d-c d))
        (layout:parse-layout-spec (vislib:middle-reference d-c c))
        (layout:parse-layout-spec (vislib:middle-reference c-b c))
        (layout:parse-layout-spec (vislib:middle-reference c-b b))
        (layout:parse-layout-spec (vislib:middle-reference g-d d))
        (layout:parse-layout-spec (vislib:middle-reference g-d g))
        (layout:parse-layout-spec (vislib:middle-reference b-g g))
        (layout:parse-layout-spec (vislib:middle-reference b-g b))
        (layout:parse-layout-spec (vislib:middle-reference b-a b))
        (layout:parse-layout-spec (vislib:middle-reference b-a a))
        (layout:parse-layout-spec (vislib:middle-reference f-g g))
        (layout:parse-layout-spec (vislib:middle-reference f-g f))
        (layout:parse-layout-spec (vislib:middle-reference e-d e))
        (layout:parse-layout-spec (vislib:middle-reference e-d d))
        (layout:parse-layout-spec (vislib:middle-reference a-f a))
        (layout:parse-layout-spec (vislib:middle-reference a-f f))
        (layout:parse-layout-spec (vislib:middle-reference f-e f))
        (layout:parse-layout-spec (vislib:middle-reference f-e e))

        (tv:as-group d-c
              c-b
              g-d
              b-g
              b-a
              f-g
              e-d
              a-f
              f-e a b c d e f g
              )
        (list d-c
              c-b
              g-d
              b-g
              b-a
              f-g
              e-d
              a-f
              f-e a b c d e f g
              )))))


(defconstant l-junction-id 13548)
(defconstant k-junction-id 30146)
(defconstant y-junction-id 23823)

(defclass edge-view-item
  (tv:moveable-view-item-mixin tv:view-item)
  ((name :initarg :name :reader edge-name)
   (possible-values :accessor possible-values :initarg :possible-values
                    :initform 'consat::unconstrained))
  (:default-initargs :bordered-p nil)
  (:metaclass pcl:indirect-slots-class))

(defun scalar-product (p factor)
  (let ((h (tv:point-h p))
        (v (tv:point-v p)))
    (tv:make-point (round (* h factor))
                   (round (* v factor)))))
```

```
                                                                          half-circle-height))))
      (tv:move-to dialog p1)
      (tv:line-to dialog p2)
      (tv:fill-arc dialog
                   tv:*white-pattern*
                   0 360
                   circle-left-top
                   circle-right-bottom)
      (tv:frame-arc dialog
                    0 360
                    circle-left-top
                    circle-right-bottom)
      (if (eq (possible-values item) 'consat::unconstrained)
          ; Hier müßte der Plural des Bezeichners possible-values eigentlich
          ; überdacht werden...
        (progn (tv:move-to dialog (tv:add-points m #@(-3 3)))
               (tv:draw-string dialog "?"))
        (do ((index 0 (1+ index)))
            ((>= index (length (possible-values item))))
          (case (nth index
                     (possible-values item))
            (+
             (tv:move-to dialog (tv:add-points #@(-5 0) m))
             (tv:draw-string dialog "+"))
            (-
             (tv:move-to dialog (tv:add-points #@(0 5) m))
             (tv:draw-string dialog "-"))
            (consat::out (draw-180-degrees-arc dialog p1 p2 m
                                               half-circle-height 'consat::out))
            (consat::in (draw-180-degrees-arc dialog p1 p2 m
                                              half-circle-height 'consat::in)))))))

(defmethod tv:view-item-undraw ((item edge-view-item) view dialog position size
references)
  (let* ((p1 (tv:reference-position (first references)))
         (p2 (tv:reference-position (second references)))
         (half-circle-height (round (/ value-circle-height 2)))
         (m (tv:add-points (scalar-product (tv:subtract-points p2 p1)
                                           0.5)
                           p1))
         ;; Quickdraw verlangt die Angabe eines Rechtecks zum Zeichnen
         ;; eines Kreises bzw. Kreisbogens.
         (circle-left-top (tv:subtract-points m
                                              (tv:make-point half-circle-height
                                                             half-circle-height)))
         (circle-right-bottom (tv:add-points m
                                             (tv:make-point half-circle-height
                                                            half-circle-height))))
    (let ((previous-pen-pattern (tv:pen-pattern dialog)))
      (setf (tv:pen-pattern dialog) tv:*white-pattern*)
      (tv:move-to dialog p1)
      (tv:line-to dialog p2)
      (setf (tv:pen-pattern dialog) previous-pen-pattern))
    (tv:erase-arc dialog
                  0 360
                  circle-left-top
                  circle-right-bottom)))


(defun make-edge (name)
  (let* ((name (intern (symbol-name name) (find-package 'consat)))
         (constraint-var-info
          (cdr (assoc name (zl:send (zl:send consat::*current-constraint-base*
                                             :get 'consat::cube)
                                    :net-spec)))))
    (let ((edge-item
           (make-instance 'edge-view-item
```

```
                        :name name
                        :possible-values
                            #`(constraint-var-info consat::var-info-values)
                        :left-offset 10
                        :top-offset 10
                        :right-offset 10
                        :bottom-offset 10)))
      (pcl:add-slot-if-modified-demon
       constraint-var-info
       'consat::values
       (gensym "edge-update")
       #'(lambda (ignore-1
                  ignore-2
                  ignore-3
                  ignore-4) (if (tv:own-view edge-item)
                               (tv:view-item-redraw edge-item))))
      edge-item)))

(defclass vertex-view-item
  (tv:icon-view-item)
  ((name :initarg :name :reader vertex-name)))

(defun make-vertex (name h v)
  (make-instance 'vertex-view-item
                 :name name
                 :icon-id (ecase name
                                    ((a e c) l-junction-id)
                                    (g y-junction-id)
                                    ((b d f) k-junction-id))
                 :view-item-position (make-point h v)))

(defmethod vislib:name-annotation ((item vertex-view-item)
                                   ignore-1 ignore-2 ignore-3 ignore-4)
  (let ((position (tv:view-item-position item))
        (size (tv:view-item-size item)))
    (list (vislib:make-name-annotation-view-item
           (symbol-name (vertex-name item))
           item
           (+ (point-h position)
              (point-h size))
           (+ (point-v position)
              20
              (point-v size))))))

(defun cube-visualization ()
  (let ((cube-view (layout:make-layouted-view :scroll-bars ':both
                                              :bordered-p nil
                                              :auto-scrolling t)))

    (layout:make-layouted-dialog :window-size #@(300 300)
                                 :window-type ':document-with-zoom
                                 :window-title "Cube Example"
                                 :window-font '(9 "Monaco")
                                 :layout (:fbox () cube-view))


    (setf (layout:layout cube-view)
          (:vbox ()
                 30
                 (:hbox ()
                        30
                        (:gbox (:annotation
                                #'vislib:name-annotation
                                (:cube 150
                                       #'make-edge
                                       #'make-vertex))))))
    nil))
```