Modellvisualisierung für die Spezifikationssprache ADORA

DISSERTATION

DER WIRTSCHAFTSWISSENSCHAFTLICHEN FAKULTÄT
DER UNIVERSITÄT ZÜRICH

ZUR ERLANGUNG DER WÜRDE
EINES DOKTORS DER INFORMATIK

vorgelegt von
STEFAN BERNER
von
Deutschland

genehmigt auf Antrag von

PROF. DR. MARTIN GLINZ
PROF. DR. HORST LICHTER

JUNI 2002

Die Wirtschaftswissenschaftliche Fakultät der Universität Zürich, Lehrbereich Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 15. Mai 2002*

Der Lehrbereichsvorsteher: Prof. Dr. Martin Glinz

* Datum der Promotionsfeier

"… Wir wissen nicht, wir raten … Alles Wissen ist Vermutungswissen … Wissenschaft ist nicht Besitz von Wissen, sondern Suche nach der Wahrheit … Das Verfahren der Wissenschaft sollte Falsifikation und nicht Verifikation sein, ihre Methode Mutmassung statt Anmassung …"

Karl Popper

Danksagung

Die vorliegende Dissertation ist während meiner Tätigkeit am Institut für Informatik der Universität Zürich entstanden. Ausser den nachfolgend namentlich Erwähnten, haben viele weitere Personen zum Gelingen dieser Arbeit beigetragen. Ich danke daher an dieser Stelle all jenen.

Mein besonderer Dank gilt meinem Doktorvater Herrn Prof. Dr. Martin Glinz, der mich unter anderem gelehrt hat, was es heisst, wissenschaftlich zu arbeiten und welcher darüber hinaus essentiell für die gesamte Publikationsarbeit war. Ein besonderes Dankeschön auch für die sorgfältige und konstruktive Betreuung meiner Arbeit. Bei Herrn Prof. Dr. Horst Lichter bedanke ich mich für die kritische Durchsicht der Arbeit und die vielen fundierten Ratschläge.

Weiter möchte ich mich bei den Kollegen am Institut für die angenehme und inspirierende Arbeitsatmosphäre sowie für viele hilfreiche Ratschläge bedanken. Besonders hervorheben möchte ich an dieser Stelle die produktive Zusammenarbeit mit Denis Antonioli, Martin Arnold, Stefan Joos, Marcus Pilz, Mathias Richter, Johannes Ryser, Nancy Schett, Reto Schmid sowie Anca Vaduva. Stefan Joos gebührt hierbei noch ein ganz spezieller Dank für die harten, jedoch stets konstruktiven Diskussionsrunden.

Zwei Studenten haben mir bei der Realisierung meiner Ideen geholfen: Robert von Känel und Nancy Schett haben durch ihre Diplomarbeiten zum Gelingen der vorliegenden Arbeit beigetragen.

Der eher undankbaren Aufgabe diese Arbeit korrekturzulesen, sind Cornelia Metzler und Nancy Schett nachgekommen, wofür ich mich hier ganz herzlich bedanken möchte.

Zum Schluss möchte ich mich noch bei meiner Mutter Bertl und meinem Vater Fritz Berner bedanken. Ohne sie und alles was sie für mich getan haben, wäre diese Arbeit nicht möglich gewesen.

Zürich, im April 2002

Stefan Berner

Zusammenfassung

Diese Arbeit präsentiert einen Ansatz für die Visualisierung von hierarchisch gegliederten Objekt-Modellen, welcher auf der Verwendung von Fischaugensichten (*fish eye views*) beruht. Hiermit können Sichten auf ein Objektmodell bereitgestellt werden, welche lokales Detail und globalen Kontext im gleichen Diagramm integrieren. Dies ermöglicht dem Benutzer eine vereinfachte Navigation innerhalb der hierarchischen Struktur eines Modells unter voller Erhaltung der im Modell enthaltenen Abstraktionen. Die Arbeit stellt die Ideen vor, welche hinter dem Konzept liegen, illustriert den Verfeinerungsmechanismus, erläutert den hierfür entwickelten, unterliegenden Algorithmus sowie die Implementierung eines Werkzeugprototypen. Diese Arbeit ist im Rahmen eines Projekts entstanden, welches die Entwicklung einer objektorientierten Modellierungsmethode genannt ADORA zum Ziel hat.

Abstract

This work presents an approach for the visualization of hierarchical object models that is based on the notion of fisheye views. This concept can display local detail and global context of a view in the same diagram, thus allowing a user to navigate easily in hierarchical structures without offending the principle of abstraction. This work introduces the ideas behind the concept, illustrates the zooming mechanism, the algorithm for the implementation and a tool prototype. The work presented here is part of an effort to create a new object modeling method called ADORA that provides strong support for hierarchical composition and decomposition.

Inhaltsverzeichnis

Grundlagen Visualisierung

Visualisierung von ADORA-Modellen

ADORA Werkzeugumgebung

Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Einführung

Dieses Einführungskapitel steckt den Rahmen der vorliegenden Arbeit ab. Wie für eine Einführung allgemein üblich, werden viele der hier verwendeten Begriffe nicht oder nur sehr kurz erläutert. Eine eingehendere Behandlung erfolgt in den nachfolgenden Kapiteln.

In Kapitel 1 wird in den Kontext dieser Arbeit eingeführt, die eigentliche Aufgabe motiviert sowie die Einbettung der Arbeit beschrieben.

Im Anschluss werden in Kapitel 2 die wichtigsten Ergebnisse dieser Arbeit kurz zusammengefasst.

Kapitel 3 beendet die Einführung mit einer Beschreibung des weiteren Aufbaus dieser Arbeit.

1 MOTIVATION

Es ist eine allgemein bekannte und anerkannte Tatsache, dass Software, wie auch deren Entwicklung, nicht nur komplex ist, sondern auch zunehmend komplexer wird, da die Systeme, welche realisiert werden, immer umfassender werden. Software-Systeme sind die komplexesten Systeme, die bisher von Menschenhand geschaffen wurden.

Von Seite der Software-Entwicklung wird u.a. versucht, diese Komplexität mit immer leistungsfähigeren Entwicklungsmethoden zu beherrschen. Rückgrat einer solchen Methode ist i.d.R. die Sprache. Seien es nun Programmier-, Entwurfs- oder Spezifikationssprachen, Sinn und Zweck einer solchen Sprache ist es (fast) immer, die bei der Entwicklung von Software anfallenden Beschreibungen, zumindest werkzeugseitig, zunehmend auf die 'Brooks'sche Essenz' [Broo87] reduzieren zu können.

1.1 ADORA & ADORA-L

ADORA ist eine objektorientierte Methode zur Spezifikation von Anforderungen und Beschreibung von Software-Entwürfen, welche in der Forschungsgruppe Requirements Engineering (kurz RERG) der Universität Zürich entwickelt wird.

ADORA steht für 'Analyse und Beschreibung von Anforderungen und Architekturen' oder als Akronym für (*A)nalysis and (D)escription (O)f (R)equirements and (A)rchitecture*.

Eine Methode besteht – gemäss Methodendreieck [Fru+91] – gemeinhin aus drei aufeinander abgestimmten Teilen: einer Notation bzw. Sprache, einem Werkzeug und einem Prozess. Die ADORA zugrundeliegende Sprache wird ADORA-L genannt, wobei die Begriffe 'Methode' und 'Sprache' bzw. ADORA und ADORA-L oft synonym verwendet werden. Auch hier wird dies mitunter so gehandhabt – zumindest dort, wo der Kontext offensichtlich und daher eine penible Unterscheidung nicht zwingend notwendig ist.

ADORA-L ist primär eine grafische Spezifikationssprache, d.h. der Hauptteil eines ADORA-L-Modells wird mit Hilfe grafischer Symbole beschrieben. Mittels ADORA wird ein System durch ein in der Sprache ADORA-L beschriebenes Modell spezifiziert. Dieses Modell – auch *Systemmodell* genannt – ist eine Beschreibung der Struktur, des Verhaltens sowie der Funktionalität eines Systems durch ein *einziges*, integriertes Modell.

Ein zentrales Grundparadigma eines ADORA-Modells ist die Modellierung auf der Ebene *abstrakter Objekte* und nicht die Klassenmodellierung. Modellierung auf Ebene abstrakter Objekte führt in Verbindung mit entsprechenden Gliederungs- sowie Abstraktionsmechanismen zu sehr verständlichen Spezifikationen, die aber dennoch dort präzise sind, wo dies notwendig oder gewünscht ist.

Der zentrale Abstraktionsmechanismus in ADORA ist eine rekursiv aufgebaute *Teil-Ganzes-Hierarchie*: Hiermit wird ein System konsequent in Komponenten gegliedert, welche zur Bildung von Kompositionen verwendet werden. So konstruierte Kompositionen können dann ggf.

selbst wieder als Komponenten Verwendung finden. Komponente und Komposition sind also Rollen, welche ein Objekt in einer – durch ein Systemmodell repräsentierten – Systembeschreibung einnehmen kann.

Struktur und Verhalten eines Systems lassen sich in ADORA auf verschiedenen Abstraktionsebenen beschreiben. In einem ADORA-Modell kann jederzeit sowohl von Detailstrukturen als auch von Detailverhalten abstrahiert werden. Da dieses Konzept und speziell der zugrundeliegende Abstraktionsmechanismus auch für grosse Modelle hervorragend skaliert, lassen sich mittels ADORA-Modellen komplexe Systeme beschreiben.

Eine Voraussetzung hierfür ist das Vorhandensein eines integrierten Modells. Nur so ist es formal zu handhaben, welche Konsequenzen beispielsweise die Abstrahierung oder Detaillierung gewisser Teilstrukturen eines Systemmodells auf die Beschreibung anderer, hiervon abhängiger Beschreibungen hat, ohne dass diese erneut und explizit – entweder in detaillierter oder in abstrakter Form – modelliert werden müssen. Grundsätzlich wäre es ohne ein integriertes Modell zumindest recht aufwendig, Aussagen darüber zu machen, auf welcher Abstraktionsebene welche Beschreibungen für den Betrachter relevant sind. Wird beispielsweise ein Taschenrechner spezifiziert und der Modellierer arbeitet gerade an den Details der Steuerung des Rechenwerks, so ist es hilfreich, wenn er diejenigen Teilsysteme – wie z.B. das Bedienfeld – kennt, die mit dem Rechenwerk zusammenarbeiten, ohne jedoch permanent mit den Details dieser Teilsysteme konfrontiert zu werden. Muss er diese Informationen separat beschreiben oder erst aus einer Vielzahl von Detailbeschreibungen zusammenstellen, so hat er es bedeutend schwerer als sein Kollege, der auf Basis einer Methode (wie z.B. ADORA) arbeitet, die ihn anleitet, ein System derart strukturiert zu beschreiben, dass diese Betrachtungsweisen ein natürlicher Bestandteil des Modellierungsprozesses sind und sich dementsprechend auch im Systemmodell wiederfinden.

1.2 Modellvisualisierung

Damit diese Merkmale einer Methode bzw. der unterliegenden Sprache auch entsprechend durch den Modellierer oder Modellbetrachter genutzt werden können, ist es wichtig, dass sie adäquat durch eine entsprechende Visualisierung unterstützt werden. Die vorliegende Arbeit beschäftigt sich in erster Linie mit der Visualisierung von ADORA-Modellen.

Im Zentrum der Arbeit steht die Frage, auf welche Art und Weise die Sprache ADORA-L durch eine adäquate Visualisierung unterstützt werden kann, d.h. wie visualisiert werden muss, damit möglichst viel von dem, was mit ADORA-L modelliert werden kann, auch in einer intuitiven und natürlichen Form beim Benutzer ankommt. Im Kontext von ADORA ist es wichtig, dass die Visualisierung sowohl den Abstraktionsmechanismen von ADORA-L Rechnung trägt, wie auch berücksichtigt, dass das zugrundeliegende Modell integriert ist, also Aspekte wie Struktur, Verhalten und Funktionalität eines Systems nicht durch separate, mehr oder weniger zusammenhängende Teilmodelle beschrieben werden.

Wie bereits angedeutet, erlaubt es ADORA-L, ein System auf verschiedenen Abstraktionsebenen zu beschreiben. Es kann somit in ADORA-L *nicht* davon ausgegangen werden, dass ein Diagramm, eine Teilansicht, o.ä. existiert, welche ein bestimmtes 'konstantes' Abstraktionsniveau hat. Vielmehr muss davon ausgegangen werden, dass auf Basis des integrierten Modells die Darstellung generiert wird, welche dem Benutzer (Modellierer, Betrachter, etc.) genau die Informationen bereitstellt, welche er für die aktuelle Aktivität benötigt, also die Darstellung auf dem passenden Abstraktionsniveau mit den entsprechenden Aspektbeschreibungen wie Struktur, Verhalten und/oder Funktionalität. Wird dies bei der Visualisierung nicht entsprechend unterstützt, so kommen viele Vorteile, welche die Sprache aufweist, gar nicht erst zur Geltung, mit der Konsequenz, dass sie für den Benutzer der Sprache nicht als solche zu erkennen sind und dementsprechend auch nicht genutzt werden können.

1.3 Einbettung

Diese Dissertation ist im Rahmen des Projekts 'LATO – Language and Tool for Object-Oriented, Semiformal Requirements Specification' in der Forschungsgruppe Requirements Engineering (RERG) am Institut für Informatik (IFI) der Universität Zürich entstanden. Das Projekt wird von Prof. Dr. Martin Glinz geleitet und wurde u.a. durch den Schweizerischen Nationalfonds zur Förderung der wissenschaftlichen Forschung (Projekte Nr. 21-40518.94 und Nr. 20-47196.96) finanziell unterstützt.

2 BEITRAG

In dieser Arbeit wird der Bereich Modellvisualisierung für die Sprache ADORA-L behandelt. Hierfür werden Ideen zur perspektivbasierten Sichtengenerierung aus dem Bereich 'Mensch-Maschine-Kommunikation (HCI)' übertragen, die geeignet sind, Darstellungen strukturierter Modelle mit variablem Abstraktionsgehalt zu ermöglichen und die in dieser Form bislang zur Visualisierung von Spezifikationsmodellen nicht angewandt wurden.

Konkret wird zur Visualisierung von ADORA-Modellen die Idee der Fischaugensichten [Furn86] aufgegriffen und gezeigt, wie – basierend auf diesem Konzept – Sichten für hierarchische Objektmodelle generiert werden können, welche simultan lokales Detail und globalen Kontext bereitstellen. Derartige Sichten tragen insbesondere dann zu einer Reduktion des kognitiven Ballasts beim Betrachter bzw. Benutzer bei, wenn die darzustellenden Modelle hierarchisch strukturiert sind und hohe Integrationsdichte aufweisen. Beides ist zutreffend für ADORA-Modelle. Mit steigender Modellgrösse bzw. abnehmender Übersichtlichkeit verstärkt sich i.d.R. der Nutzen einer Sicht, die lokales Detail und globalen Kontext zusammen darstellt.

Ansätze, die sich mit der Visualisierung bestimmter Strukturen, Modelle, etc. auseinandersetzen, beschränken sich oft auf den Bereich Repräsentation, d.h. sie beschäftigen sich hauptsächlich mit der Frage, auf welcher technischen und algorithmischen Basis Sichten generiert werden. Aspekte der Sekundärnotation wie auch die Navigation werden hierbei meist ausser

acht gelassen. Das in dieser Arbeit vorgestellte Visualisierungskonzept umfasst sowohl die Repräsentation der Modelle, wie auch die Navigation in denselben. Im Rahmen der Diskussion verschiedener potentiell geeigneter Konzepte wird dieser Bereich auf eine solide begriffliche und modelltheoretische Basis gestellt. Insbesondere logische Navigation mittels eines selektiven Zooms wird als eine Möglichkeit identifiziert, welche äusserst vorteilhaft sein kann, in heutigen CASE-Werkzeugen aber keine Verwendung findet.

Die hier vorgestellten Resultate sind nicht nur die Grundlage für die Visualisierung von ADORA-Modellen, sie können genauso für die Repräsentation anderer, hierarchisch strukturierter Modelle, wie z.B. StateCharts [Hare87] oder ROOM [Sel+94] Modelle benutzt werden. Ebenso lassen sich Generalisierungs/Spezialisierungs-Strukturen auf diese Art und Weise betrachten. Eine Untersuchung, ob und in welchem Masse das hier vorgestellte Visualisierungskonzept auch dafür geeignet wäre, ist allerdings nicht Gegenstand dieser Arbeit.

Neben der Herleitung des eigentlichen Visualisierungskonzepts sowie dessen algorithmischer Realisierung wird ausserdem demonstriert, wie ein derartiges Konzept in eine entsprechende Werkzeugumgebung für die Sprache ADORA-L eingebettet werden kann. Die Werkzeugumgebung ist hierbei derart gestaltet, dass sie mit geringem Aufwand an anfallende Änderungen der unterliegenden Sprache angepasst werden kann.

‘Angepasst werden’ heisst in diesem Zusammenhang zweierlei: Erstens ist es mit geringem Aufwand möglich, neue Elemente bzw. Sprachkonstrukte in Werkzeugkomponenten wie beispielsweise den grafischen Editor zu integrieren oder bestehende Elemente zu ändern bzw. diese wieder zu entfernen. Zweitens können Integritätsbedingungen für die unterliegenden Modelle formuliert werden, wobei das Werkzeug (bei entsprechender Konfiguration) in der Lage ist, ein Modell automatisch gegen die formulierten Bedingungen zu prüfen.

Die zur Formulierung von Integritätsbedingungen entwickelte Notation – ADORA-IB genannt – ist hier nicht nur ein weiteres ‘Add-On’ zur Basissprache ADORA-L (wie etwa OCL [IBM+97] zur UML), welches werkzeugtechnisch nicht weiter von Bedeutung ist, sondern Voraussetzung dafür, dass bei der Anpassung der Werkzeugumgebung an Änderungen der unterliegenden Sprache die Modellkonsistenz sichergestellt werden kann, ohne dass der eigentliche Code der Werkzeugumgebung geändert werden muss.

3 INHALTSÜBERSICHT

Die Arbeit gliedert sich in vier aufeinander aufbauende Teile. Begonnen wird mit den Teilen ‘Grundlagen ADORA’ und ‘Grundlagen Visualisierung’. Diese beiden Teile führen in die grundlegenden Konzepte und Begriffe ein, welche für das Verständnis dieser Arbeit notwendig sind und welche dementensprechend auch immer wieder verwendet werden. Darauf folgen die Ausführungen zur Visualisierung von ADORA-Modellen. Im Anschluss wird auf die Umsetzung des Visualisierungskonzepts in der ADORA Werkzeugumgebung eingegangen.

Grundlagen ADORA

- Kapitel 4 beginnt mit einer kurzen Einführung in die allgemeine Modelltheorie. Es werden einige Begriffe eingeführt, welche im weiteren Verlauf der Arbeit immer wieder vorkommen. Nicht nur im Teil ‘Grundlagen ADORA’, sondern auch im Teil ‘Grundlagen Visualisierung’ wird auf einige der eingeführten Begriffe zurückgegriffen.
- Schwerpunkt dieser Arbeit ist die Visualisierung von ADORA-Modellen. So erfolgt zunächst in Kapitel 5 eine kurze und mit Beispielen ergänzte Einführung in die Spezifikationssprache ADORA-L.

Grundlagen Visualisierung

- Ein Grossteil dieser Arbeit beschäftigt sich mit der adäquaten Visualisierung von ADORA-Modellen. Daher werden in Kapitel 6 die Visualisierung betreffende Grundlagen und Begriffe eingeführt. Auf die Visualisierung hierarchischer Strukturen wird speziell gegen Ende des Kapitels in einem eigenen Unterkapitel eingegangen.
- In Kapitel 7 wird auf Visualisierungskonzepte an sich eingegegangen. Hierfür werden die Merkmale angesprochen, die zusammengenommen ein Visualisierungskonzept ausmachen. Hier wird – vorbereitend auf nachfolgende Kapitel – eine Struktur für die Beschreibung verschiedener Visualisierungskonzepte eingeführt. Nachfolgend werden typische Vertreter bestehender Visualisierungskonzepte vorgestellt.
- Eine Zusammenfassung findet sich in Kapitel 8. Insbesondere werden hier existierende Fischaugenkonzepte im Hinblick auf ihre Eignung zur Visualisierung von ADORA-Modellen diskutiert.

Visualisierung von ADORA-Modellen

In diesem Teil wird das eigentliche Konzept zur Visualisierung von ADORA-Modellen beschrieben. ADORA-Modelle weisen zwei Besonderheiten auf: Zum einen sind sie durch die entsprechenden Abstraktionsmechanismen hierarchisch gegliedert und zum anderen sind es

integrierte Aspektmodelle. Wie diesen Eigenschaften bei der Visualisierung Rechnung getragen werden kann, ist Gegenstand dieses Teils der Arbeit.

- Kapitel 9 beschäftigt sich mit den Anforderungen an ein Visualisierungskonzept für ADORA-Modelle. Dies erfolgt mitunter rückblickend auf die in Kapitel 7.1 vorgestellten Konzepte.
- Wie das zuvor eingeführte Visualisierungskonzept realisiert werden kann, wird in Kapitel 10 erläutert. Hier wird schwerpunktmässig auf die Algorithmik der im Rahmen dieser Arbeit entwickelten Projektionstechnik eingegangen, die für die Generierung von Sichten zur Anwendung kommt.
- Kapitel 11 zeigt die Anwendung des Konzepts für ADORA. In Kapitel 12 findet sich dann eine Zusammenfassung sowie eine abschliessende Diskussion des zuvor vorgestellten Visualisierungskonzepts für ADORA-Modelle.

ADORA Werkzeugumgebung

Nachdem nun das Visualisierungskonzept vorgestellt ist, wird in diesem Teil ein Werkzeug behandelt, welches dieses Konzept umsetzt.

- Kapitel 13 gibt einen kurzen konzeptionellen Überblick. Neben der Grobarchitektur des Werkzeugs wird einerseits eine Übersicht darüber vermittelt, wie das Werkzeug konfiguriert wird, und andererseits angeschnitten, wie der Bereich Modellintegrität angegangen wird.
- In Kapitel 14 wird detailliert darauf eingegangen, wie die Konfiguration des Werkzeugs tatsächlich abläuft, d.h. was mittels welcher Notation konfiguriert werden kann, wie die Arbeit mit dem Konfigurationsskript abläuft und wo bei Erweiterungen wie implementiert werden muss.
- Kapitel 15 behandelt dann eingehend den Bereich Modellintegrität. Es wird beschrieben, wie Integritätsbedingungen für ADORA-Modelle formuliert werden, wie das Zusammenspiel dieser Bedingungen mit der Konfiguration des Werkzeugs abläuft und wie diese Bedingungen automatisch überprüft werden können.
- Der Teil zur Werkzeugumgebung endet in Kapitel 16 mit einer abschliessenden Diskussion und Zusammenfassung.

Anmerkung: Die Zusammenfassung und Diskussion befindet sich nicht in einem eigenen Kapitel am Ende der Arbeit, sondern ist auf zwei bzw. drei Kapitel verteilt. In Kapitel 12 wird auf das Visualisierungskonzept eingegangen und in Kapitel 16 auf die Werkzeugumgebung. Vorgelagert werden in Kapitel 8 bekannte wie umgesetzte Visualisierungskonzepte diskutiert. Auf eine Zusammenfassung am Ende der Arbeit wurde bewusst verzichtet, da sie aus Sicht des Autors hauptsächlich eine Wiederholung o.g. Kapitel wäre.

Grundlagen ADORA

Kapitel 4 beginnt mit einer kurzen Einführung in die allgemeine Modelltheorie. Es werden einige Begriffe eingeführt, welche im weiteren Verlauf der Arbeit immer wieder vorkommen. Nicht nur im Teil ‘Grundlagen ADORA’, sondern auch im Teil ‘Grundlagen Visualisierung’ wird auf einige der eingeführten Begriffe zurückgegriffen.

Schwerpunkt dieser Arbeit ist die Visualisierung von ADORA-Modellen. So erfolgt zunächst in Kapitel 5 eine kurze und mit Beispielen ergänzte Einführung in die Spezifikationssprache ADORA-L.

4 Modelltheoretische Grundlagen

Der Begriff des Modells wird in dieser Arbeit sowohl im Kontext der Sprache ADORA-L als auch im Kontext der Visualisierung von ADORA-L-Modellen intensiv gebraucht. Daher wird eine kurze Einführung in die Ideen der allgemeinen Modelltheorie [Stac73] gegeben. Dieses Kapitel klärt den Begriff des Modells und beschreibt ausserdem einige wichtige Merkmale und Eigenschaften von Modellen.

4.1 Modellbegriff

Nach Luft [Luft84] sind *Modelle* Darstellungen, Muster oder Schemata gegebener oder erst noch zu schaffender Phänomene, Dinge, Ereignisse, Handlungen, Prozesse oder Sachverhalte, etc., die in einem gegebenen Kontext bestimmten Personen bei der Verfolgung bestimmter Ziele dienen.

Stachowiak [Stac73] sieht ein *Modell* als Abbild oder Vorbild eines Originals. Das *Original* ist ein Ausschnitt der realen Welt oder wiederum selbst ein Modell. Der Prozess der Erstellung eines Modells wird *Modellierung* oder *Modellbildung* genannt.

Stachowiak untersucht in [Stac73] den Begriff des Modells eingehend und nennt drei Merkmale, welche jedes Modell aufweist: das *Abbildungsmerkmal*, das *Verkürzungsmerkmal* und das *pragmatische Merkmal*. In den nachfolgenden Kapiteln (4.2, 4.3 und 4.4) werden diese Merkmale näher erläutert.

4.2 Abbildungsmerkmal

> *… Modelle sind stets Modelle von etwas, nämlich Abbildungen, Repräsentationen natürlicher oder künstlicher Originale, die selbst wieder Modelle sein können. [Stac73]*

Ein Modell ist eine Abbildung eines natürlichen oder künstlichen Originals. Die Abbildung legt hierbei fest, welche Eigenschaften des Originals auf welche Art und Weise im Modell repräsentiert werden. Ein Modell ist somit eine Repräsentation ganz bestimmter Eigenschaften des Originals, wobei das Original wiederum selbst ein Modell eines anderen Originals sein kann (und oft auch ist).

Die mitunter unvollständige Darstellung eines Spezifikationsmodells auf einem Bildschirm ist ein Beispiel hierfür. Die Abbildungsvorschrift legt hierbei fest, welcher Ausschnitt des Spezifikationsmodells wie auf dem Bildschirm darzustellen ist.

Was in einem bestimmten Kontext als Modell dient, kann in einem anderen Kontext als Original dienen. Die Abbildung legt hierbei fest, welche Eigenschaften des Originals im Modell auf welche Art und Weise wiedergegeben werden.

> *… Zwischen Original und Modell lässt sich mit dieser Auffassung des Modellbegriffs nicht per Definition, losgelöst vom jeweiligen Kontext, eine simple Trennlinie ziehen. [Luft84]*

Sowohl Original wie auch Modell sind nach Stachowiak *Systeme*[1]. Abgebildet werden Objekte und Relationen[2] eines Systems auf Objekte und Relationen eines anderen Systems. Das abzubildende System hat eine bestimmte Rolle, nämlich die des Originals. Das System, auf welches abgebildet wird, nimmt die Rolle des Modells ein.

Steht eine bestimmte *Notation* zur Verfügung, um ein Modell zu beschreiben, so spricht man auch von einer *Modellierungssprache*. Die von einer solchen Modellierungssprache bereitgestellten Modellelemente, also die Objekte und die (gültigen) Relationen dieser Objekte untereinander, bezeichnet man als *Syntax* der Sprache. Die Beziehungen der Modellelemente zu dem Originalgegenstand, den sie bezeichnen – also die Bedeutung der Symbole – bilden die *Semantik* der Sprache (siehe Abb. 1).

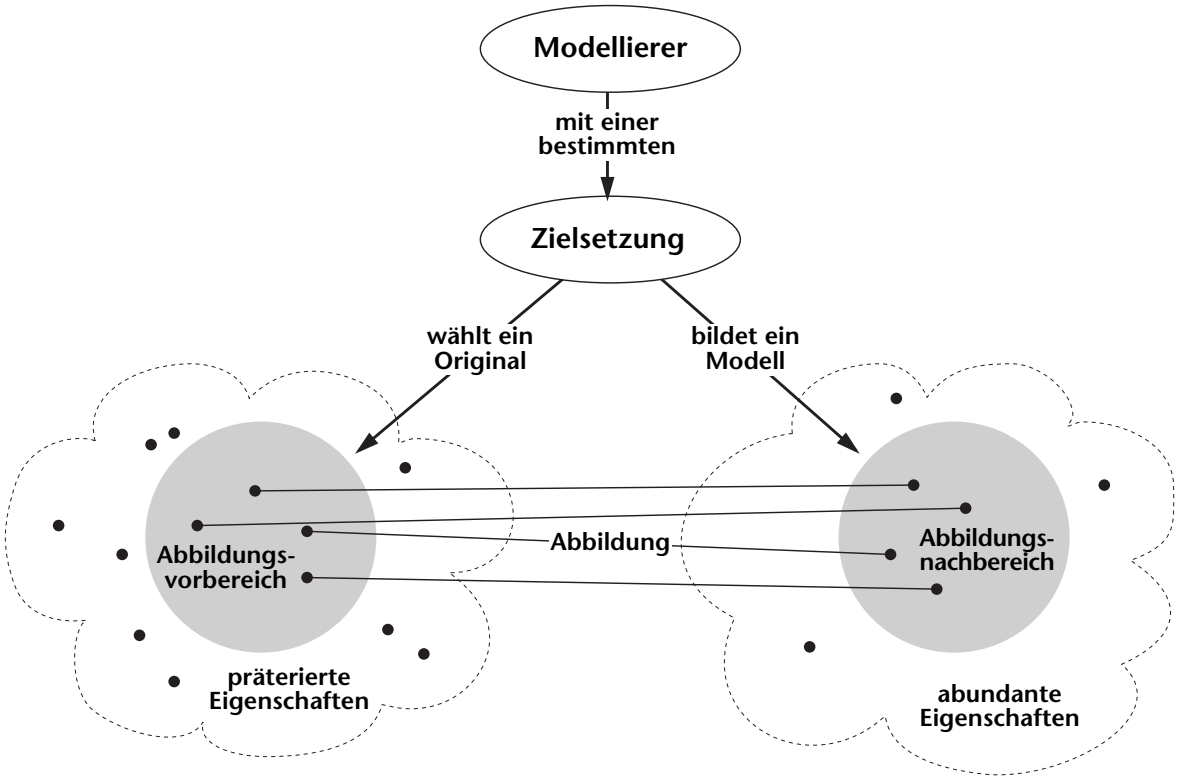


Abbildung 1: *Modell als Abbildung mit entsprechender Verkürzung und Pragmatik. Abgebildet werden die Eigenschaften (als schwarze Punkte dargestellt) des Abbildungsvorbereichs in den Abbildungsnachbereich. Nicht abgebildet werden die sog. präterierten Eigenschaften. Hinzu kommen die sog. abundanten Eigenschaften (siehe Kapitel 4.3).*

---

1 Eine in sich geschlossene Gruppe von Dingen mit bestimmten Abhängigkeiten, Beziehungen, Interaktionen zwischen diesen Dingen, welche zusammen ein Ganzes formen.

2 Hiermit sind nicht die Objekt- und Relationenbegriffe der objektorientierten Modellierung gemeint, sondern vielmehr die natürlichsprachliche Bedeutung von Objekt und Relation.

4.3 Verkürzungsmerkmal

> *… Modelle erfassen im allgemeinen nicht alle Objekte und Relationen des durch sie repräsentierten Originals, sondern nur solche, die dem jeweiligen Modellerschaffer und/oder -benutzer relevant scheinen. [Stac73]*

Das Modell ist gegenüber seinem Original (immer) *verkürzt*, d.h. nicht alle Eigenschaften des Originals – Entitäten bzw. Objekte sowie Relationen – werden abgebildet und finden im Modell eine entsprechende Repräsentation. Bedingt durch die Modellierung kann ein Modell aber auch neue, artifizielle Eigenschaften aufweisen, also Eigenschaften, die sich im Original so nicht finden.

Diejenigen Objekte und Relationen, welche im jeweiligen Betrachtungskontext nicht relevant sind (siehe Kapitel 4.4) und daher nicht abgebildet werden, nennt Stachowiak *präteriert*. Objekte und Relationen des Modells ohne Entsprechung im Original, nennt Stachowiak *abundant* (siehe auch Abb. 1).

Nimmt man abermals die Darstellung eines Spezifikationsmodells auf einem Bildschirm als Beispiel, so ist die Farbe des Bildschirmhintergrunds hier eine abundante Eigenschaft. Das Original – das Spezifikationsmodell und nicht seine Darstellung – wird diese Eigenschaften in aller Regel nicht aufweisen, sondern sie kommt bedingt durch die Abbildungsvorschrift neu hinzu. Wenn bedingt durch die Grösse des Bildschirms nicht alle Modellelemente in der aktuellen Darstellung angezeigt werden können, so sind dies – bezogen auf diese Darstellung – präterierte Attribute.

4.4 Pragmatisches Merkmal

> *… Modelle sind ihren Originalen nicht per se eindeutig zugeordnet, sie erfüllen ihre Ersetzungsfunktion a) für bestimmte – erkennende und/oder handelnde, modellbenutzende – Subjekte, b) innerhalb bestimmter Zeitintervalle und c) unter Einschränkung auf bestimmte gedankliche Operationen.*
>
> *… über die abbildungsmässige Originalbezogenheit hinaus ist mithin der allgemeine Modellbegriff dreifach pragmatisch zu relativieren. Modelle sind nicht nur Modelle von etwas. Sie sind auch Modelle für jemanden, einen Menschen oder einen künstlichen Modellbenutzer. Sie erfüllen dabei ihre Funktion in der Zeit, innerhalb eines Zeitintervalls. Und sie sind schliesslich Modelle zu einem bestimmten Zweck. Man könnte diesen Sachverhalt auch so ausdrücken. Eine pragmatisch vollständige Bestimmung des Modellbegriffs hat nicht nur die Frage zu berücksichtigen, wovon etwas Modell ist, sondern auch, für wen, wann und wozu bezüglich seiner je spezifischen Funktionen es Modell ist. [Stac73]*

Modelle sind keine objektiven Abbilder eines Originals. Sie werden für bestimmte Nutzer zur Untersuchung ganz bestimmter Fragestellungen erstellt. Ein Modell ist nicht einfach nur ein Modell, sondern stets ein Modell für jemanden, zu einer bestimmten Zeit und unter Einschrän-

kung auf bestimmte gedankliche oder tatsächliche Operationen; es dient also einem ganz bestimmten Zweck. Welche Objekte und Relationen jeweils relevant sind, hängt immer davon ab, wer das Modell verwendet, wann das Modell verwendet wird und für welchen Zweck – zum Beispiel Kommunikation, Komplexitätsreduktion, Modellexperiment, Visualisierung, etc. – das Modell dienen soll.

Wie bereits angedeutet, kann einunddasselbe Modell, je nach intendierter Nutzung, zu einem bestimmten Zeitpunkt oder in einem bestimmten Kontext Abbild, zu einem anderen Zeitpunkt oder in einem anderen Kontext Vorbild sein. Abbilder werden hierbei nach [Lude89] sowie [Lich93] auch als *deskriptive Modelle*, Vorbilder auch als *präskriptive Modelle* bezeichnet.

5 ADORA-L

Dieses Kapitel gibt eine kurze Einführung in die Spezifikationssprache ADORA-L. Vorgestellt werden zentrale Ideen und Sprachkonzepte von ADORA-L. Anhand dieser werden dann die entsprechenden Sprachkonstrukte eingeführt, die das jeweilige Konzept unterstützen. Zum besseren Verständnis folgt stets ein Beispiel.

Ziel des Kapitels ist es, ADORA-L soweit zu erläutern, wie es für das Verständnis dieser Arbeit notwendig ist. Eine ausführliche Beschreibung von ADORA-L findet sich in [Joos99].

5.1 Sprachkonzepte von ADORA-L

Die unterliegenden Sprachkonzepte bestimmen grundlegend das Erscheinungsbild einer Sprache. Für ADORA-L werden in [Joos99] u.a. die folgenden, zentralen Konzepte genannt:

- Objektorientierung und Modellierung auf der Ebene abstrakter Objekte (siehe Kapitel 5.2).
- Verwendung eines hierarchischen und integrierten Gesamtmodells (siehe Kapitel 5.3).
- Objektkommunikation durch Versenden von Nachrichten (siehe Kapitel 5.4 und 5.5).
- Integration der Verhaltensbeschreibung in das hierarchische Objektmodell (siehe Kapitel 5.5 und 5.6).
- Beschreibung der Objektfunktionalität in variablem Formalisierungsgrad (siehe Kapitel 5.7).
- Ein zum Objektmodell orthogonal positioniertes Klassenmodell mit einer Klassenhierarchie im Sinne einer Generalisierungsabstraktion (siehe Kapitel 5.8).

In den nun folgenden Kapiteln (5.2 - 5.8) wird auf diese Konzepte sowie deren sprachliche Umsetzung und Integration in ADORA-L eingegangen.

5.2 Objektorientierung und abstrakte Objekte

ADORA-L ist eine objektorientierte Sprache. Diesem Paradigma folgend, wird ein System grundsätzlich als ein Netzwerk eigenständiger, interagierender Objekte betrachtet, welche untereinander mittels Nachrichten kommunizieren und so die Ausführung von Operationen bzw. Methoden anstossen. Operationen werden als Reaktion auf das Eintreffen einer Nachricht ausgeführt. Die Zustandsveränderung eines Objekts ist abhängig von der Semantik der auszuführenden Operation, welche je nach aktuellem Objektzustand unterschiedlich sein darf. Somit *kann* die Ausführung einer Operation zu einer Veränderung des Objektzustandes führen. Die Ausführung einer Operation führt jedoch nicht zwingend zu einer Zustandsänderung.

5.2.1 Sprachkonstrukte

Um ein System mittels ADORA-L adäquat zu beschreiben, muss der Modellierer die Objekte des zu modellierenden Systems bestimmen, jedem dieser Objekte bestimmte Verantwortlichkeiten und Dienste zuordnen sowie die Kommunikation dieser Objekte untereinander beschreiben. Ebenso muss der Modellierer diejenigen Objekte identifizieren, welche die Systemgrenze bilden, d.h. die Objekte – insofern diese vorhanden und für den Modellierer identifizierbar sind – die das System bedienen, steuern, treiben, überwachen, etc. sowie die Objekte, auf deren bereits vorhandener Funktionalität sich das System abstützt.

Die primären Sprachkonstrukte, welche hierfür zur Verfügung stehen sind:

- das *abstrakte Objekt*
- die *abstrakte Objektmenge*
- der *externe Akteur*
- das *externe Objekt*.

Das aus diesen Elementen entstehende Modell ist ein *Objektmodell*. ADORA-L kennt zwar auch Klassen bzw. Typen und Klassenmodelle, jedoch steht die Modellierung derselben nicht im Mittelpunkt. Im Unterschied zu vielen anderen Ansätzen – wie beispielsweise UML [Rum+99], OPEN [Fir+98], ROOM [Sel+94], etc. – ist in ADORA-L ein Objektmodell und nicht ein Klassenmodell das zentrale Modell.

In ADORA-L wird konzeptionell unterschieden zwischen der Modellierung *extensionaler*, kontextabhängiger Aspekte und *intensionaler*, kontextunabhängiger Aspekte. Im ADORA-L Objektmodell werden extensionale Aspekte – wie Kommunikation, Interaktion, Beziehungen des Objekts, etc. – auf Basis abstrakter Objekte bzw. Objektmengen beschrieben. Die Beschreibung der intensionalen Aspekte – also der gemeinsamen Eigenschaften aller Objekte desselben Typs – erfolgt durch Typen, Klassen, Objektschablonen, etc. im sogenannten ADORA-L Typverzeichnis (siehe Kapitel 5.8 oder [Joos99]).

Dahinter steckt die Erkenntnis, dass für ein grundlegendes Systemverständnis die extensionalen Aspekte wesentlich sind. Die Modellierung dieser Aspekte mittels abstrakter Objekte führt zu ausdrucksstärkeren Modellen und ermöglicht eine bessere, da weitgehend anomalienfreie, Systemkomposition [Joo+97].

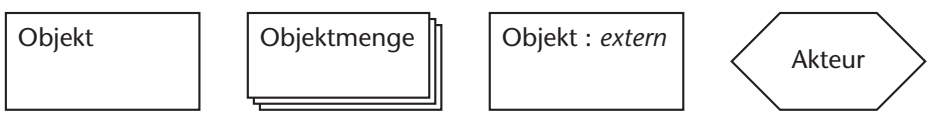


Abbildung 2: *Konkrete Syntax für ein abstraktes Objekt und eine abstrakte Objektmenge sowie ein externes Objekt und einen externen Akteur. Analog zur abstrakten Objektmenge gibt es in ADORA-L auch externe Objekt- und Akteurmengen (diese sind hier nicht abgebildet).*

Sprachkonstrukt: Abstraktes Objekt (siehe Abb. 2)

Ein *abstraktes Objekt* ist ein Platzhalter bzw. Repräsentant einer konkreten Objektinstanz.

Ein abstraktes Objekt definiert die Attribute, die Funktionalität und das Verhalten des Objekts. Abstrahiert wird von der eigentlichen Identität des Objekts, d.h. es bleibt offen, um welche konkrete Objektinstanz (kurz Instanz) es sich genau handelt. Da offen bleibt, um welche Objektinstanz es sich handelt, dürfen auch bestimmte Attributwerte undefiniert bleiben. Es muss 'lediglich' sichergestellt sein, dass diejenigen Attributwerte, die notwendig sind, damit das Objekt seiner Rolle im spezifizierten System gerecht werden kann, entsprechend sinnvoll belegt sind.

Anmerkung: Wenn im Folgenden von Objekten die Rede ist, so sind immer abstrakte Objekte gemeint. Überall wo dies nicht der Fall ist, wird konkret darauf hingewiesen.

Sprachkonstrukt: Abstrakte Objektmenge (siehe Abb. 2)

Analog zum abstrakten Objekt, welches einen Repräsentanten einer konkreten Objektinstanz darstellt, ist eine *abstrakte Objektmenge* ein Platzhalter bzw. Repräsentant einer Menge von konkreten Objektinstanzen der gleichen Klasse. Oder einfacher: eine abstrakte Objektmenge ist einen Menge von abstrakten Objekten derselben Klasse. Die Grösse einer abstrakten Objektmenge ist variabel, d.h. es können ggf. Objekte hinzugefügt oder entfernt werden.

Sprachkonstrukt: Externes Objekt (siehe Abb. 2)

Ein *externes Objekt* repräsentiert ein eigenständiges System, welches mit dem Systemmodell – dem eigentlichen System – interagiert. Ein externes Objekt ist Bestandteil und somit auch Objektkomponente des Systemmodells. Es verhält sich nach aussen genauso wie ein abstraktes Objekt, nur dass die Interna des externen Objekts nicht (in dem Systemmodell, in welchem sie verwendet werden) modelliert sind.

Sprachkonstrukt: Externer Akteur (siehe Abb. 2)

Ein *externer Akteur* repräsentiert ein eigenständiges externes System – beispielsweise eine Person – welches mit dem Systemmodell interagiert. Ein externer Akteur beschreibt die Kommunikation des Akteurs mit dem durch das ADORA-L-Modell spezifizierten System.

Sprachkonstrukte: Externe Akteurmenge und Externe Objektmenge (ohne Abb.)

Die *externe Objektmenge* und die *externe Akteurmenge* seien hiermit der Vollständigkeit halber noch erwähnt. Analog zur abstrakten Objektmenge gibt es in ADORA-L auch noch eine *externe Objektmenge* und eine *externe Akteurmenge*. Diese sind sinngemäss entsprechend der abstrakten Objektmenge definiert. Auf diese Sprachkonstrukte wird hier nicht weiter eingegangen; für Details sei auf [Joos99] verwiesen.

5.2.2 Beispiel Taschenrechner

Als durchgängiges Beispiel zur Darstellung von ADORA-L dient ein einfacher Taschenrechner (Calculator). Der Taschenrechner bzw. dessen Rechenwerk (Engine) soll Fliesskommazahlen verarbeiten und die vier Grundrechenarten (+, –, x, ÷) beherrschen. Da es ein einfacher Taschenrechner sein darf, muss Punkt-Vor-Strich nicht beachtet werden.

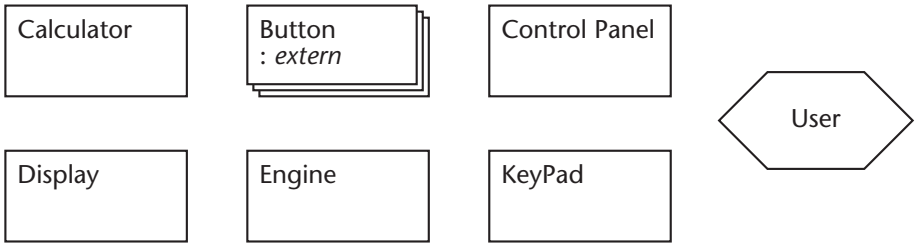


Abbildung 3: *Die Objekte des Taschenrechners: Calculator, ControlPanel, Display, Engine und KeyPad sind abstrakte Objekte. Button ist ein externes Objekt (bzw. eine externe Objektmenge) und User ist ein externer Akteur.*

Der Taschenrechner verfügt über ein Bedienfeld (ControlPanel), bestehend aus einer einzeiligen Anzeige (Display) und einer Tastatur (KeyPad) mit entsprechenden Tasten (Button) zur Eingabe von Zahlen und Operationen. Die Tasten sind bereits gegeben und müssen nicht mehr entwickelt werden. Bedient wird der Taschenrechner vom externen Akteur User über die Tastatur. Abb. 3 zeigt die abstrakten Objekte bzw. die Objektmenge, aus welchen dieser Beispielrechner bestehen soll.

5.3 Verwendung eines hierarchischen und integrierten Gesamtmodells

In ADORA-L wird ein System durch ein einziges Gesamtmodell beschrieben. Alle zu modellierenden Aspekte – wie beispielsweise Struktur, Verhalten, etc. – sind in dieses Gesamtmodell integriert. Praktisch umgesetzt bedeutet dies, dass in einer *Basisstruktur*, welche die grundlegende Struktur des Systems vorgibt, Aspekte wie Verhalten und Funktionalität ergänzt werden. Sämtliche Aspektbeschreibungen sind in diese Basisstruktur integriert und werden nicht als konzeptionell separate Modelle oder Sammlung von Diagrammen modelliert. Für den Modellierer hat dies den Vorteil, dass insbesondere Zusammenhänge und Wechselwirkungen zwischen den Aspektbeschreibungen explizit sichtbar werden. Hiervon verspricht man sich generell robustere und konsistentere Modelle.

5.3.1 Aspektbezogene Einblendungen

Um in ADORA-L Aspekte wie Struktur, Verhalten, Funktionalität, etc. zu beschreiben, gibt es sogenannte *aspektbezogene Einblendungen*. Zur Formulierung einer jeden Einblendung ist eine bestimmte Menge an Sprachkonstrukten vorgesehen. Die mittels dieser Konstrukte formulierte Einblendung wird dann an der entsprechenden Stelle in der Basisstruktur platziert (siehe Kapitel 5.4). ADORA stellt derzeit folgende Einblendungen bereit [Joos99]:

- Mittels der *strukturellen Einblendung* werden die *statische Struktur* des Systems, die *Benutzungsbeziehungen* der Systemkomponenten sowie der *Zusammenhang zwischen System und Systemgrenzen* modelliert. Ein System wird hierbei als Dekomposition von aktiven, eigenständigen Objekten interpretiert, welche sich wechselseitig benutzen (siehe Kapitel 5.5). Beziehungen stellen hierbei u.a. auch Kommunikationskanäle zwischen den Objekten dar.
- Die *verhaltensorientierte Einblendung* beschreibt das (zustandsabhängige) *Systemverhalten*. Grundlage für die Beschreibung des Systemverhaltens sind – an den Harel'schen Statecharts [Hare87] orientierte – hierarchische Zustandsautomaten. Für jedes Objekt kann eine entsprechende Verhaltensbeschreibung angegeben werden. Die Komponenten eines Objekts werden ggf. als eingeschachtelte Zustände interpretiert (siehe Kapitel 5.6).
- Die (verhaltensunabhängige) Funktionalität eines Objekts kann durch die *funktionale Einblendung* beschrieben werden. Grundlage ist die Spezifikation der Klassenzugehörigkeit sowie die damit verbundenen Operationen. Operationen eines Objekts werden als Dienste interpretiert, die das Objekt anbietet und welche von anderen Objekten entsprechend in Anspruch genommen werden können (siehe Kapitel 5.7).
- Use Cases und Szenarien können über die sog. Interaktionseinblendung[1] beschrieben werden.
- Im sogenannten *Typverzeichnis* werden Klassen, Stereotypen und Datentypen spezifiziert. Hier werden auch Generalisierung/Spezialisierungszusammenhänge zwischen Klassen beschrieben. Das Typverzeichnis ist orthogonal zur Basisstruktur (siehe Kapitel 5.8).

5.4 Basisstruktur – Systemdekomposition

Wie in Kapitel 5.3 bereits erwähnt, bildet die sogenannte Basisstruktur das Skelett eines ADORA-L-Modells. Es ist offensichtlich, dass der Basisstruktur eine besondere Bedeutung zukommt. Sie muss geeignet sein, sämtliche Aspektbeschreibungen aufzunehmen und zu einem konsistenten, geordneten Systemmodell zu verbinden. Eine geeignete Wahl der Basisstruktur fördert die Erstellung gut strukturierter, verständlicher Modelle. Insbesondere die Verständlichkeit grosser Modelle ist davon abhängig, ob zur Gliederung der Basisstruktur entsprechende Abstraktionsmechanismen vorhanden sind.

Die Basisstruktur in ADORA-L ist eine durch Teil-Ganzes-Beziehungen gegliederte Objekthierarchie. In einer Teil-Ganzes-Objekthierarchie werden Objekte in eine baumartige Hierarchie

[1] Dieser Arbeit ist der Sprachumfang Stand [Joos99] zugrunde gelegt. Die Interaktionseinblendung ist neueren Datums (siehe [Gli+01]). Daher wird auf diese Einblendung wie auch auf andere neuere Sprachkonstrukte bzw. Änderungen nicht weiter eingegangen.

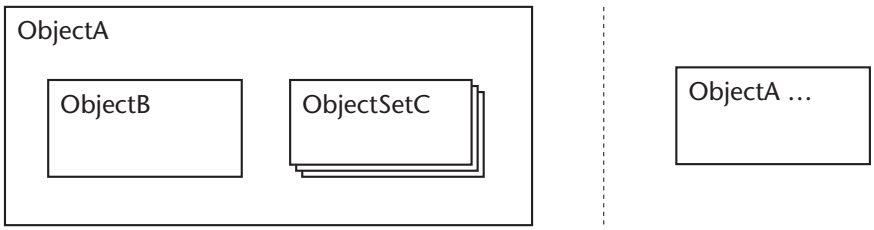


Abbildung 4: *Konkrete Syntax für die Teil-Ganzes-Beziehung.*
Die Objekte ObjectB und ObjectSetC sind Komponenten von ObjectA (linke Seite). Ferner wird eine vergröberte Sicht von ObjectA gezeigt (rechte Seite), in welcher von den Komponenten von ObjectA abstrahiert wird.Wird eine Komponente vergröbert dargestellt (hier ObjectA auf der rechten Seite), so wird dies durch drei Punkte (…) nach dem Objektnamen angezeigt.

eingeordnet, in welcher ein übergeordneter Teil – die sogenannte *Komposition* – aus endlich vielen, untergeordneten Teilen besteht – den sogenannten *Komponenten*.

Die Strukturierung von Systemen bzw. die Anordnung von Objekten in einer Teil-Ganzes-Objekthierarchie realisiert einen *Abstraktionsmechanismus*, welcher die *Kompositionsabstraktion* unterstützt. Physisch ist eine Komponente Bestandteil der Komposition. Logisch beschreibt sie einen Teil der Funktionalität, des Verhaltens und/oder der Struktur der Komposition [Joos99]. Die Kompositionsabstraktion ermöglicht eine natürliche Strukturierung des Modells und erlaubt je nach Bedarf eine vergröberte oder verfeinerte Sicht auf das Modell (siehe Abb. 4).

5.4.1 Sprachkonstrukte der Basisstruktur

Sprachkonstrukt: Komponente

Um in ADORA-L auszudrücken, dass ein Objekt Teil eines anderen Objekts ist, also die Rolle einer Komponente für ein anderes Objekts einnimmt, welches dann die Rolle der Komposition innehat, wird die graphische Repräsentation der Komponente vollständig innerhalb des Kompositionsobjekts platziert (siehe Abb. 4). Konzeptionell betrachtet wird ausgedrückt, dass die Objekte zueinander in einer Teil-Ganzes-Beziehung stehen. Die so entstehende Objekthierarchie bildet die Basisstruktur eines ADORA-Modells.

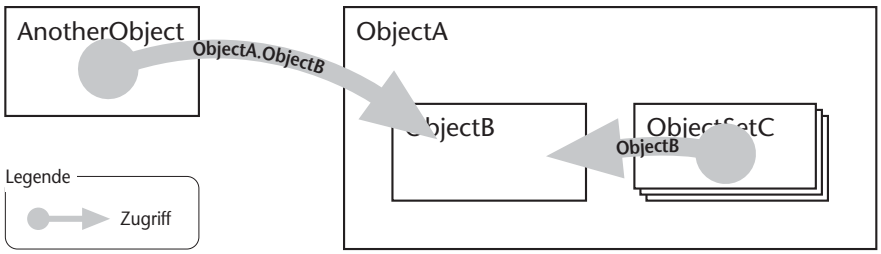


Abbildung 5: *Von den Objekten in Abb. 4 kann beispielsweise von ObjectSetC direkt auf ObjectB zugegriffen werden. Soll jedoch von ausserhalb der Komponente ObjectA auf ObjectB zugegriffen werden, so muss dies über die Angabe des Namenspfades ObjectA.ObjectB geschehen.*

Randbemerkung: In ADORA-L legt die Anordnung der Objekte in einer Teil-Ganzes-Hierarchie implizit fest, wie bestimmte Objekte ggf. direkt (und nicht über Nachrichtenkanäle, siehe Kapitel 5.6) angesprochen werden können. Sie definiert somit Sichtbarkeitsbereiche für Objektnamen bzw. -bezeichner.

Innerhalb einer Komposition können Objekte direkt über ihren Namen referenziert werden (siehe ObjectSetC in Abb. 5). Wird kein Nachrichtenkanal verwendet (siehe Kapitel 5.6) und soll von aussen (siehe AnotherObject in Abb. 5) eine der Komponenten eines Objekts adressiert werden, so muss dies über die Angabe eines kompletten Namenspfades geschehen (siehe Abb. 5).

5.4.2 Beispiel Taschenrechner

Strukturiert man die in Abb. 3 vorgestellten Objekte des Taschenrechners nach diesen Prinzipien, kommt man zu folgendem Modell (siehe Abb. 6). Dieses Modell beschreibt die statische Struktur des Taschenrechners (Calculator) so wie sie vom Modellierer gesehen und als angebracht erachtet wird.

Bereits durch diese Strukturierung des Modells bekommt der Modellbetrachter eine erste grobe Vorstellung über dessen Funktionsweise (vgl. Abb. 3). Die Strukturierung erlaubt es, bestimmte Details verfeinert darzustellen oder von diesen zu abstrahieren. So sind beispielsweise in Abb. 6 die Tastatur und das Rechenwerk vergröbert dargestellt. Stellt man auch diese Komponenten verfeinert dar, so wird ihre innere Struktur sichtbar. Deckt man beispielsweise die innere Struktur der Tastatur auf (nicht abgebildet), so werden die Button-Objekte (vgl. Abb. 3) sichtbar, welche die Tasten der Tastatur bilden.

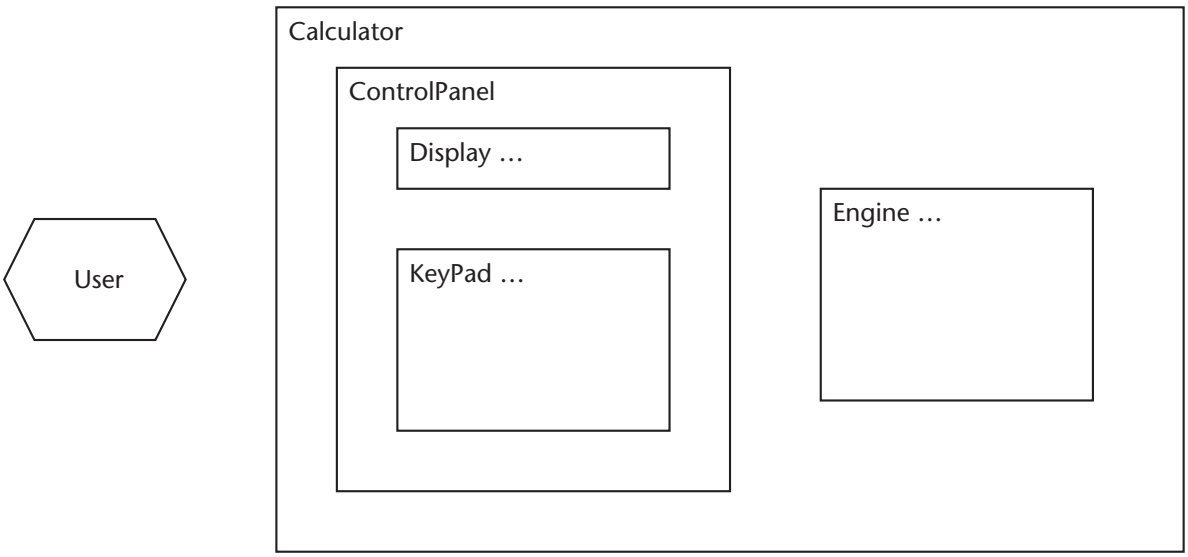


Abbildung 6: *ADORA-L Basismodell des Taschenrechners.*
Der Taschenrechner besteht aus einem Bedienfeld (ControlPanel) und einem Rechenwerk (Engine).
Das Bedienfeld besteht aus der Anzeige (Display) und der Tastatur (KeyPad).
Die Anzeige und Tastatur sowie das Rechenwerk sind hier vergröbert dargestellt. So wurde beispielsweise bei der Darstellung der Tastatur von den einzelnen Tasten abstrahiert.

5.5 Strukturelle Einblendung – Objektkommunikation

Beziehungen in der strukturellen Einblendung ergänzen und erweitern die Konstrukte der Basisstruktur. Beziehungen beschreiben nach [Joos99] die gegenseitige Referenzierbarkeit der in Beziehung gesetzten Objekte oder Objektmengen, d.h. es wird festgelegt, dass ein Objekt Kenntnis eines anderen Objekts hat. Im einzelnen kann dies bedeuten:

- Es existiert ein Informationsfluss zwischen den Objekten, repräsentiert durch Nachrichten, welche zwischen den beteiligten Objekten versendet oder empfangen werden. Eine Beziehung lässt sich hier als eine Art Nachrichtenkanal interpretieren, über welchen Sender und Empfänger einander bekannt gemacht werden und entsprechend Nachrichten versenden und empfangen können.
- Es existiert ein Informationsfluss zwischen den Objekten, ohne dass Nachrichten versendet werden, z.B. ein direkter Zugriff auf (öffentliche) Attribute eines anderen Objekts. Eine Beziehung in diesem Sinne ist vergleichbar mit einer Beziehung wie sie von Entity-Relationship-Diagrammen [Chen76] her bekannt ist. Die Beziehung steht hier für das Vorhandensein entsprechender Zugriffsrechte.

Objekte kommunizieren in ADORA-L über Nachrichten. Eine solche Kommunikation zwischen Objekten kann generell und auch in ADORA-L sowohl statisch wie auch dynamisch modelliert werden. In der strukturellen Einblendung werden mittels Beziehungen ausschliesslich die *statischen* Kommunikations- oder Kooperationszusammenhänge des Systems spezifiziert. Diese Beschreibung bildet u.a. auch die Infrastruktur für die Modellierung der dynamischen Objektkommunikation, welche dann zusammen mit der Beschreibung des abstrakten Objektverhaltens in der verhaltensorientierten Einblendung (siehe Kapitel 5.6) erfolgt.

5.5.1 Sprachkonstrukte der strukturellen Einblendung

Es geht also zunächst um die rein *statische* Modellierung von Informationsflüssen mittels Beziehungen. Die Sprachkonstrukte, welche zur Beschreibung von Beziehungen zur Verfügung stehen sind:

- die *strukturelle Beziehung*
- die *Benutzung*.

Sprachkonstrukt: Strukturelle Beziehung **(siehe Abb. 7)**

Eine strukturelle Beziehung ist eine Beziehung, welche ausdrückt, dass zwei Objekte oder Objektmengen kommunizieren und/oder kooperieren, ohne die genaue Art und Weise der Kommunikation bzw. Kooperation näher festzulegen (siehe $\mathsf{rel}_{BD/DB}$, $\mathsf{rel}_{AD/DA}$, $\mathsf{rel}_{CF/FC}$, rel_{EG} in Abb. 7). Eine strukturelle Beziehung ist gerichtet und kann wahlweise einseitig (siehe rel_{BC} oder rel_{FE} in Abb. 7) oder gegenseitig (siehe $\mathsf{rel}_{BD/DB}$, $\mathsf{rel}_{AD/DA}$, $\mathsf{rel}_{CF/FC}$ oder rel_{EG} in Abb. 7) sein. Ist sie einseitig, so ist sie nur für eines der beiden konstituierenden Objekte sichtbar. Ist sie gegenseitig, so ist sie für beide Objekte sichtbar.

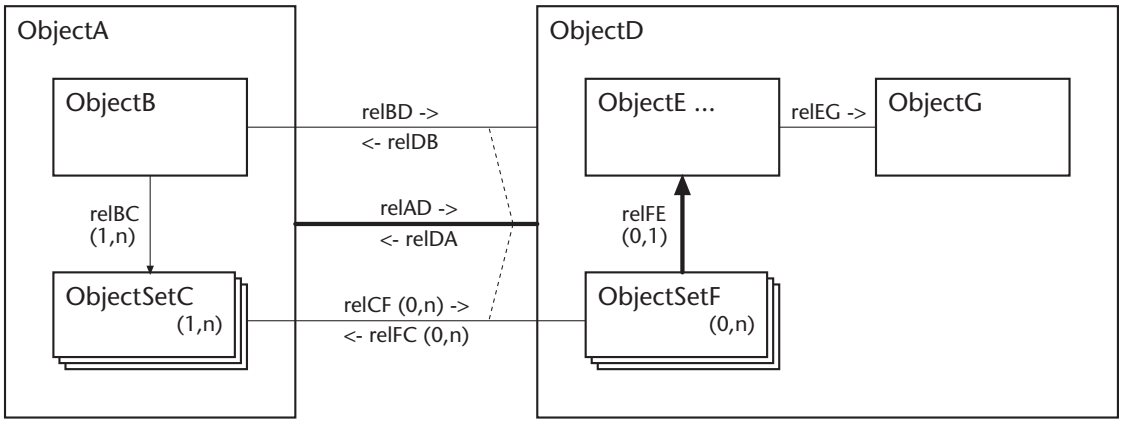


Abbildung 7: *Konkrete Syntax für Beziehungen in ADORA-L.*
Die Beziehungen rel_{BD}/rel_{DB}, rel_{AD}/rel_{DA} sowie rel_{CF}/rel_{FC} sind strukturelle Beziehungen. Rel_{BC} und rel_{FE} sind Benutzungen. Rel_{AD}/rel_{DA} ist eine Oberbeziehung für rel_{BD}/rel_{DB} und rel_{CF}/rel_{FC}. Die Benutzung rel_{FE} ist ebenfalls eine Oberbeziehung, deren Unterbeziehungen allerdings nicht dargestellt sind, da ObjectE vergröbert dargestellt ist. Rel_{EG} ist eine gerichtete strukturelle Beziehung. Bei Beziehungen mit Objektmengen sind zusätzlich Kardinalitäten annotiert (bei Beziehungen zwischen Objekten ist dies nicht notwendig).

Sprachkonstrukt: Benutzung **(siehe Abb. 7)**

Eine Benutzung ist eine einseitige Beziehung zwischen zwei Objekten/Objektmengen. Sie drückt aus, dass ein Objekt Dienste oder Leistungen eines anderen Objekts in Anspruch nimmt (Delegation). Die Benutzung ist eine Präzisierung der strukturellen Beziehung. Sie macht somit eine konkrete Aussage über die Art der Kommunikation und weist gleichzeitig den konstituierenden Objekten bestimmte, vordefinierte Rollen zu, in dem sie diese in *Auftraggeber* und *Auftragnehmer* unterteilt (siehe rel_{BC} und rel_{FE} in Abb. 7).

Nachdem nun die Sprachkonstrukte *strukturelle Beziehung* und *Benutzung* eingeführt sind, werden nun einige wichtige, generelle Eigenschaften von Beziehungen in ADORA-L erläutert.

Kardinalitäten

Wie in anderen Modellierungssprachen (UML, OML, etc.) auch können (alle) Beziehungen mit Kardinalitätsangaben ergänzt werden. Kardinalitäten geben an, wieviele Beziehungen auf Instanzebene erlaubt bzw. gefordert sind. Da auf Basis abstrakter Objekte modelliert wird (siehe Kapitel 5.2), werden in ADORA-L Kardinalitäten nicht zwischen Klassen, sondern zwischen Objektmengen oder zwischen Objektmengen und Objekten angegeben (siehe Abb. 7); bei Beziehungen zwischen Objekten ist dies nicht notwendig. Hier ist eine Beziehung grundsätzlich eine 1,1-Beziehung.

Hierachische Beziehungen

Um Beziehungen sinnvoll in die hierarchische Basisstruktur integrieren zu können, ist es notwendig, dass Beziehungen ähnlich hierarchisierbar sind wie Objekte in der Objekthierarchie.

Zu diesem Zweck gibt es in ADORA-L die Möglichkeit, für Beziehungen jeweils *Ober-* und *Unterbeziehungen* anzugeben.

Eine Oberbeziehung bildet hierbei eine Abstraktion der Objektkommunikation, indem die Beziehungen auf Detailebene (Unterbeziehungen) in vergröberter Form sichtbar gemacht werden. Analog zur Komposition, welche selbst wieder ein vollwertiges Objekt ist, ist auch eine jede Oberbeziehung selbst wieder eine vollwertige Beziehung; d.h. eine Oberbeziehung ist entweder eine strukturelle Beziehung oder eine Benutzung – selbstverständlich mit der Möglichkeit zu weiteren Oberbeziehungen.

Damit Oberbeziehungen eine echte Abstraktion ihrer Unterbeziehungen darstellen, müssen bestimmte Integritätsbedingungen erfüllt sein. Beispielsweise darf zwar von bestimmten Einzelkommunikationen abstrahiert werden, aber gleichzeitig die generelle Existenz eines Kommunikationszusammenhangs niemals komplett unterschlagen werden. An dieser Stelle werden nur die wichtigsten dieser Bedingungen genannt. Ansonsten sei auf [Joos99] verwiesen.

- *Korrekte Zuordnung der Oberbeziehungen*. Eine Beziehung $\mathsf{rel}_{\mathsf{Super}}$ ist nur dann Oberbeziehung einer anderen Beziehung $\mathsf{rel}_{\mathsf{Sub}}$, wenn mindestens eines der konstituierenden Objekte von $\mathsf{rel}_{\mathsf{Sub}}$ Komponente eines der konstituierenden Objekte von $\mathsf{rel}_{\mathsf{Super}}$ ist. Existieren mehre Möglichkeiten für eine Oberbeziehung – was durchaus vorkommen darf – so muss eine explizit ausgewählt werden. Beispiel für eine Oberbeziehung siehe Beziehung $\mathsf{rel}_{\mathsf{AD}}$ und $\mathsf{rel}_{\mathsf{FE}}$ in Abb. 7.
- *Vollständigkeit der Beziehungshierarchie*. Dies bedeutet, dass jeder Beziehung, die zwei in unterschiedlichen Kompositionen liegende Komponenten verbindet, eine Oberbeziehung zugeordnet sein muss und dass eben diese Oberbeziehung die *nächstmögliche Oberbeziehung* (Definition ‘nächstmögliche Oberbeziehung’ siehe [Joos99]) sein muss; d.h. es dürfen keine potentiellen Oberbeziehungskandidaten dazwischenliegen. Wie für die ‘korrekte Zuordnung’ (siehe oben) gilt auch hier, dass bei mehreren Möglichkeiten eine explizit ausgewählt werden muss.

5.5.2 Beispiel Taschenrechner

Durch die Hinzunahme von Beziehungen wurde die Basisstruktur des Taschenrechners (siehe Abb. 6 auf Seite 21) um eine Beschreibung der statischen Objektkommunikation ergänzt (siehe Abb. 8 auf Seite 25). Mittels dieser Beziehungen werden sowohl die statischen Kommunikationszusammenhänge im System wie auch diejenigen an der Systemgrenze mit dem eigentlichen System beschrieben.

Das dynamische Verhalten – also beispielsweise wann genau benachrichtigt das Rechenwerk wie die Anzeige, dass eine Änderung erfolgt ist – kann mit den Mitteln der strukturellen Einblendung offensichtlich nicht beschrieben werden. Eine Beschreibung dieser Aspekte erfolgt in der verhaltensorientierten Einblendung (siehe nachfolgendes Kapitel 5.6).

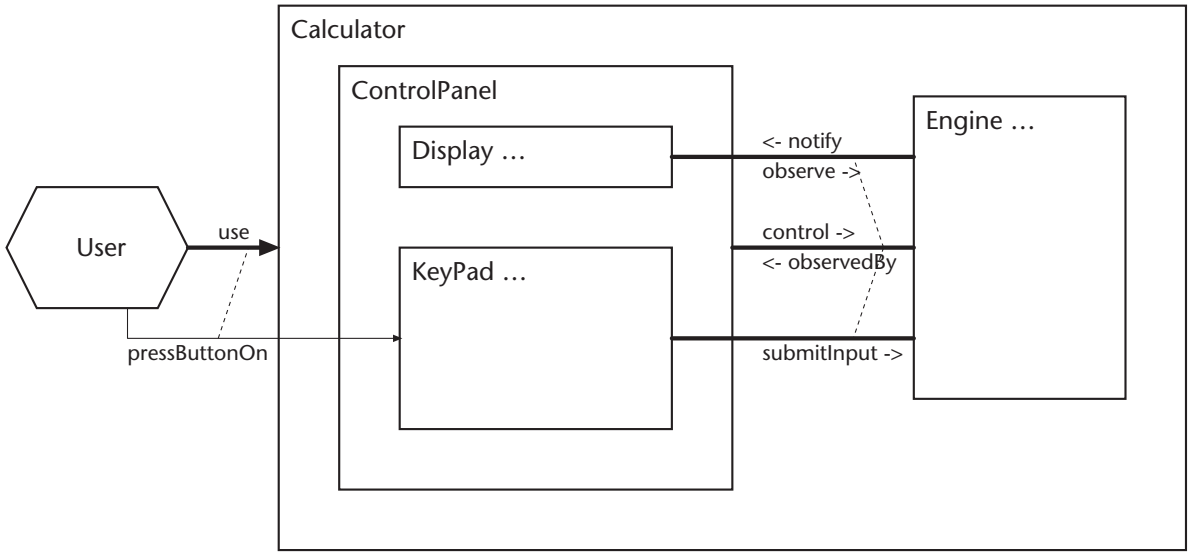


Abbildung 8: *Basisstruktur des Taschenrechners mit struktureller Einblendung. Der Taschenrechner wird vom externen Akteur User benutzt (use). Dieser bedient den Taschenrechner, indem er Tasten auf der Tastatur drückt (pressButtonOn). Zwischen Bedienfeld und Rechenwerk existiert eine strukturelle Beziehung. Das Rechenwerk wird vom Bedienfeld gesteuert (control). Zur Anzeige der Werte des Rechenwerks lässt sich dieses vom Bedienfeld beobachten (observedBy). Detailliert man diesen Kommunikationszusammenhang, indem man die Unterbeziehungen von control/observedBy betrachtet, so wird durch die gerichtete Beziehung submitInput der Sachverhalt modelliert, dass die Tastatur ihre Eingaben an das Rechenwerk weiterleitet. Das Rechenwerk reagiert auf Änderungen (Eingabe einer neuen Ziffer, durchgeführte Berechnung, etc.) dadurch, dass es die Anzeige benachrichtigt (notify). Die Anzeige ermittelt daraufhin die anzuzeigenden Werte (observe).*

5.6 Verhaltensorientierte Einblendung – Abstraktes Objektverhalten

Die verhaltensorientierte Einblendung erlaubt eine Beschreibung der dynamischen Kommunikationszusammenhänge gemeinsam mit dem abstrakten Objektverhalten. Hierfür wird in ADORA-L spezifiziert, in welchem *Zustand* ein System auf welche *Ereignisse* und unter welchen Bedingungen mit welchen *Aktionen* reagiert.

Grundlage der Verhaltensbeschreibung sind Zustandsautomaten [Hop+88], genauer gesagt an den Harel'schen Statecharts [Hare87] orientierte hierarchische Zustandsautomaten. Diese erlauben es, das abstrakte Verhalten eines Objekts diskret zu modellieren, d.h in ADORA-L wird für ein Objekt, stellvertretend für alle Objekte seiner Klasse bzw. seines Typs, das Verhalten durch einen *hierarchischen Zustandsautomaten* beschrieben.

Zustandsübergänge werden hierbei durch Ereignisse ausgelöst und können bestimmte Aktionen zur Folge haben. Ereignisse, welche einen Zustandsübergang auslösen, sind in ADORA-L i.d.R. *eingehende Nachrichten*. Es können aber auch Aussagen über *Objekt-* oder *Zeitattribute*

sein. Die auf ein bestimmtes Ereignis hin ausgelösten Aktionen geben an, welche Tätigkeiten, Aktionen, Operationen, etc. im Falle eines Zustandsübergangs ausgeführt werden. Dies kann entweder das *Versenden einer Nachricht*, die *Manipulation eines Objektattributs* oder das *Erzeugen bzw. das Löschen einer Objektinstanz* (in einer Objektmenge) sein.

In der verhaltensorientierten Einblendung wird zwar modelliert, in welchem Zustand und auf welche Ereignisse oder Nachrichten hin ein Objekt welche Tätigkeiten, Aktionen, Operationen, etc. ausführt. Jedoch werden in der verhaltensorientierten Einblendung insbesondere zustandsunabhängige Funktionalitäten nicht weiter präzisiert, d.h. die eigentliche Objektoperation, deren spezifische Vor- und Nachbedingungen, etc. werden nicht weiter beschrieben. Hierfür ist die funktionale Einblendung (siehe Kapitel 5.7) zuständig. Die verhaltensorientierte Einblendung beschreibt ausschliesslich das abstrakte, zustandsabhängige Objektverhalten einschliesslich der (Inter-)Objektkommunikation.

Der Versand bzw. das Eintreffen einer Nachricht hat in ADORA-L den Charakter eines Ereignisses. Die Nachricht selbst ist der eigentliche Träger der Information. Der in ADORA-L verwendete Nachrichtenbegriff umfasst folgende Eigenschaften:

- *Bezeichnung* – eine aussagekräftige Bezeichnung, um dem Benutzer/Leser/Modellierer/etc. eine (erste, grobe) Vorstellung von der Bedeutung der Nachricht zu vermitteln und diese von anderen Nachrichten eindeutig unterscheiden zu können.
- *Kommunikationsart* – eine Aussage über die Art und Weise wie der Nachrichtenversand zwischen Sender und Empfänger geregelt ist. In ADORA-L wird eine Nachricht *synchron*, *asynchron* oder als *asynchrone Multicast-Nachricht* versandt (siehe Unterkapitel 5.6.1.1 und 5.6.1.2).
- *Konstituenten* – spezifizieren Sender und/oder Empfänger der Nachricht. Dies kann entweder *direkt* durch Angabe des Objekts bzw. der Objekte geschehen oder *indirekt* durch die Angabe eines entsprechenden Nachrichtenkanals (siehe Kapitel 5.6.1.3).
- *Parameter* – wenn die Nachricht über ein reines Signal hinausgeht, können hier (optional) Daten in Form von Parametern angegeben werden, die mit der Nachricht versandt werden. Bei rein synchronen Nachrichten können auch Daten (unmittelbar) an den Sender zurückgegeben werden.

5.6.1 Besonderheiten bei Objektkommunikation und Nachrichtenversand

Bevor auf die eigentlichen Sprachkonstrukte der verhaltensorientierten Einblendung eingegangen wird, gilt es noch einige Eigenschaften der Kommunikation zwischen Objekten und des Nachrichtenversands zu klären. Diese Eigenschaften betreffen (a) die Synchronizität zwischen Sender und Empfänger beim Nachrichtenversand oder bei deren Beantwortung sowie (b) die Art und Weise wie beim Versand der Empfänger und beim Empfang der Sender einer Nachricht spezifiziert werden kann.

Zu (a): Die Kommunikation zwischen Objekten kann in ADORA-L entweder *synchron* oder *asynchron* erfolgen. Für eine spezielle Art der asynchronen Kommunikation mit Objektmengen gibt es sogenannte *Multicast-Nachrichten*. Auf diese Aspekte wird in den Unterkaptiteln 5.6.1.1 und 5.6.1.2 eingegangen.

Zu (b): Nachrichten können in ADORA-L entweder *direkt* oder *indirekt* versandt werden. Was dies genau bedeutet, wird im Unterkapitel 5.6.1.3 erläutert.

5.6.1.1 Synchrone vs. asynchrone Kommunikation

ADORA-L kennt zwei Arten der Kommunikation zwischen Objekten bzw. zum Versand von Nachrichten: *synchrone* und *asynchrone*.

Bei *synchroner Kommunikation* wird davon ausgegangen, dass Sender und Empfänger die gleiche Zeit und den gleichen Takt haben. Das Versenden und Beantworten einer synchronen Nachricht ist *zeitlos*. Die Nachricht kommt immer sofort beim Empfänger an. Der Empfänger reagiert mit der sofortigen, zeitlosen Ausführung einer synchronen Objektoperation. Erwartet der Sender vom Empfänger eine Antwort, so erfolgt diese sofort. Die Antwort auf eine synchrone Nachricht (beispielsweise ein bestimmter Rückgabewert) steht dem Empfänger sofort und zeitlos, also noch während des Zustandswechsels, zur Verfügung.

Zeitlos in diesem Kontext heisst nicht, dass das Versenden der Nachricht oder die Ausführung der Operation real keine Zeit verbraucht, sondern dass im Modell davon ausgegangen wird:

- dass der Versender ggf. eine Antwort erhalten muss, bevor irgendwo ein Zustandswechsel stattfindet und
- dass der Empfänger die Nachricht ggf. unmittelbar beantworten kann, d.h. unabhängig von seinem derzeitigen Zustand.

Bei *asynchroner Kommunikation* haben Sender und Empfänger nicht die gleiche Zeit bzw. den selben Takt. Nach dem Versenden einer Nachricht dürfen Zustandwechsel stattfinden. Erwartet der Sender vom Empfänger eine Rückantwort, so muss beim Sender dieses Warten auf die Antwort explizit modelliert werden. Die Abläufe bei den beteiligten Objekten sind nicht wie bei der synchronen Kommunikation implizit aufeinander abgestimmt, sondern müssen *explizit* modelliert werden. Insbesondere können Verzögerungen beim Austausch von Nachrichten entstehen (Anfrage/Rückantwort).

5.6.1.2 Multicast Nachrichten

Eine besondere Art der Kommunikation kann notwendig werden, wenn asynchron mit oder zwischen Objektmengen kommuniziert wird. Hier ist es manchmal notwendig, Nachrichten explizit an alle Objekte einer Objektmenge zu versenden oder auch darauf zu warten, von allen, oder besser genau von den Objekten einer Objektmenge eine Nachricht zu bekommen, die zum Zeitpunkt der Versendung der ursprünglichen Nachricht in der Objektmenge enthalten sind bzw. waren.

Hierfür gibt es in ADORA-L sogenannte *Multicast-Nachrichten*. Mittels Multicast-Nachrichten können Nachrichten an alle Objekte einer Objektmenge versandt werden. Ebenso kann darauf gewartet werden, von allen Objekten einer Objektmenge eine Nachricht zu bekommen.

Multicast-Nachrichten sind somit spezielle asynchrone Nachrichten. Speziell deswegen, weil automatisch eine bestimmte Art der Synchronisation zwischen Sender und Empfänger stattfindet. Zustandswechsel finden erst statt, wenn alle Nachrichten versandt oder Nachrichten von allen Objekten einer Objektmenge empfangen wurden.

5.6.1.3 Direkter vs. indirekter Nachrichtenversand

Werden Nachrichten direkt versandt, so wird das Empfängerobjekt auch direkt referenziert (siehe auch Anmerkungen in Kapitel 5.4.1). Bei indirekter Versendung von Nachrichten wird anstatt einer direkten Referenzierung ein entsprechender Nachrichtenkanal angesprochen. Wie bereits angedeutet (siehe Kapitel 5.3.1 und 5.5), werden die in der strukturellen Einblendung (siehe Kapitel 5.5) spezifizierten Beziehungen als *Nachrichtenkanäle* interpretiert. Somit kann *jede* in der strukturellen Einblendung modellierte Beziehung als Nachrichtenkanal verwendet werden.

Nachrichten sollten (mit einer Ausnahme) prinzipiell über einen durch eine Beziehung spezifizierten Nachrichtenkanal verschickt werden. Nur wenn die Zielobjekte gleichzeitig Objektkomponenten sind, dürfen Nachrichten direkt (d.h. durch direkte Referenzierung des Empfängers) versandt werden. Da keine Beziehungen modelliert werden müssen, ermöglicht die direkte Variante zwar ein rasches Erstellen von Modellen, aber da die Kommunikation quasi ‘hartverdrahtet’ ist, sind die entstehenden Modelle u.U. aufwendig zu ändern, da hier die Beschreibungen des Verhaltens umgebender Objekte abhängig vom Bezeichner des Objekts sind. Die Verwendung von Nachrichtenkanälen führt zu robusteren Modellen, da die Beschreibung des Objektverhaltens von seiner Umgebung entkoppelt wird. Solange die inanspruchgenommenen bzw. bereitgestellten Leistungen gleich sind und das Protokoll beibehalten wird, sind hier Objekte austauschbar, ohne dass umgebende Objekte geändert werden müssten.

5.6.2 Sprachkonstrukte der verhaltensorientierten Einblendung

Der besseren Übersicht wegen, werden die Sprachkonstrukte der verhaltensorientierten Einblendung in Etappen eingeführt:

- zuerst die Sprachkonstrukte für den hierarchischen Zustandsautomaten (siehe Kapitel 5.6.2.1),
- danach diejenigen für die Übergangsbedingung und die Konstrukte zur Formulierung der Übergangsaktion (siehe Kapitel 5.6.2.2).

5.6.2.1 Zustand & Zustandsübergang

Ein Zustand beschreibt einen Zeitabschnitt, in welchem ein Objekt ein bestimmtes Verhalten zeigt, d.h. eine bestimmte Konstellation des Objektdatenraums, in welcher das Objekt auf eine bestimmte Menge von Nachrichten in einer bestimmten Art und Weise reagiert. Jeder Zustand

wird durch einen eindeutigen Namen bzw. Bezeichner identifiziert. Bestimmte Zustandsarten in ADORA-L sind hierarchisierbar und erlauben vergröberte, abstrahierende Sichten auf die Verhaltensbeschreibung. In ADORA-L gibt es drei Arten von Zuständen:

- der *elementare Zustand*
- der *komplexe Zustand*
- der *Komponentenzustand*

Sprachkonstrukt: Elementarer Zustand **(siehe Abb. 9)**

Elementare Zustände (siehe state2, state4 und state5 in Abb. 9) identifizieren direkt einen eigenständigen Objektzustand. Ein elementarer Zustand wird verwendet, um einen bestimmten Zeitabschnitt im Lebenslauf eines Objekts direkt und ohne eine zusätzlich eingeschachtelte Verhaltensbeschreibung anzugeben.

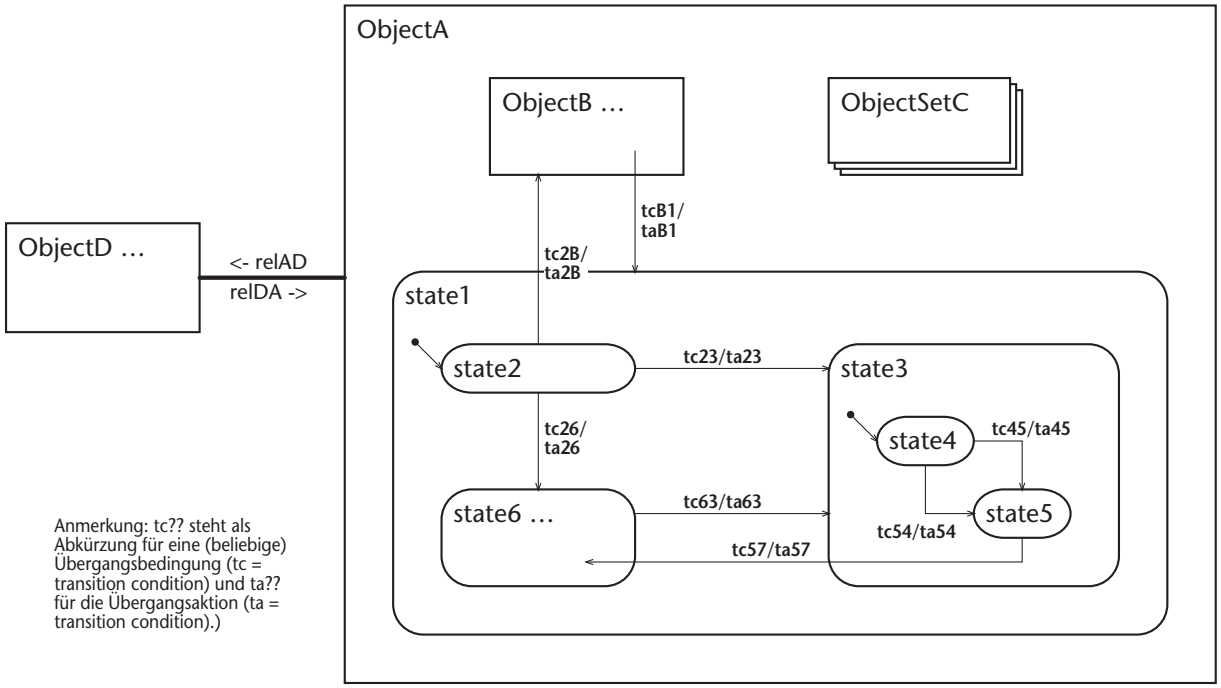


Abbildung 9: *Konkrete Syntax für Zustände in ADORA-L.*
state2, state4 und state5 sind elementare Zustände, wobei state2 und state4 Startzustände sind. State1, state3 und state6 sind komplexe Zustände. ObjectB und ObjectSetC, etc. sind Komponentenzustände. Zwischen state1 und ObjectB (hier vergröbert dargestellt) existieren Zustandsübergänge (tc_{2B}/ta_{2B} und tc_{B1}/ta_{B1}). ObjectB ist also nicht nebenläufig, sondern nur dann aktiv, wenn ObjectA nicht im Zustand state1 ist. Wird ObjectB verlassen (Zustandsübergang tc_{B1}/ta_{B1}), so verliert ObjectB seinen gegenwärtigen Zustand. Wird es wieder betreten, so wird es neu initialisiert (-> Startzustand). Zwischen ObjectA, ObjectSetC und ObjectD existieren keine Zustandsübergänge. Diese Objekte (bzw. deren Zustandsautomaten) sind zueinander nebenläufig, d.h. Sie behalten stets ihren Zustand.
Die Objekte ObjectA, ObjectB und ObjectC können Nachrichten nur direkt austauschen, da keine Beziehungen/Nachrichtenkanäle zwischen ihnen existieren. Zwischen ObjectA (incl. dessen Komponenten ObjectB und ObjectSetC) und ObjectD können Nachrichten auch indirekt über die Beziehung rel_{AD}/rel_{DA} (die dann als Nachrichtenkanal interpretiert wird) ausgetauscht werden.

Ein elementarer Zustand in ADORA-L ist in etwa vergleichbar mit einem Zustand in einem (flachen) endlichen Automaten.

Sprachkonstrukt: Komplexer Zustand (siehe Abb. 9)

Genau wie elementare Zustände auch, beschreiben komplexe Zustände (siehe state1, state3 und state6 in Abb. 9) direkt einen konkreten Objektzustand. Im Unterschied zu elementaren Zuständen sind komplexe Zustände hierarchisierbar, d.h. ein komplexer Zustand kann eingeschachtelte Zustände und damit weitere Verhaltensbeschreibungen enthalten. Mit komplexen Zuständen können umfangreiche(re) Zeitabschnitte gruppiert und unter Verwendung eingeschachtelter Zustände weiter präzisiert werden.

Ein hierarchischer Zustand ist in etwa vergleichbar mit den ‘Complex States’ der Harel’schen Statecharts [Hare87].

Sprachkonstrukt: Komponentenzustand (siehe Abb. 9)

Ein Komponentenzustand ist ein Objekt oder eine Objektmenge in der Rolle eines Zustandes (siehe ObjectB, ObjectSetC in Abb. 9). Eine solche Interpretation eines Objekts als Zustandsautomat ist direkt und einfach möglich, da in ADORA-L für jedes Objekt grundsätzlich eine entsprechende Zustandsbeschreibung vorhanden sein kann (vgl. auch [Glin93]). Komponentenzustände integrieren die Verhaltensbeschreibung in die Teil-Ganzes-Objekthierarchie der Basisstruktur. Komponentenzustände können zusätzlich durch die beiden anderen Zustandsarten ergänzt werden (siehe beispielsweise state1 in Abb. 9).

Existiert ein Zustandsübergang von einem Komponentenzustand ausgehend (siehe Zustandsübergang tc_{B1} in Abb. 9), so verliert ein Objekt oder eine Objektmenge (siehe ObjectB in Abb. 9) seinen Zustand (und damit auch sämtliche Wertebelegungen), wenn sein Komponentenzustand verlassen wird – die Lebensdauer des Objekts endet mit dem Verlassen des Komponentenzustands. Entsprechend ist es auch umgekehrt, existiert ein Zustandsübergang zu einem Komponentenzustand hin (siehe Zustandsübergang tc_{2B} in Abb. 9), so wird das Objekt neu initialisiert, wenn der Komponentenzustand betreten wird. Die Lebendauer des Objekts beginnt mit dem Betreten des Komponentenzustands.

Führen Zustandsübergänge zu einem Objekt hin bzw. von einem Objekt weg (siehe Zustandsübergänge zwischen state1 und ObjectB in Abb. 9), so ist das Objekt, bezogen auf den Ursprung der hin- oder wegführenden Übergänge, niemals nebenläufig.

Objekte oder Objektmengen können nur Nachrichten empfangen, wenn sie als Komponentenzustand aktiv sind, andernfalls verfällt die Nachricht. ObjectB in Abb. 9 beispielsweise kann keine Nachrichten empfangen, solange ObjectA im Zustand state1 ist.

Existieren keine Zustandsübergänge zwischen Komponentenzuständen (siehe ObjectA und ObjectD in Abb. 9), so sind die Objekte automatisch nebenläufig. Solche Objekte haben einen eigenen Kontrollfluss und behalten ihren Zustand (zumindest solange die Komposition existiert, deren Komponenten sie sind).

Sprachkonstrukt: Zustandsübergang **(siehe Abb. 9 & Abb. 10)**

Abbildung 10: *Zustandsübergang in ADORA-L.*

Ein Zustandsübergang in ADORA-L (siehe tc/ta in Abb. 9) definiert den Zeitpunkt und die Umstände, unter welchen ein Objekt einen gegebenen Zustand verlässt und in einen neuen Zustand übergeht sowie die Aktionen, die dieser Übergang zur Folge hat. Somit ist ein Zustandsübergang hier charakterisiert durch:

- *Quellzustand* – der Zustand, in welchem das Objekt sein muss, um bei erfüllter Übergangsbedingung in den Zielzustand zu wechseln.
- *Zielzustand* – der Zustand, in welchen das Objekt bei erfüllter Übergangsbedingung und nach ausgeführter Übergangsaktion wechselt.
- *Übergangsbedingung* – die Bedingung(en), die, wenn sich das Objekt im Quellzustand befindet, erfüllt sein müssen, damit der Zustandwechsel stattfindet.
- *Übergangsaktion* – die Aktion(en), die beim Zustandwechsel ausgeführt werden.

Zustandsübergänge werden verwendet, um eine Änderung des Objektzustandes zu beschreiben. Die Angabe der Übergangsbedingung sowie der Übergangsaktion beschreibt diskret das Objektverhalten. Die eigentlichen Übergangsbedingungen und -aktionen sind in Abb. 9 aus Übersichtlichkeitsgründen (noch) nicht näher ausgeführt, sondern nur durch tc (= transition condition) und ta (= transition action) angedeutet. Auf die Übergangsbedingung und -aktion wird im folgenden Kapitel 5.6.2.2 eingegangen.

Hingewiesen sei an dieser Stelle ausserdem auf einen notationellen Unterschied zwischen den in ADORA-L verwendeten hierarchischen Zustandsautomaten und den Harel'schen Statecharts [Hare87], auf welchen die Verhaltensbeschreibung basiert. In ADORA-L wird Nebenläufigkeit nicht explizit modelliert. Objekte bzw. Komponentenzustände sind in ADORA-L automatisch nebenläufig, wenn kein Zustandsübergang zwischen ihnen existiert (siehe auch Ausführung zum Sprachkonstrukt: Komponentenzustand sowie Abb. 9 auf Seite 29).

5.6.2.2 Übergangsbedingung & Übergangsaktion

Zustandsübergänge werden näher spezifiziert durch eine *Übergangsbedingung* und eine *Übergangsaktion*. Sowohl die Übergangsbedingung wie auch die Übergangsaktion lassen sich in ADORA-L mit variablem Formalitätsgrad – *informal*, *teilformal*, *formal* – angeben.

Unter anderem zur Spezifikation von Übergangsbedingung und -aktion steht in ADORA-L die Detailsprache ADORA-FSL[1] zur Verfügung. Hier werden die wichtigsten Teile dieser Detailsprache zur Beschreibung von Übergangsbedingung und -aktion vorgestellt.

[1] FSL steht für Formal Specification Language.

Ist ein System soweit modelliert, dass Übergangsbedingung(en) und -aktion(en) angegeben werden, ist die Spezifikation bereits auf einer recht feingranularen Ebene angelangt. Eine vollständig formale Beschreibung *sämtlicher* Aspekte[1] der Übergangsbedingung(en) und -aktion(en) *zu erzwingen*, würde meist den Aufwand und damit die Kosten für die Erstellung der entsprechenden Modelle und somit deren Nutzen nicht mehr rechtfertigen – ganz abgesehen davon, dass dies der Verständlichkeit der Modelle nicht unbedingt zuträglich wäre.

Hingegen ist es durchaus sinnvoll, diejenigen Teile der Übergangsbedingung und -aktion weitgehend formal anzugeben, welche den Versand sowie den Empfang von Nachrichten betreffen, also sozusagen die 'dynamische (Inter-) Objektkommunikation'. Die Modelle bekommen so eine brauchbare Ausführungssemantik, sind aber noch vergleichsweise schnell zu erstellen und gut verständlich. Bei der Vorstellung der Sprachkonstrukte wird daher auch schwerpunktmässig darauf eingegangen, wie die o.g. 'wichtigen' Aspekte, formal oder teilformal beschrieben werden können.

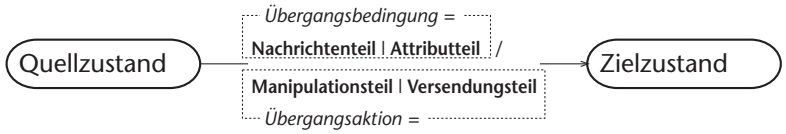


Abbildung 11: *Übergangsbedingung (tc) und -aktion.*
Die Übergangsbedingung zerfällt in Nachrichtenteil und Attributteil; die Übergangsaktion in Manipulationsteil und Versendungsteil.

Sprachkonstrukt: Übergangsbedingung (siehe Abb. 11, Abb. 12 & Abb. 13)

Die Übergangsbedingung beschreibt eine mögliche Zustandsänderung im Objekt; d.h. unter welchen Umständen (im Automaten, welcher das abstrakte Objektverhalten beschreibt) ein Zustandsübergang vom jeweiligen Quellzustand in den Zielzustand erfolgt. Diese Beschreibung zerfällt in zwei Teile (siehe Abb. 11):

- den *Nachrichtenteil* und
- den *Attributteil*.

Nachrichtenteil. Im Nachrichtenteil wird eine Nachricht spezifiziert, deren Eintreffen den entsprechenden Zustandsübergang auslöst. Spezifiziert werden hierbei der *Name*, die *Parameter* und der *Ursprung* der Nachricht. Name und Ursprung (via Nachrichtenkanal oder direkte Referenz) müssen in jedem Fall angegeben werden. Die Parameter (d.h. Anzahl, Typ und Reihenfolge) müssen denjenigen der zu empfangenden Nachricht entsprechen. Die nachfolgende Abb. 12 zeigt einige Beispiele hierfür:

---

[1] Ein Zustand stellt bereits eine bestimmte Abstraktion dar, welche eine genau bestimmte Belegung des Objektdatenraums bzw. der Werte der Objektattribute repräsentiert. Eine detaillierte, formale Beschreibung, welche Sequenz von Elementarmanipulationen am Datenraum eines Objekts einen Zustandswechsel eigentlich ausmacht, trägt oft nur wenig zu einem besserem Systemverständnis bei, da diese Manipulationen bei geeigneter Wahl und Bezeichnung der Zustände meist offensichtlich sind.

```
// (a)  Empfang einer Nachricht ohne Parameter über einen Nachrichtenkanal
receive ( messageWithoutParameters ) over channel

// (b)  Empfang einer Nachricht ohne Parameter direkt von einem Objekt
receive ( anotherMessageWithoutParameters ) from ObjectSet

// (c)  Empfang einer Nachricht mit Parametern über einen Nachrichtenkanal
receive ( messageWithParameter: p1[TypeOfP1] and: … and: pn[TypeOfPn] ) over Channel

// (d)  Empfang einer Nachricht mit Parametern von einem Objekt
receive ( anotherMessageWithParameter: p2[TypeOfP2] and: p3[TypeOfP3] ) from Object

// (e)  Empfang einer Multicast-Nachricht über einen Nachrichtenkanal
receiveAll ( mulitcastMessageWithOneParameter: p4[TypeOfP4] ) over Channel

// (f)  Empfang einer Multicast-Nachricht von einer Objektmenge
receiveAll ( mulitCastMessageWithoutParameters ) from ObjectSet

// (g)  Informale Spezifikation des Nachrichtenempfangs
-- warte auf bestimmte Nachricht von Irgendwem

// (h)  teilformale Spezifikation des Nachrichtenempfangs
receive bestimmte Nachricht from Irgendwem
```

Abbildung 12: *Beispiele für die Spezifikation des Nachrichtenempfangs in ADORA-FSL im Nachrichtenteil der Übergangsbedingung eines Zustandsübergangs.*
Abb. 12 (a) - (f) zeigen formale Spezifikationen des Nachrichtenempfangs; der Name der Nachricht sowie deren formale Parameter und der Ursprung sind formal angegeben. Dies erlaubt ggf. eine Simulation oder Animation des Objektverhaltens wie auch eine automatische Konsistenzprüfung bestimmter Teile des Modells. In Abb. 12 (a) - (c) wird ein normaler Nachrichtenempfang beschrieben während in Abb. 12 (e), (f) den Empfang einer Multicast-Nachricht gezeigt wird. Abb. 12 (g), (h) zeigen die informale bzw. die teilformale Variante; welche Nachricht empfangen wird, wird natürlichsprachlich beschrieben.

Die Beschreibung zum Empfang einer Nachricht wird in ADORA-FSL durch das Schlüsselwort receive eingeleitet. Bei Multicast-Nachrichten wird das Schlüsselwort receiveAll verwendet. Darauf folgt in Klammern der Name der zu empfangenden Nachricht. Hat die Nachricht Parameter (optional), so werden diese jeweils mit dem Bezeichner des formalen Parameters sowie dessen Typ angeführt. Der Ursprung/Versender der Nachricht wird im Anschluss angegeben.

Wird die Nachricht indirekt über einem Nachrichtenkanal empfangen, so wird, eingeleitet vom Schlüsselwort over, der Name des Kanals angegeben. Wird die Nachricht direkt von einem Objekt (und nicht über einen Nachrichtenkanal) empfangen, so wird dies eingeleitet mit dem Schlüsselwort from (und nicht mit dem Schlüsselwort over).

Der Vollständigkeit halber und zur Erinnerung sei dies hier nochmals explizit angeführt. Jede Beziehung (siehe strukturelle Einblendung, Kapitel 5.5) in ADORA-L kann für die Verhaltensbeschreibung in der verhaltensorientierten Einblendung als Nachrichtenkanal verwendet werden (siehe $rel_{AD/DA}$ in Abb. 9 auf S. 29).

Attributteil. Im Attributteil können (zusätzlich) Bedingung(en) über den Wert bestimmter Attribute und/oder Nachrichtenparameter sowie zeitbezogene Aussagen formuliert werden. Nur wenn diese Bedingungen erfüllt sind, findet auch ein Zustandswechsel statt.

Für eine formale oder teilformale Beschreibung dieser Bedingungen innerhalb von ADORA-L ist ebenfalls die Detailsprache ADORA-FSL vorgesehen, welche über die Möglichkeit zur Formulierung von Bedingungen auf prädikatenlogischer Basis verfügt. Dieser Teil von ADORA-FSL wird hier nur der Vollständigkeit wegen genannt und dementsprechend lediglich beispielhaft vorgestellt (siehe Abb. 13), da er für das grundlegende Verständnis nicht notwendig ist.

```
// Parameter p1 hat einen bestimmten Wert, nämlich den Wert 1
receive … | p1 = 1

// es existiert ein Objekt i in der Menge anObjectSet, dessen Attribut anAttribute den Wert aValue hat
receive … | ( thereExists i in anObjectSet )( i.anAttribute = aValue )

// Ampelsteuerung aus [Joos99]
receive ( deactivate: t[Time] ) over … | wait( t )
```

Abbildung 13: *Beispiele für ADORA-FSL-Ausdrücke im Attributteil der Übergangsbedingung.*

Sprachkonstrukt: Übergangsaktion (siehe Abb. 11)

Die Übergangsaktion gibt an, welche Tätigkeiten bei einem Zustandsübergang ausgeführt werden. Sie spezifiziert somit die eigentliche (Aus-)Wirkung eines Zustandsübergangs. Möglich ist hier das Versenden von Nachrichten, die Manipulation von Objektattributen sowie das Erzeugen oder Löschen von Instanzen in einer Objektmenge. Die Ausführung der Übergangsaktionen erfolgt zeitlos (siehe Kapitel 5.6.1.1).

Die Übergangsaktion besteht aus:

- dem *Manipulationsteil*
- dem *Versendungsteil*

Manipulationsteil. Im Manipulationsteil kann die Veränderung von Objektattributen beschrieben werden. Ferner können *synchrone* Nachrichten versandt und somit die Ausführung synchroner Operationen angestossen werden. Zur Erinnerung: auf synchrone Nachrichten wird vom Empfänger sofort mit Ausführung einer synchronen Operation reagiert. Hat die synchrone Operation einen Rückgabewert, so steht dieser als unmittelbares Resultat der Operation für weitere Berechnungen zur Verfügung.

Anmerkung: Der Manipulationsteil der Übergangsaktion wird wohl in der Regel informal oder teilformal angegeben. Die Tätigkeiten, welche hier angegeben werden (müssen), sind meist so elementar, dass sich eine formale Spezifikation erübrigt (siehe auch Anmerkungen zu Beginn des Kapitels 5.6.2.2). Daher werden die Möglichkeiten zur formalen Beschreibung des Manipulationsteils eher fragmentarisch vorgestellt. Eine vollständig formale Formulierung ist zwar

möglich, aber sicherlich nur dort angebracht, wo es durch das (potentielle) Risiko gerechtfertigt ist.

```
// Ausführung einer synchronen Operation zur Berechnung eines Wertes
aValue := send aSynchronousMessage to anObject

// einfache Berechnung
aValue := aValue + 1

// anschliessende Manipulation eines Attributs
anAttribute := aValue
```

Abbildung 14: *Drei Beispiele für formale Beschreibungen der Tätigkeiten im Manipulationsteil der Übergangsaktion eines Zustandsübergangs.*

Versendungsteil. Im Versendungsteil wird angegeben, welche *asynchronen* Nachrichten beim Zustandsübergang versandt werden. Die Versendung dieser Nachrichten erfolgt *nach* den Aktionen im Manipulationsteil. Im Manipulationsteil durchgeführte Berechnungen können ggf. im Versendungsteil als Nachrichtenparameter verwendet werden.

Eine besondere Rolle beim Versenden von Nachrichten spielt der *asynchrone Multicast* (siehe Kapitel 5.6.1.2). Hier werden Nachrichten an all diejenigen Objekte geschickt, welche sich zum Zeitpunkt der Versendung in der entsprechenden Objektmenge befinden.

```
// (a) indirekter Versand einer Nachricht ohne Parameter
send ( messageWithoutParameters ) over channel

// (b) direkter Versand einer Nachricht ohne Parameter
send ( anotherMessageWithoutParameters ) to Object

// (c) indirekter Versand einer Nachricht mit Parametern
send ( messageWithParameter: p1[TypeOfP1] and: … and: pn[TypeOfPn] ) over Channel

// (d) direkter Versand einer Nachricht mit Parametern
send ( anotherMessageWithParameter: p2[TypeOfP2] and: p3[TypeOfP3] ) to Object

// (e) indirekter Versand einer Multicast-Nachricht mit einem Parameter
send multicast ( mulitcastMessageWithOneParameter: p4[TypeOfP4] ) over Channel

// (f) direkter Versand einer Multicast-Nachricht an eine Objektmenge
send multicast ( mulitCastMessageWithoutParameters ) to ObjectSet

// (g) Informale Spezifikation des Nachrichtenempfangs
-- versende bestimmte Nachricht irgendwo hin

// (h) teilformale Spezifikation des Nachrichtenempfangs
send bestimmte Nachricht over Irgendwem
```

Abbildung 15: *Konkrete Syntax für den Nachrichtenversand.*
In der Reihenfolge analog zu Abb. 12 zeigen Abb. 15 (a) - (f) formale Beschreibungen für den Nachrichtenversand jeweils mit oder ohne Parameter, die entweder direkt oder indirekt versandt werden. Abb. 15 (e), (f) zeigen den Versand von Multicast-Nachrichten. Hier gibt das Schlüsselwort ***multicast*** *an, dass es sich um einen asynchronen Multicast handelt. Abb. 15 (g), (h) zeigen die informale und teilformale Variante.*

Die formale Beschreibung zum Versand einer Nachricht wird in ADORA-FSL durch das Schlüsselwort send eingeleitet. Für Multicast-Nachrichten wird send multicast verwendet. Anschliessend folgt – analog zum Empfang – der Name der Nachricht und die Parameter. Wird die Nachricht indirekt empfangen, so wird das Schlüsselwort over verwendet, bei direktem Empfang das Schlüsselwort to.

Zusammengefasst und zur besseren Übersicht ist hier (siehe Abb. 16) ein kompletter Zustandsübergang mit ausformulierter Übergangsbedingung sowie -aktion aufgeführt.

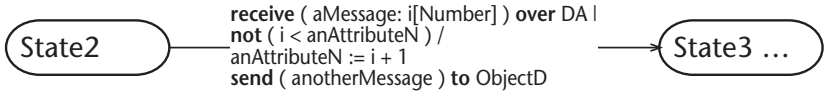


Abbildung 16: *Beispiel für die Vervollständigung des Zustandsübergangs tc_{23}/ta_{23} aus Abb. 9 auf Seite 29. Beschrieben wurde folgendes Verhalten:*
Sei ObjectA im Zustand state1. Wird nun eine Nachricht aMessage empfangen mit einem numerischen Parameter i über den Kanal rel_{DA} empfangen und ist i grösser als der Wert des Attributs anAttributeN, dann bekommt das Attribut anAttributeN den Wert i + 1 zugewiesen. Dann wird an ObjectD direkt die parameterlose Nachricht anotherMessage geschickt. Danach befindet sich das Objekt ObjectA im Zustand state3.

5.6.3 Beispiel Taschenrechner

Abb. 8 auf Seite 25 zeigt ein ADORA-L-Modell, welches die statischen Aspekte des Taschenrechners beschreibt, d.h. es zeigt die Basisstruktur des Taschenrechners mit struktureller Einblendung. Wie bereits erwähnt, können mit der Mitteln der Basisstruktur und der strukturellen Einblendung dynamische Aspekte nicht beschrieben werden.

Nun sollen unter Verwendung der Konstrukte der verhaltensorientierten Einblendung die dynamischen Aspekte modelliert werden.

Begonnen wird mit der Tastatur (KeyPad): Die Tastatur besteht aus bereits vorgegebenen Tasten. Ihr Komponentenverhalten ist einfach: in Abhängigkeit von der jeweils vom Benutzer gedrückten Taste soll dem Rechenwerk eine entsprechende Nachricht der Form übermittelt werden. Abb. 17 auf Seite 37 zeigt die verhaltensorientierte Einblendung der Tastatur. Das Verhalten der Tastatur ist nicht zustandsabhängig (vgl. Kapitel 5.2), daher gibt es auch nur einen Zustand im Automaten. Die Tastatur hat lediglich die Aufgabe, jederzeit die Benutzereingaben zu deuten und dem Rechenwerk die entsprechenden Nachrichten zukommen zu lassen. Beispielsweise übermittelt auf das Ereignis ‘Ziffertaste N1 gedrückt’ hin, die Tastatur dem Rechenwerk die Nachricht ‘Ziffer 1 eingegeben’. Die Zustandsübergänge für die Tastatur wurden formal in ADORA-FSL beschrieben.

Nun zum Rechenwerk (Engine): Das Verhalten des Rechenwerks ist im Gegensatz zur Tastatur zustandsabhängig. Das Rechenwerk wartet auf die Eingaben der Tastatur, d.h. es wartet auf das

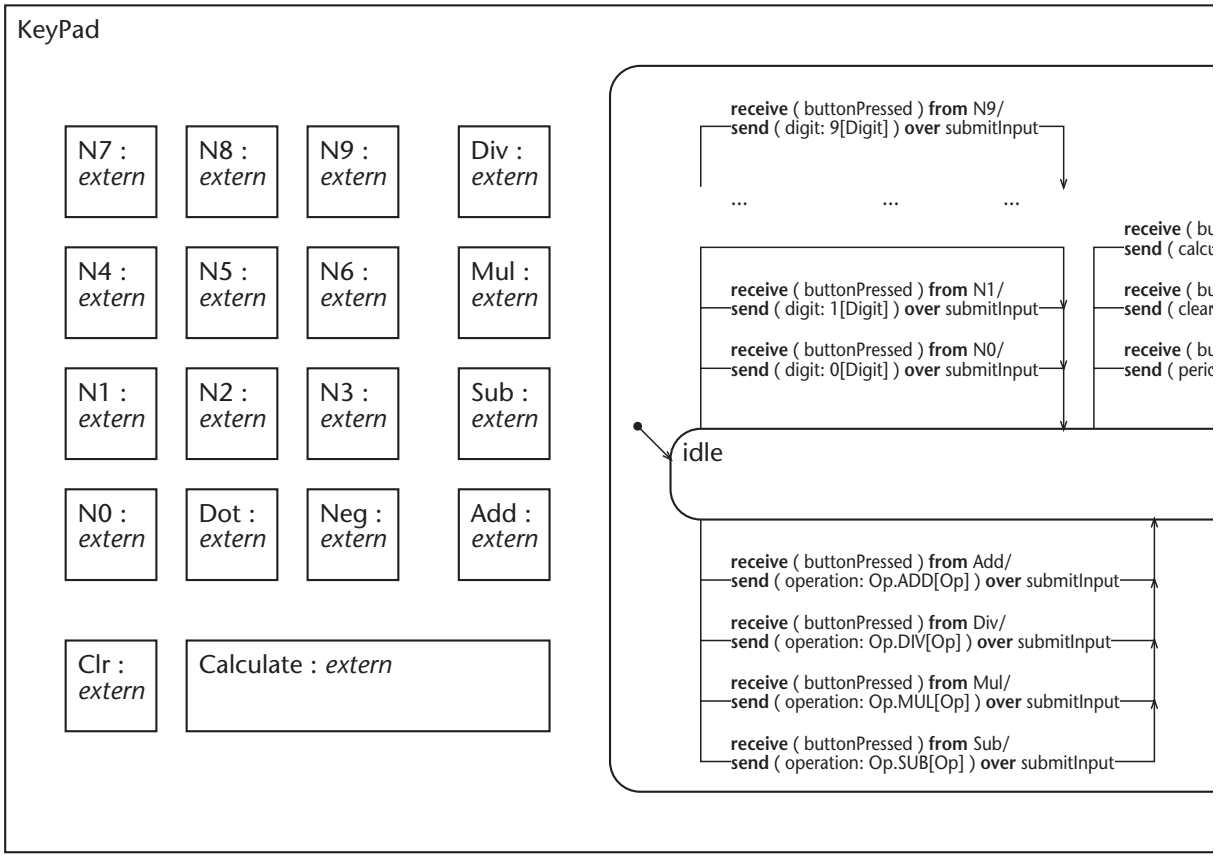


Abbildung 17: *Ausschnitt aus dem ADORA-L-Modell der Tastatur des Taschenrechners. Die rechte Seite wurde aus Platzgründen abgeschnitten. Die Tasten des Rechners sind sämtlich externe Objekte. Alle Tasten haben den gleichen Typ, jedoch haben alle Tasten einen unterschiedlichen Zweck bzw. eine unterschiedliche Rolle, wie z.B. Zifferneingabe, Berechnung, etc. Da in ADORA-L mit abstrakten Objekten und nicht mit Klassen modelliert wird, lässt sich dieser Zusammenhang direkt beschreiben. Das Verhalten der Tastatur ist einfach. In Abhängigkeit davon, welche Taste gedrückt wurde (**receive** (buttonPressed) **from** …), wird eine entsprechende Nachricht über den Kanal submitInput ans Rechenwerk geschickt. Wenn beispielsweise die Taste N1 gedrückt wird (**receive** (buttonPressed) **from** N1), so wird als Reaktion auf dieses Ereignis dem Rechenwerk mitgeteilt (**send** (digit: 1[Digit]) **over** submitInput), dass die Ziffer 1 eingegeben wurde.*

Eintreffen der entsprechenden Nachrichten. Je nachdem in welchem Zustand sich das Rechenwerk gerade befindet, akzeptiert es bestimmte Eingaben. Auch können die Eingaben in Abhängigkeit vom gerade aktuellen Zustand *unterschiedliche* Bedeutung haben; beispielsweise hat die Eingabe einer Ziffer, nachdem ein Dezimalpunkt eingegeben wurde, eine andere Bedeutung als zuvor.

Bevor hier mit der Beschreibung der Zustandsübergänge begonnen werden kann, gilt es zuerst die möglichen Zustände des Rechenwerks sowie deren Bedeutung zu identifizieren. Abb. 18 auf Seite 38 zeigt die Zustände des Rechenwerks noch ohne Zustandsübergänge.

In einem nächsten Schritt werden die Zustandsübergänge hinzugefügt. Hiermit wird festgelegt, unter welchen Umständen das Rechenwerk auf welches Ereignis wie reagiert. Zuerst erfolgt

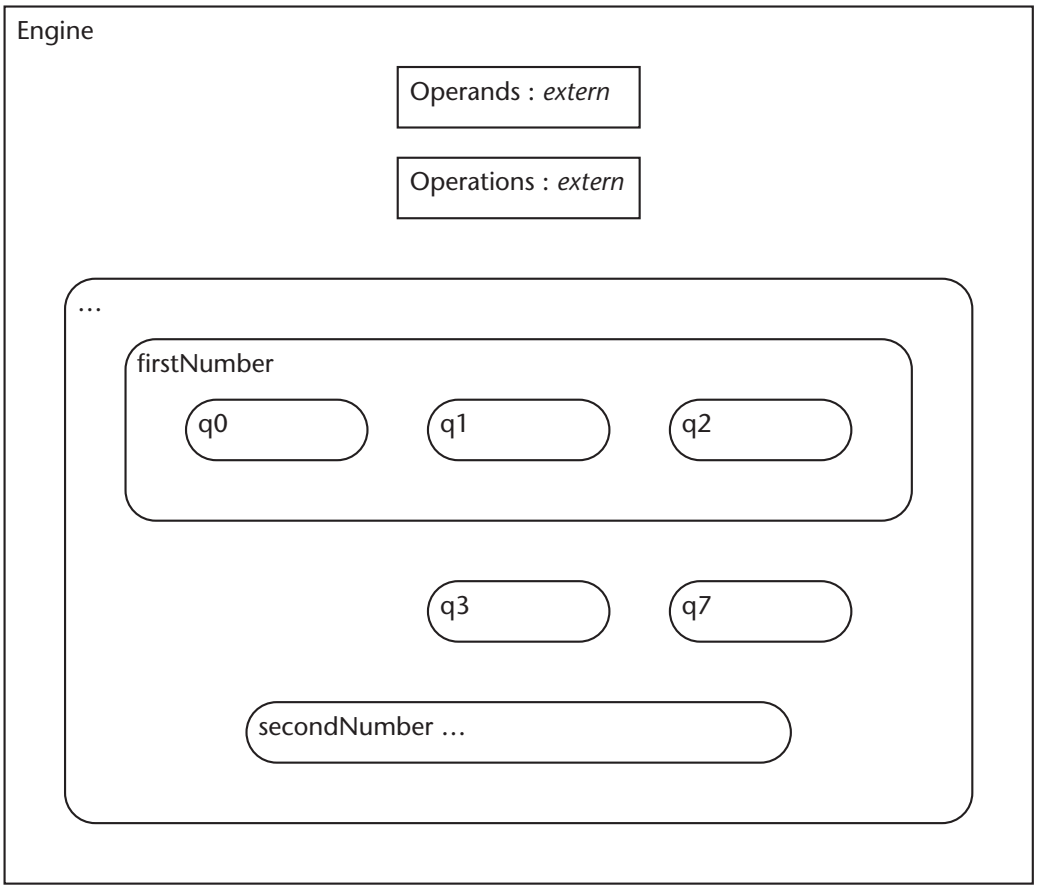


Abbildung 18: *ADORA-L-Modell des Rechenwerks des Taschenrechners. Verhaltensorientierte Einblendung noch ohne Zustandsübergänge.*
Das Rechenwerk (Engine) verfügt über die beiden Komponenten Operands und Operations. Diese sind externe Objekte, genauer gesagt Stapel/Stacks und dienen zum Speichern der eingegebenen Zahl(en) und der durchzuführenden Operation(en). Anmerkung: Stacks wären eigentlich nicht notwendig, solange der Rechner kein 'Punkt-vor-Strich' beherrscht. Für die Verhaltensbeschreibung wurden sieben elementare Zustände und zwei komplexe Zustände identifiziert. Im komplexen Zustand FirstNumber ist das Rechenwerk während die erste Zahl eingegeben wird. Für die Feinheiten der Zahleneingabe enthält der komplexe Zustand firstNumber die elementaren Zustände q0, q1, q2. Im Zustand q0 ist das Rechenwerk während die Vorkommastellen der ersten Zahl eingegeben werden. Im Zustand q1 ist das Rechenwerk solange ein Dezimalpunkt, aber noch keine Nachkommastelle eingegeben wurde. Im Zustand q2 ist das Rechenwerk während der Eingabe der Nachkommastellen der ersten Zahl. Wurde eine erste Zahl und eine (zweiwertige) Operation eingegeben, ist das Rechenwerk im Zustand q3. Der komplexe Zustand secondNumber ist hier vergröbert dargestellt und enthält die elementaren Zustände q4, q5 und q6). Er dient der Eingaben der zweiten Zahl und ist analog zum Zustand firstNumber aufgebaut. Im Zustand q7 ist das Rechenwerk, nachdem die Durchführung einer Berechnung abgeschlossen wurde.

die Beschreibung der Zustandsübergänge eher teil- oder informal. Abb. 19 auf Seite 39 zeigt die Verhaltensbeschreibung, nachdem die Zustandsübergänge ergänzt wurden.

Nachdem die Beschreibung des zustandsabhängigen Verhaltens des Rechenwerks nun soweit vollständig ist, wird beispielsweise deutlich, dass die Eingabe einer Ziffer je nach Zustand eine unterschiedliche Bedeutung hat: Je nachdem in welchem Zustand sich der Rechner befindet, ist

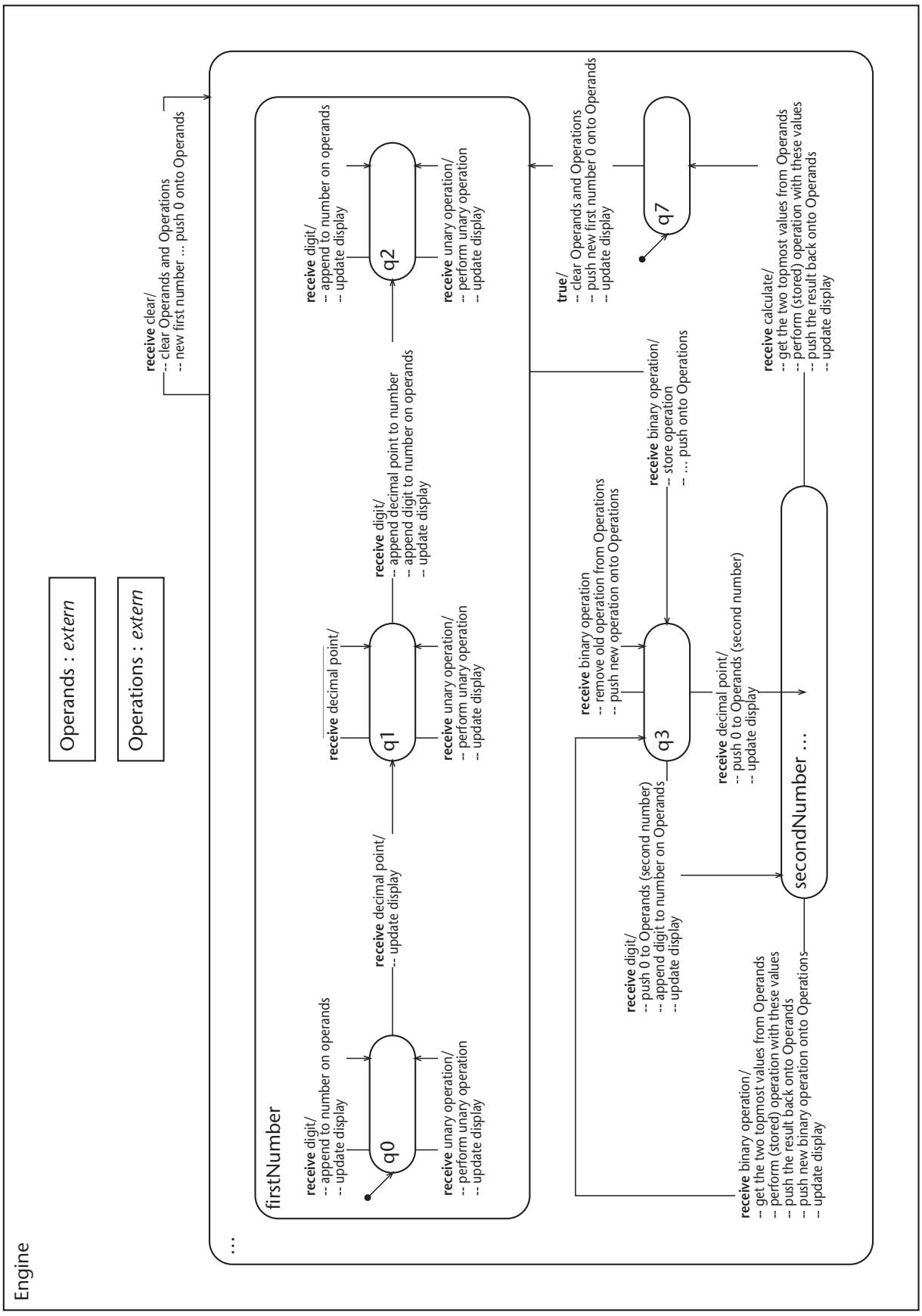


Abbildung 19: *ADORA-L-Modell des Rechenwerks. Verhaltensorientierte Einblendung nun mit (informal beschriebenen) Zustandsübergängen.*

eine eingegebene Ziffer entweder eine Vor- oder Nachkommastelle entweder der ersten oder der zweiten Zahl. Analog zur Tastatur kann – dort wo gewünscht – die hier noch teil- oder informale Beschreibung der Zustandsübergänge schrittweise formalisiert werden, um zu einer formale(re)n Beschreibung zu gelangen. Dies wird hier im Detail nicht weiter ausgeführt, sondern nur noch angedeutet (siehe Abb. 20 auf Seite 41).

Anzumerken ist noch, dass bei einem Modell mit diesem Formalitätsgrad (siehe Abb. 20 auf Seite 41, teilformal … Kommunikation wurde formalisiert, der 'Rest' ist aber noch informal) erste Simulationen oder Animationen möglich werden. Das Modell besitzt in wichtigen Teilbereichen, wie beispielsweise der 'Interobjektkommunikation', bereits eine präzise Ausführungssemantik. Es kann nun die Kommunikation und Kollaboration von Objekten nachvollzogen und ggf. in einem geeigneten Simulator gezeigt werden, um so eine frühe Validierung des geplanten Systems zu ermöglichen.

Zum Abschluss der verhaltensorientierten Einblendung des Taschenrechner-Beispiels soll – der Vollständigkeit wegen – noch die Beschreibung des Anzeigefelds gezeigt werden (siehe Abb. 21 auf Seite 40). Das Verhalten der Anzeigt ist, genau wie das der Tastatur auch, nicht zustandsabhängig. Die Anzeige (Display) wird ggf. vom Rechenwerk benachrichtigt, wenn eine Änderung erfolgt ist und die Anzeige aktualisiert werden müsste, wobei das Rechenwerk selbst entscheidet, wann oder besser aufgrund welcher Änderung des eigenen Zustandes die Anzeige entsprechend zu benachrichtigen ist. Die Anzeige erfragt auf eine solche Nachricht hin beim Rechenwerk die darzustellenden Informationen bzw. den darzustellenden Wert und stellt diesen anschliessend dar.

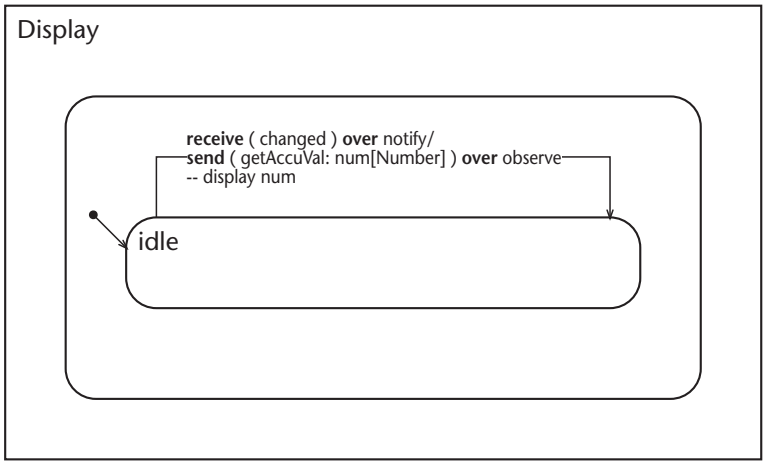


Abbildung 21: *Verhaltensbeschreibung der Anzeige des Taschenrechners. Wird die Anzeige (Display) über den Kanal notify benachrichtigt, dass sich etwas (im Rechenwerk) geändert hat, so erfragt sie (im Manipulationsteil) mittels einer synchronen Nachricht (send (getAccuVal: …) over observe) den aktuellen Wert des Akkus im Rechenwerk und stellt diesen Wert (num) dar.*

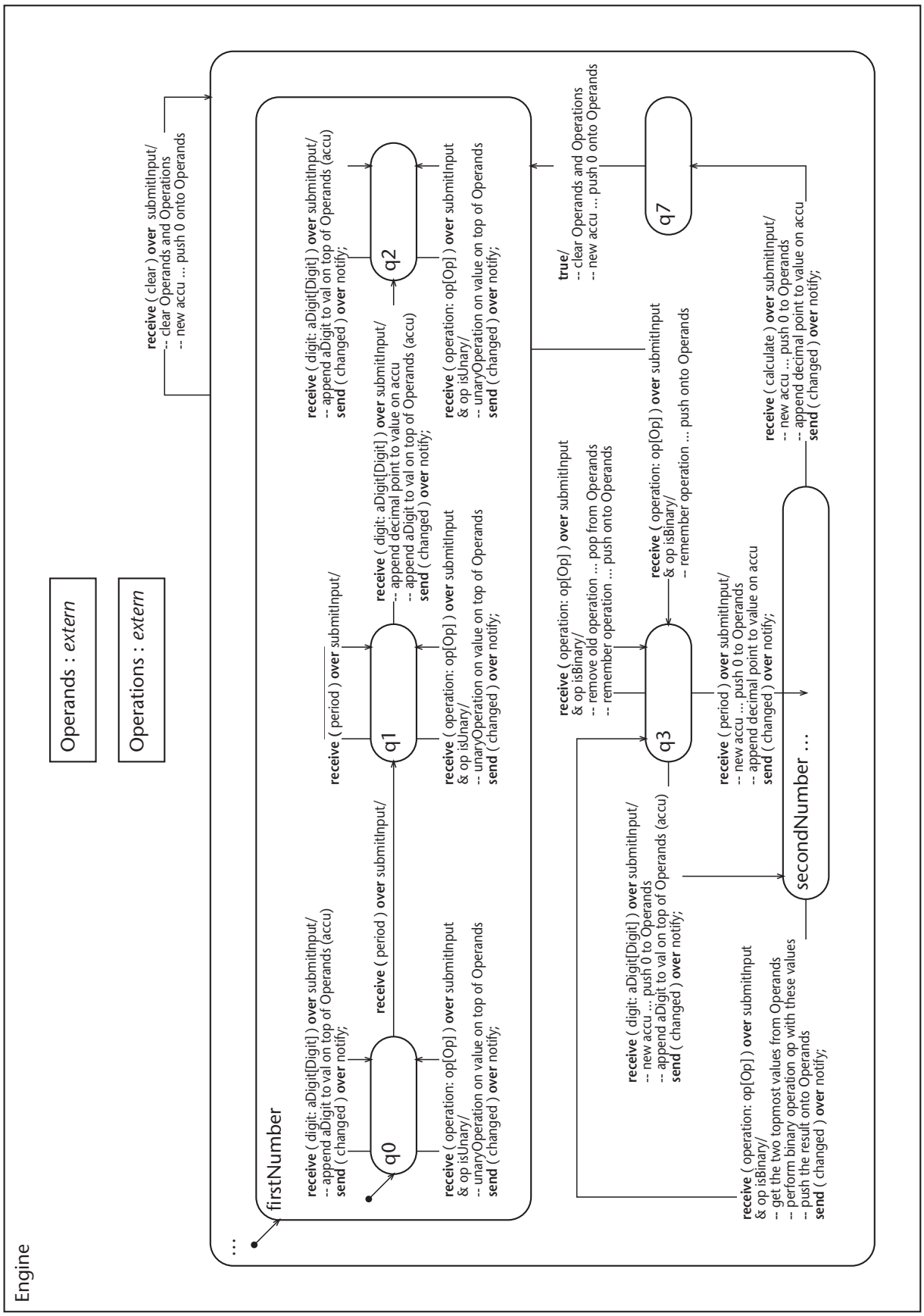


Abbildung 20: *Teilweise Formalisierung des Modells aus Abb. 19. Formal beschrieben sind hier die Nachrichtenteile sowie die Versendungsteile. Der Attributteil sowie der Manipulationsteil wurde zwar präzisiert, ist aber immer noch informal spezifiziert.*

5.7 Funktionale Einblendung – Objektoperationen

Die funktionale Einblendung dient der Beschreibung von zustandsunabhängigem Verhalten, eben ausgehend davon, dass ein Objekt selbst aktiv ist und je nach Zustand auf bestimmte Ereignisse hin (Eintreffen einer Nachricht) mit bestimmten Aktionen (Ausführung entsprechender Objektoperationen/Methoden) reagiert. Die eigentliche Objektoperation, also der zustandsunabhängige Teil der Verhaltensbeschreibung eines Objekts (hier auch Funktionalität genannt), wird in der verhaltensorientierten Einblendung nicht näher spezifiziert. Dies geschieht in ADORA-L in der sogenannten *funktionalen Einblendung*.

In der funktionalen Einblendung werden u.a. die Objektoperationen näher beschrieben, welche in der zustandsbasierten Verhaltensbeschreibung – also in der verhaltensorientierten Einblendung – verwendet werden können. Neben der Beschreibung der Objektoperationen werden in der funktionalen Einblendung auch noch weitere zustandsunabhängige Eigenschaften eines Objekts beschrieben, z.B. welcher Klasse ein Objekt zugeordnet ist oder welche Stereotypen Anwendung finden.

5.7.1 Sprachkonstrukte

Anmerkung: Die Elemente der funktionale Einblendung werden ausschliesslich textuell (und nicht grafisch) notiert.

Sprachkonstrukt: Elementarbeschreibung **(siehe Abb. 22)**

Zustandsunabhängige Informationen über ein Objekt oder eine Objektmenge werden in ADORA-L in der sogenannten *Elementarbeschreibung* zusammengefasst. Im einzelnen enthält diese Beschreibung Aussagen über:

- Die Klasse, welcher das Objekt zugeordnet ist.
- Die Stereotypen, die dem Objekt zugewiesen sind.
- Die Attribute des Objekts sowie deren Typen und Initialisierungswerte.
- Die Dienste, die das Objekt anbietet, also die zeitbehafteten (asynchronen) und zeitlosen (synchronen) Objektoperationen.

Für Objektmengen gibt es ebenfalls eine Elementarbeschreibung. Neben den o.g. Informationen enthält diese Beschreibung ausserdem noch:

- Eine Spezifikation der Metaoperationen der Objektmenge, d.h. der Operationen, zum Einfügen und Löschen von Instanzen in einer Objektmenge.

Da die Kenntnis sämtlicher Details der Elementarbeschreibung (ausführliche Syntax zur Annotation von Stereotypen, synchronen/asynchronen Objektoperationen, etc.) für ein grundlegendes Verständnis von ADORA-L nicht unbedingt notwendig ist (und ausserdem einiges dazu beitragen würde, diese bereits recht längliche Einführung noch länger zu gestalten), wird hier die Elementarbeschreibung nicht vollständig, sondern nur auszugsweise vorgestellt (ggf.

sei auf [Joos99] verwiesen). Abb. 22 zeigt den grundlegenden Aufbau der Elementarbeschreibung in ADORA-L.

```
object specification anObjectSet is
    class aClass
    stereotypes
        aStereotype( aSTVariable := anExpression, … );
        anotherStereotype( … );
    attributes
        aType anAttribute;
        aType anAttributeWithInitialization = anExpression;
    asynchronous operations
        anAsyncOperation: aParameter [aType] …
        …
    synchronous operations
        [aType] aSyncOperation: aParameter …
        …
    meta operations
        …
```

Abbildung 22: *Elementarbeschreibung in ADORA-L. Die (detaillierten) Beschreibungen der Stereotypen, Attribute und Operationen sind angedeutet.*

5.7.2 Beispiel Taschenrechner

Für das Rechenwerk des Taschenrechners wurde eine Elementarbeschreibung erstellt. Neben der Klasse des Rechenwerks und den zugewiesenen Stereotypen wird hier ersichtlich, welche Dienste bzw. Operationen der Taschenrechner bereitstellt und ob diese synchron oder asynchron sind. Wenn insbesondere die Objektoperationen ausreichend formal beschrieben wären – was hier nicht der Fall ist – könnte u.a. auf Basis dieser Beschreibung die verhaltensorientierte Einblendung weiter präzisiert werden.

```
object specification Engine is
    class CalculatorEngine
    stereotypes
        Observer( role=Observable, … )
    asynchronous operations
        operation digit: aDigit[Digit]
            entry -- Operandenstack nicht leer
            exit  …
        …
    synchronous operations
        getAccuVal num[Number]
            entry
            exit  num := …
        …
```

Abbildung 23: *Auszug aus der Elementarbeschreibung für das Rechenwerk (Engine) des Taschenrechners. Der Taschenrechner (Calculator) ist Objekt der Klasse CalculatorEngine. Er ist als Observable Teil eines Observer-Patterns. Die Operation digit ist eine asynchrone Operation, während die Operation getAccuVal eine synchrone Operaton ist (vgl. Abb. 21, verhaltensorientierte Einblendung der Anzeige (Display)).*

5.8 Typverzeichnis – Klassen & Stereotypen

Die letzte der aspektorientierten Einblendungen in ADORA-L ist das sogenannte *Typverzeichnis*. Während das Objektmodell die extensionalen Aspekte eines Systems auf der Basis abstrakter Objekte beschreibt, erlaubt es das Typverzeichnis intensionale Aussagen zu machen, d.h. es werden Aussagen über (die Art der) Ähnlichkeiten oder Gemeinsamkeiten zwischen abstrakten Objekten gemacht.

Das Typverzeichnis ist eine orthogonale Erweiterung des Objektmodells, es ist jedoch nicht eigenständig, wie bespielsweise ein 'konventionelles' Klassenmodell, sondern vielmehr eine Art Strukturierungsmittel, um im Objektmodell bereits vorhandene Informationen entsprechend bereinigt zusammenzufassen und so intensionale Aussagen zu treffen. Eigenschaften, welche bereits im Objektmodell beschrieben wurden, werden im Typverzeichnis nicht nochmals beschrieben.

Im ADORA-L-Typverzeichnis werden folgende Sachverhalte modelliert:

- Die Objektschablonen und Klassen – Hier werden die typspezifischen sowie intensionalen Eigenschaften der im Objektmodell vorkommenden Objekte und Objektmengen zusammengefasst.
- Die Klassenhierarche – Eine nach Generalisierungs/Spezialisierungs-Aspekten angeordnete Hierarchie der Klassen der im Objektmodell vorkommenden Objekte.
- Die Stereotypen – Die im Objektmodell zur Verfügung stehenden Stereotypen.
- Die Datentypen – Beschreibung des Formats der im Objektmodell vorkommenden, identitätslosen Daten, wie z.B. Attribute oder Nachrichtenparameter.

5.8.1 Sprachkonstrukte

Sprachkonstrukt: Klasse

Eine Klasse (synonym: Objekttyp) ist ein Typ, der die gemeinsamen, typspezifischen Eigenschaften einer Menge von gleichartigen Objekten intensional definiert. Der Klassenbegriff in ADORA-L ist rein intensional. Zur Beschreibung extensionaler Aspekte stehen abstrakte Objekte und Objektmengen zur Verfügung.

Die typspezifischen Eigenschaften einer Klasse werden durch die (Referenzierung der) entsprechenden Objektschablone (grafisch und textuell) beschrieben. Weiterhin werden sie im Typverzeichnis in Form einer Klassenhierarchie (grafisch) annotiert.

Sprachkonstrukt: Objektschablone

Die *Objektschablone* beschreibt die typspezifischen Eigenschaften der Objekte oder Objektmengen die einer Klasse angehören. Alle Objekte einer Klasse müssen die in der Objektschablone definierten Eigenschaften aufweisen. Eine Objektschablone besteht aus der:

- Basisstruktur – für alle Objekte die Komponenten des Objekts sind, welches durch die Objektschablone beschriebenen ist. Die Namen der Komponenten sind hierbei für alle Objekte einer Klasse identisch.
- strukturellen Einblendung – die dort vorhandenen typspezifischen Beziehungen. Dies sind ausschliesslich diejenigen Beziehungen, welche zwischen den Objektkomponenten der entsprechenden Komposition vorhanden sind. Nicht typspezifisch und damit nicht enthalten sind strukturelle Beziehungen zu lediglich im Kontext stehenden Objekten (also zu Objekten, die keine Komponenten sind).
- verhaltensorientierten Einblendung – sämtliche Zustände, d.h. elementare Zustände, komplexe Zustände sowie Komponentenzustände. Die aufgeführten Übergangsbedingungen und -aktionen werden in reduzierter Form (siehe Abb. 24) übernommen.
- funktionalen Einblendung – alle Objektattribute sowie in reduzierter Form (Nachrichtenversand, Initialisierungen) alle Objektoperationen bzw. Metaoperationen.

Die durchgeführte(n) bzw. durchzuführende(n) Reduktion(en) werden hier nicht im Detail behandelt, sondern nur exemplarisch (für Details sei wie üblich auf [Joos99] verwiesen), um das Prinzip verständlich zu machen. Hierfür soll der in Abb. 16 gezeigte Zustandsübergang entsprechend reduziert werden: aus der Übergangsbedingung sowie der Übergangsaktion werden alle Bezüge zu ‘Nicht-Objektkomponenten’ entfernt.

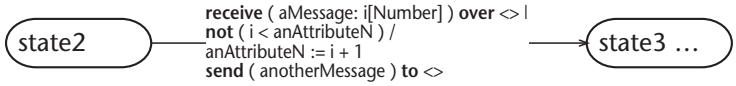


Abbildung 24: *Für die Objektschablone reduzierter Zustandsübergang (vgl. Abb. 16). Die Nachricht aMessage und anotherMessage kommen von oder gehen an ObjectD (siehe Abb. 9 auf Seite 29). ObjectD ist nicht Komponente von ObjectA, also wird entsprechend reduziert; Sender und Empfänger werden durch einen Platzhalter ersetzt (<>).*

Sprachkonstrukt: Generalisierung/Spezialisierungs-Beziehung

Eine Generalisierungs/Spezialisierungs-Beziehung drückt aus, dass eine Klasse (die sogenannte *Unterklasse*) ein Spezialfall einer anderen Klasse (der sogenannten *Oberklasse*) ist, oder andersherum die Oberklasse ist eine Verallgemeinerung der Unterklasse. Eine Klasse kann dann Spezialfall also Unterklasse einer bestimmten Klasse sein, wenn alle Eigenschaften dieser Klasse auch für die Unterklasse gelten.

Mittels der Generalisierungs/Spezialisierungs-Beziehung werden Klassen in einer sogenannten *Klassenhierarchie* (siehe Abb. 25) angeordet.

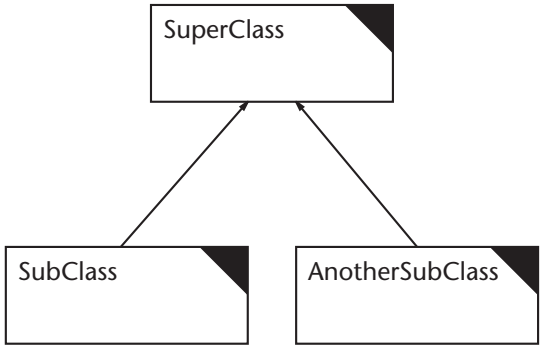


Abbildung 25: *Klassenhierarchie. SuperClass, SubClass, AnotherSubClass sind Klassen. Die Klasse SuperClass ist Oberklasse der Klassen SubClass und AnotherSubClass. Die Klassen SubClass und AnotherSubClass sind Unterklassen der Klasse SuperClass.*

Sprachkonstrukt: Stereotyp (siehe Abb. 26)

Stereotypen erlauben es, die Elemente eines Modells, zusätzlich zu den bereits vorhandenen Klassifikationsarten zu klassifizieren. ADORA-L unterstützt die Beschreibung und Verwendung von deskriptiven und restriktiven Stereotypen [Be+99b]. Die Beschreibung eines Stereotyps in ADORA-L geschieht nach dem in Abb. 26 gezeigten Schema.

stereotype *Name des Stereotyps*
  for
    Angabe der Sprachelemente, für welche der Stereotyp definiert ist
    Sprachelement,
    ein anderes Sprachelement,
    …
  description
    kurze Beschreibung des Stereotyps
    …
  variables
    optionale Deklaration der Variablen des Stereotyps
    aStereotypeVariable **is** …
    anotherOne **is** …
    …
  conditions
    wenn es sich um einen restriktiven Stereotyp handelt,
    dann folgt hier die Angabe entsprechender Bedingungen
    (entweder formal, teil-formal oder informal)
    …
end stereotype

Abbildung 26: *Schema für die Definition eines Stereotyps in ADORA-L.*

Sprachkonstrukt: Datentyp (siehe Abb. 27)

Ein Datentyp definiert die Bedeutung, die Struktur und die Wertebereiche von Daten ohne eigenständige Identität. Datentypen werden in ADORA-L verwendet, um lokale Variablen in Operationen, Objektattribute und Nachrichtenparameter zu definieren. Sie finden demnach in der verhaltensorientierten wie auch der funktionalen Einblendung Verwendung.

ADORA-L kennt elementare (Aufzählungstypen, Boolean, Integer, Real, String, Time, etc.) und zusammengesetzte Datentypen (Listenstrukturen, Records, etc.). Um Datentypen ggf. formal zu beschreiben, wird die Teilsprache ADORA-FSL verwendet. Abb. 27 gibt einige Beispiele für durch ADORA-FSL definierte Datentypen.

```
Boolean
Integer
typedef Cardinal is Integer where ( Integer >= 0 )
typedef Digit is Cardinal where ( (0 <= Cardinal) and (Cardinal <= 9) )
typedef RedLightColors is ( red, yellow, green )
typedef Point is structure of ( x : Real; y : Real )
typedef Points is list of Point
```

Abbildung 27: *Beispiel für die Deklaration von Datentypen in ADORA-L.*

5.8.2 Beispiel Taschenrechner

Die Klassenmodellierung im Taschenrechnerbeispiel ist nicht allzu ergiebig, da mit Ausnahme der Tasten der Tastatur (und diese sind externe Objekte) von jeder Klasse nur ein Objekt existiert. Insofern wäre es wenig lehrreich, die entsprechenden Klassendefinitionen oder Objektschablonen zu zeigen. Auch existieren keine Generalisierungs/Spezialisierungs-Beziehungen, und somit erübrigt sich die Anführung der entsprechenden Klassenhierarchie.

Die Anzeige und das Rechenwerk des Taschenrechners realisieren ein Observer-Pattern [Gam+95]. Abb. 28 zeigt schematisch die Definition des hier verwendeten Stereotyps Observer (vgl. auch Abb. 23 auf Seite 43, Elementarbeschreibung des Rechenwerks).

```
stereotype Observer
    for Object, ObjectSet
    description
        Observer-Pattern nach [Gam+95]
    variables
        role is (observable, observer);
        …
    conditions
        ( role = 'Observable' ) implies
            …
        and
        ( role = Observer ) implies
            …
end stereotype
```

Abbildung 28: *Ausschnitt aus der Definition des restriktiven Stereotyps Observer. Je nachdem welche Rolle (Observable oder Observer) das 'gestereotypte' Objekt einnimmt, wird die Stereotyp-Variable/der Tag role entsprechend belegt. In Abhängigkeit von dessen Belegung müssen dann bestimmte Restriktionen erfüllt sein (hier nicht weiter ausgeführt).*

Vorgestellt werden soll hier der Datentyp Op. Dieser wird in der Tastatur und im Rechenwerk verwendet. Wenn eine Operationstaste gedrückt wurde, wird das Rechenwerk darüber informiert, dass eine Operationstaste gedrückt wurde (-> operation: op[Op]). Am Parameter dieser Nachricht – ein Wert vom Typ Op – kann/könnte das Rechenwerk ggf. feststellen, welche

Operationstaste gedrückt wurde und die entsprechende Operation durchführen (lassen). Abb. 29 zeigt die Definition des Datentyps Op.

```
typedef Op is ( ADD, SUB, MUL, DIV, NEG )
```

Abbildung 29: *Der Datentyp OP ist ein Aufzählungstyp*

Grundlagen Visualisierung

Ein Grossteil dieser Arbeit beschäftigt sich mit der adäquaten Visualisierung von ADORA-Modellen. Daher werden in Kapitel 6 die Visualisierung betreffende Grundlagen und Begriffe eingeführt. Auf die Visualisierung hierarchischer Strukturen wird speziell gegen Ende des Kapitels in einem eigenen Unterkapitel eingegangen.

In Kapitel 7 wird auf Visualisierungskonzepte an sich eingegegangen. Hierfür werden die Merkmale angesprochen, die zusammengenommen ein Visualisierungskonzept ausmachen. Hier wird – vorbereitend auf nachfolgende Kapitel – eine Struktur für die Beschreibung verschiedener Visualisierungskonzepte eingeführt. Nachfolgend werden typische Vertreter bestehender Visualisierungskonzepte vorgestellt.

Eine Zusammenfassung findet sich in Kapitel 8. Insbesondere werden hier existierende Fischaugenkonzepte im Hinblick auf ihre Eignung zur Visualisierung von ADORA-Modellen diskutiert.

6 MODELL-VISUALISIERUNG

Nachdem im vorangegangen Kapitel 5 der Gegenstand der Visualisierung – nämlich mittels der Sprache ADORA-L erstellte Modelle – eingeführt wurde, erfolgt nun ein Themenwechsel.

Dieses Kapitel kann im Wesentlichen ohne ADORA-L Kenntnisse gelesen werden. Es geht nicht darum, in die Sprache ADORA-L einzuführen, sondern darum, sprachunabhängige Grundlagen vorzustellen, die für das Verständnis des nachfolgend vorgestellten Ansatzes zur Visualisierung von ADORA-Modellen wichtig sind.

Begriffe wie Visualisierung, Sicht, Fokus oder Navigation sind oft mit mehr oder weniger unterschiedlichen Bedeutungen belegt. Hier ist es notwendig, eine einheitliche Begrifflichkeit zu schaffen und im Zusammenhang mit der Visualisierung hierarchischer Modelle Definitionen ggf. zu ergänzen oder zu präzisieren. Zu diesem Zweck werden in Kapitel 6.1 zuerst einige zentrale Begriffe eingeführt und definiert. Teils sind dies vollständig eigene Definitionen und teils wird – unter Angabe der Referenz – auf vorhandenen Definitionen aufgebaut. Anschliessend werden in Kapitel 6.2 Projektionstechniken angeschnitten. Projektionstechniken werden hier anhand ihres Betrachtungsparadigmas unterschieden. Diese Klassifikation ist neu und zielt darauf ab, die Thematik ‘Kontexterhalt’ hervorzuheben. Zum Abschluss wird in Kapitel 6.3 auf einige konzeptionelle Besonderheiten bei der Visualisierung hierarchischer Modelle eingegangen. Dieses Kapitel stützt sich auf Ergebnisse ab, die im Projekt im Rahmen der Diplomarbeit von Känel [Kaen96] gewonnen wurden.

6.1 Grundbegriffe

Visualisierung

Unter *Modell-Visualisierung* wird verstanden, wie – mittels Benutzung eines Rechners und unter Verwendung von Software-Werkzeugen – ein Modell auf einem Anzeigegerät, unter Anwendung einer grafischen Notation, dargestellt werden kann. Im Zusammenhang mit Visualisierung tauchen oft Begriffe wie Sicht, Fokus, etc. auf. Diese Begriffe sollen nun erläutert werden.

Sicht

Visualisierung geschieht auf der Basis von Sichten. Eine *Sicht* ist ein bestimmter, aktuell dargestellter Ausschnitt eines Modells. Konzeptionell kann eine Sicht somit als Modell des Originalmodells, also als Teilmodell betrachtet werden. Dieses Teilmodell wird i.d.R. mittels einer bestimmten *Projektionstechnik* (siehe Kapitel 6.2) automatisch generiert und weist – wie jedes Modell – eine bestimmte Verkürzung (siehe Kapitel 4.3) auf.

Die Verkürzung des Originals geschieht typischerweise auf zwei Arten:

- *ausprägungsbezogen* oder *schemabezogen*.

Wird ausprägungsbezogen verkürzt, so werden entweder bestimmte Modell-Elemente auf gleicher oder tieferen Abstraktionsebenen weggelassen. Das Weglassen von Modell-Elementen auf gleicher Abstraktionsebene wird auch als sogenannte horizontale oder partitionierende Verkürzung bezeichnet. Das Weglassen von bestimmten Detailinformationen auf tiefer liegenden Abstraktionsebenen nennt man vertikale oder abstrahierende Verkürzung.

Wenn schemabezogen verkürzt wird, so werden *alle* Modellelemente weggelassen, welche Ausprägung eines bestimmten Sprachkonstrukts sind; beispielsweise alle Attribute oder alle Beziehungen bzw. Beziehungstypen. Dies würde einer vertikalen oder horizontalen Verkürzung im Schema bzw. Metamodell entsprechen.

Horizontale Verkürzungen führen zu Fragmenten bzw. Partitionen des Originalmodells. Sie beschreiben nicht mehr das gesamte ursprüngliche System, erhalten aber den Detaillierungsgrad und bilden eine offene Abbildung des Originalmodells, da nicht mehr das gesamte Modell gezeigt wird. Vertikale Verkürzungen führen zu Abstraktionen des Originalmodells. Hier wird zwar generell das gesamte System abgebildet, jedoch in mehr oder weniger vergröberter Form. Die Abbildung ist geschlossen, da immer noch das gesamte Modell gezeigt wird.

Welche Elemente in einer Sicht dargestellt werden und wie der Detaillierungsgrad dieser Elemente ist, wird durch die Intention des Modellierers oder des Modellnutzers determiniert (vgl. Kapitel 4.4, Pragmatisches Merkmal). Der Modellierer oder der Modellnutzer bestimmt – explizit oder implizit – die jeweilige Verkürzung des Originalmodells, um eine entsprechende Sicht zu erhalten und sich so die Informationen zu beschaffen, die er momentan benötigt.

Abhängig von der Flexibilität der Sprache oder des Werkzeugs, kann der Modellnutzer eigene Sichten generieren oder er bleibt – wie beispielsweise bei UML-Modellen – auf die durch den Modellierer vorgegebenen Sichten beschränkt. Ist letzteres der Fall, so muss sich der Modellnutzer die gewünschten Informationen durch abwechselndes Betrachten vorgegebener Sichten beschaffen.

Die Bildung horizontaler Verkürzungen ist weitgehend sprachunabhängig, so dass i.d.R. entsprechende Sichten nicht durch den Modellierer vorgegeben sein müssen. Problematisch wird es meist erst dann, wenn abstrahierende Sichten durch den Modellnutzer gebildet werden sollen. Wie alle Sichten, die auf vertikalen Verkürzungen basieren, können abstrahierende Sichten überhaupt nur dann von einem Werkzeug automatisch generiert werden, wenn dies im Schema vorgesehen ist, also durch die Sprachdefinition unterstützt wird. Das Vorhandensein entsprechender Sprachkonstrukte ist zwar notwendig, jedoch nicht hinreichend. Zusätzlich müssen diese Sprachkonstrukte im Modell sinnvoll verwendet werden, d.h. der Modellierer reichert bei der Modellierung das Modell bereits mit derartigen Informationen an.

Werden die entsprechenden Sprachkonstrukte vom Modellierer nicht oder nicht sinngemäss verwendet oder sind überhaupt keine sprachlichen Mittel hierfür vorhanden, so müssen derartige Sichten manuell vom Modellierer vorgegeben werden und sind für den Modellnutzer nur dann wirklich von Nutzen, wenn der Modellierer weiss oder vorausahnt, auf welcher Abstraktionsebene der Modellnutzer arbeiten möchte.

Fokus

Gibt es in einem Modell einen bestimmten Bereich, Punkt oder Ort, welcher zeitweise für den Modellnutzer von speziellem Interesse ist, so bezeichnet man diesen als *Fokus* [Sar+92].

Werden Sichten durch ein Werkzeug automatisch generiert, so kann der Fokus als Grundlage für die Bestimmung der in einer Sicht darzustellenden Elemente dienen; d.h. macht der Benutzer den Fokus – mittels eines Eingabegeräts wie beispiesweise einer Maus – explizit, so kann diese Information verwendet werden, um automatisch unter Anwendung einer bestimmten Projektionstechnik die gewünschte Sicht zu generieren. Die in dieser Sicht darzustellenden Modellelemente sowie deren Detaillierungsgrad sind in diesem Fall implizit über ihren Bezug zum Fokus bestimmt.

Navigation und kognitiver Ballast

> *… one of the common problems associated with large computer-based information systems is the relatively small window through which an information space can be viewed [Leu+94]*

Bedingt durch die Limitationen von Ausgabe- bzw. Anzeigegeräten oder die Begrenztheit des menschlichen Auffassungsvermögens stellt eine Sicht i.d.R. nur einen Teil des Originalmodells dar. Soll ein anderer Teil des Modells dargestellt werden, so ist es erforderlich zu navigieren. *Navigation* ist die Bestimmung der in einer Sicht anzuzeigenden Elemente sowie die Art und Weise, wie diese Elemente darzustellen sind. Hierbei können die in einer Sicht anzuzeigenden Elemente entweder direkt oder indirekt (siehe oben, ‘Fokus’) bestimmt werden.

Navigation ist ein nicht-essentieller Teil [Broo87] einer Benutzeraufgabe. Ein Benutzer navigiert, um das System dazu zu veranlassen, die Sicht bereitzustellen, die er gerade benötigt. Das ist *nicht* die Aufgabe, die der Benutzer eigentlich ausführen will, sondern vielmehr die Vorbereitung, die für die Ausführung dieser Aufgabe notwendig ist.

Fragt man sich, welche grundlegenden Arten von Tätigkeiten benutzerseitig notwendig sind, um (auf einem Rechner unter Benutzung von Software-Werkzeugen) die in einer Sicht anzuzeigenden Elemente zu bestimmen, zerfällt Navigation in zwei Aspekte:

- einen *kognitiven* und einen *mechanischen* Aspekt.

Der kognitive Aspekt umfasst den mentalen Aufwand, der benutzerseitig notwendig ist, um den aktuellen Fokus zu bestimmen und diesen anschliessend zu verändern, also den Prozess der Wahrnehmung und Verarbeitung von Information bis zur Umsetzung in Handlungen (vgl. [Char94]). Der mechanische Aspekt bezeichnet den rein motorischen Aufwand – wie beispielsweise das Bewegen einer Maus – der aufgebracht werden muss, um ein kognitives Navigationsziel umzusetzen.

Der kognitive, nicht-mechanische Aufwand für Navigationsaktivitäten wird als *kognitiver Ballast (cognitive overhead)* [Bea+90] bezeichnet. Kognitiver Ballast sind all diejenigen Überlegungen und mentalen Aufwendungen, die ausschliesslich dazu dienen, die zum Erreichen

eines Navigationsziels notwendigen Bedienungsschritte zu determinieren[1]. Kognitiver Ballast ist *kein* Mass, welches (derzeit) konkret quantifiziert werden kann, sondern eine Art Denkmodell, um zu beurteilen, in welchem Umfang ein Benutzer mit im Brooks'schen Sinne nicht essentiellen Tätigkeiten belastet wird. Im Idealfall – leider eher unrealisierbar – ist die Bestimmung und Veränderung des Fokusses frei von kognitivem Ballast. Die Maschine wäre dann grundsätzlich in der Lage, den neuen Fokus korrekt vorherzusagen bzw. zu raten.

6.2 Projektionstechniken

… oder wie schaffe ich den Platz für die zusätzlichen Details

Eine Projektionstechnik gibt an, auf welche Art und Weise die Elemente eines Modells in einer Sicht dargestellt werden. Eine Projektionstechnik ist eine Interpretations- oder Abbildungsvorschrift, mit dem Ziel, bestimmte Sichten des Modells zu erzeugen.

Da der Gegenstand dieser Arbeit die Visualisierung von ADORA-Modellen ist, werden Projektionstechniken speziell dann interessant, wenn verfeinert oder vergröbert dargestellt werden soll. Unter der Voraussetzung, dass für den Betrachter in der neuen Sicht ein gewisser Bezug zur vorangegangenen Sicht erhalten bleibt, müssen im Fall der Verfeinerung irgendwann bestimmte Ausschnitte der aktuellen Sicht vergrössert dargestellt und ggf. Modell-Elemente repositioniert werden. Mit anderen Worten: dient die alte Sicht als Basis für die Erzeugung der neuen Sicht, muss Platz für zusätzliche Details geschaffen werden, welche für eine verfeinerte Modellsicht notwendig sind. Für die Vergröberung gilt Entsprechendes.

Wie eine Projektionstechnik zusätzlichen Platz für eine verfeinerte Sicht beschafft oder überflüssigen Platz für eine vergröberte Sicht wieder los wird, ist abhängig vom Betrachtungsparadigma. Grundsätzlich können hier zwei Paradigmen unterschieden werden:

- *perspektivlose* und *perspektivbehaftete*.

Nach diesen Betrachtungsparadigmen richtet sich auch die hier vorgenommene Einteilung von Projektionstechniken[2].

Für die perspektivlose Betrachtung kommen *lineare* Projektionstechniken zum tragen, während die perspektivbehaftete Betrachtung *nicht-lineare* Techniken benötigt.

---

[1] Analog zum kognitiven Ballast kann auch mechanischer Ballast definiert werden. Dieser ist bei der Arbeit mit Software-Werkzeugen vernachlässigbar, da der Weg von der Determination eines Bedienungsschrittes zu dessen Ausführung i.d.R. klein ist und beide unmittelbar korreliert sind. Auch in der entsprechenden Literatur [Bea+90] wird hierauf nicht weiter eingegangen.

[2] Projektionstechniken oder besser deren Arbeitsweise können unterschiedlich klassifiziert werden. Bestehende Klassifikationen wie sie bei [Leu+94] oder [Keah97] zu finden sind, klassifizieren Projektionstechniken anhand bestimmter algorithmischer Merkmale, wie z.B. anhand des Aufbaus der Transformationsfunktion oder ob sie zusammengesetzt oder atomar arbeiten, etc. Dies ist zweckmässig, wenn Interesse an den Interna bestimmter Ausprägungen einer bestimmten Projektionstechnik besteht. Wird so klassifiziert, kann es vorkommen, dass Projektionstechniken, die unterschiedlich klassifiziert sind, zur gleichen Sicht führen (was sinnvoll ist, insofern Interesse an der Arbeitsweise besteht). Genau dies ist hier aber eher hinderlich, da lediglich Interesse daran besteht, wie die resultierende Sicht aufgebaut ist.

6.2.1 Perspektivlose Strategien und lineare Projektionen

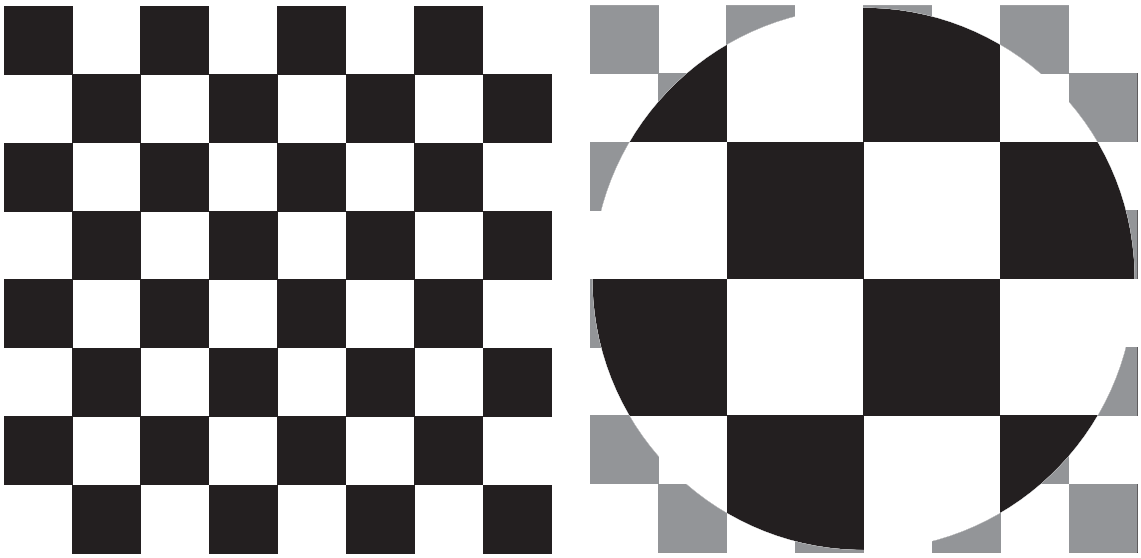


Abbildung 30: *Lineare Vergrösserung als Beispiel für eine lineare Projektion.*

Bei linearen Projektionen wird der gesamte zu verfeinernde Bereich gleichmässig vergrössert. Dies schafft den Platz, der notwendig ist, um die zusätzlichen Details darstellen zu können. Hierbei bleiben die Proportionen des Originals erhalten. Dies ist oft vorteilhaft, weil ein vorgegebenes Layout ohne grossen Aufwand erhalten werden kann und die neue Sicht ähnlich zur alten Sicht ist. Der Bezug zur alten Sicht ist offensichtlich(er) und somit findet sich der Betrachter schnell in der neuen Sicht zurecht. Reicht jedoch der zur Verfügung stehende Platz nicht aus – was recht schnell der Fall ist – so kann nur noch ein Teil des Originals gezeigt werden (siehe Abb. 30). Die Sicht stellt dann nur noch eine Partition des Originalmodells dar (vgl. Kapitel 6.1) und es müssen Möglichkeiten zur Navigation bestehen, wenn das Gesamtmodell betrachtet werden soll (vgl. Kapitel 6.3.2).

Soll beispielsweise das in Abb. 30 (auf der rechten Seite) dargestellte Schachbrettmuster linear vergrössert werden, so müssen Bereiche des Originals weggelassen werden. Selbst wenn auf dem Anzeigegerät verhältnismässig viel Platz zur Verfügung steht, ist dieser schnell erschöpft, da bei Projektionen in der Ebene der Platzbedarf im Quadrat mit dem Vergrösserungsfaktor steigt. Das kann soweit gehen, dass es besser ist, das Objekt ganz wegzulassen und nur noch die vergrösserte Projektion anzuzeigen (Explosive Zoom, vgl. S. 69)

Lineare Projektionen dienen der *perspektivlosen* Betrachtung. Sie sind inbesondere dann angebracht, wenn jeder Bereich oder besser jeder Punkt für den Betrachter gleich wichtig ist, da bedingt durch die Projektionstechnik alles gleichberechtigt – mit anderen Worten isometrisch – dargestellt wird. Ist der Betrachter eigentlich nur daran interessiert, einen vergleichsweise kleinen Bereich verfeinert darzustellen, so erzeugen sie zuviel unnötiges Detail.

6.2.2 Perspektivbehaftete Strategien und nicht-lineare Projektionen

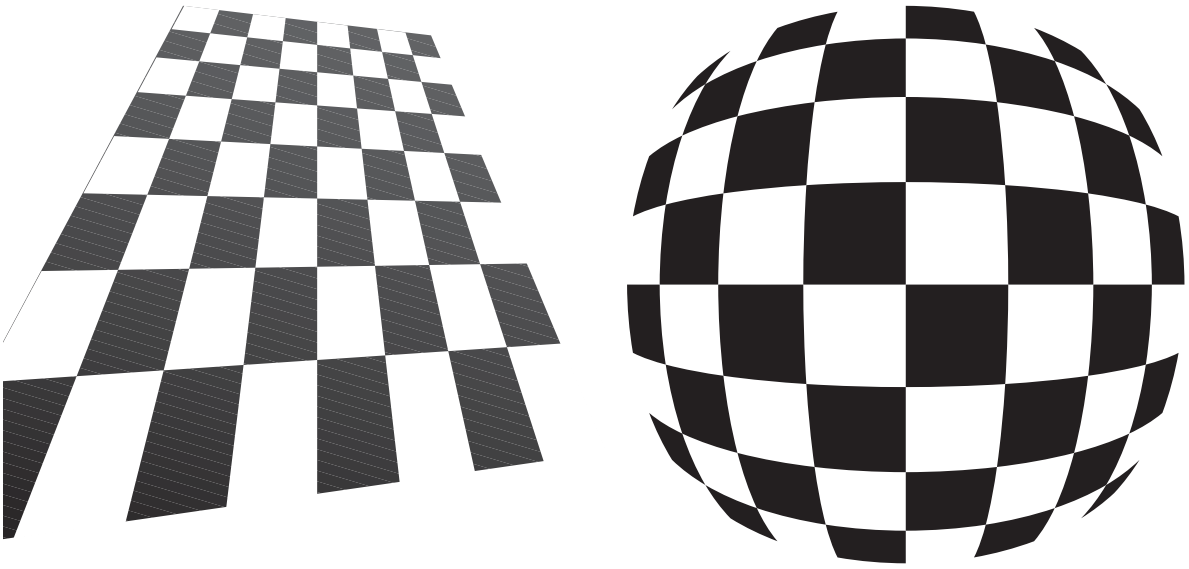


Abbildung 31: *Perspektivische Darstellung und Projektion auf eine Halbkugel als Beispiel für nicht-lineare Projektionen mit einem Fokus.*
Anmerkung: Da hier der Informationsgehalt bei normaler und verfeinerter Version gleich ist, wird durch die Vergrösserung keine zusätzlich Information sichtbar und alles was sich ändert, ist lediglich die Grösse (und Gestalt) der Quadrate.

Eines der einfachen und zugleich anschaulichen Beispiele, um nicht-lineare Projektionen und deren Eigenschaften beispielhaft zu demonstrieren, ist die Projektion eines Schachbrettmusters auf eine Kugel bzw. auf eine Halbkugel (siehe Abb. 31) bzw. dessen Betrachtung mit einer Weitwinkellinse, auch Fish-eye genannt. Der Bereich um den Fokus (hier ist der Fokus der Linse gemeint) wird vergrössert, während mit wachsendem Abstand von diesem zunehmend verkleinert wird.

Man kann sich die Anwendung einer solchen Technik so vorstellen, dass der Benutzer mittels eines Eingabegeräts – beispielsweise einer Maus – den Fokus angibt und dieser Bereich sinngemäss so dargestellt wird, wie in Abb. 31. Der Bereich um den Fokus wird vergrössert, während mit zunehmender Entfernung vom Fokus verkleinert wird.

Verglichen mit linearen Konzepten (siehe Kapitel 6.2.1), weisen klassische nicht-lineare Projektionen folgende Eigenschaften bzw. Vorteile auf (vgl. auch [Keah97]):

- *Detailbetonung*, d.h. verbesserte Detailsicht relevanter Bereiche
- *Kontexterhaltung*, d.h. der globale Kontext geht nicht verloren, sondern wird verkleinert oder vergröbert mitdargestellt.
- *Okklusionsfreiheit*, d.h. die Vergrösserung eines Bereichs blockiert bzw. verdeckt keine angrenzenden Bereiche.
- *In-Situ-Vergrösserung*, d.h. die detaillierte Sicht wird im Kontext gezeigt und nicht in einem separaten Fenster.

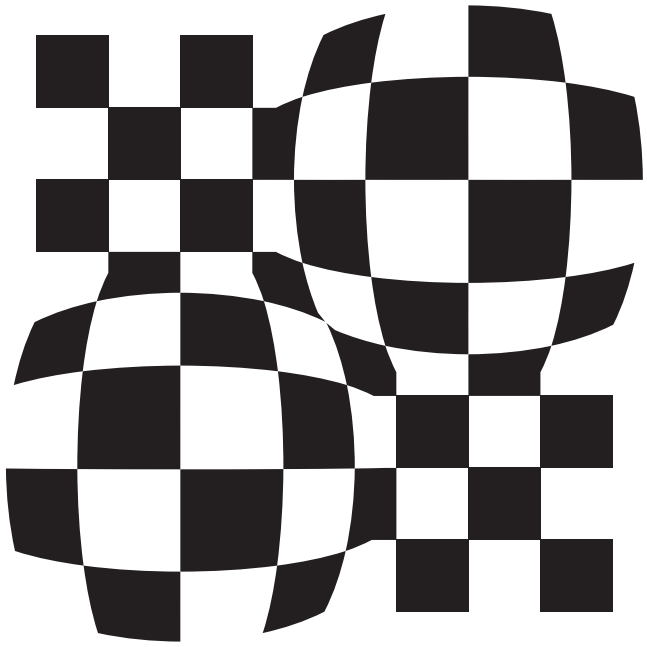


Abbildung 32: *Beispiel für eine nicht-lineare Projektion mit zwei Foki.*

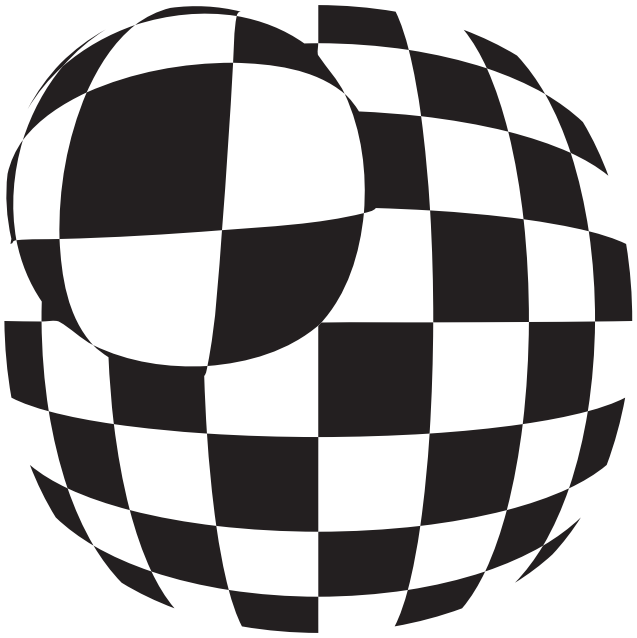
Traditionelle Konzepte (siehe Kapitel 6.2.1) – also Konzepte, die auf rein linearen Projektionen basieren – sind nicht in der Lage, alle o.g. Vorteile zu vereinen, da sich bei linearen Projektionstechniken bestimmte Eigenschaften gegenseitig ausschliessen, z.B. Detailbetonung bei gleichzeitiger Okklusionsfreiheit.

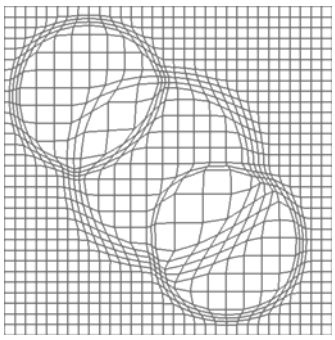
Nicht-lineare Projektionen dienen einer *perspektivbehafteten* Betrachtung und können vorteilhaft eingesetzt werden, wenn nicht jeder Bereich für den Betrachter gleich wichtig ist. Erlaubt es die Projektionstechnik nur einen einzigen Bereich vergrössert darzustellen, so spricht man von einem *Einzelfokus* (single focus).

Manchmal ist es für den Betrachter wünschenswert, nicht nur einen Bereich, sondern gleich mehrere Bereiche – jedoch niemals gleichmässig die gesamte Sicht, sonst wäre es eine lineare Projektionstechnik – detailliert darzustellen. Erlaubt es die Projektionstechnik, mehr als einen Bereich detailliert zu betrachten, während die restlichen Bereiche vergröbert bleiben, so spricht man von einem *Mehrfachfokus* (multiple foci, multi-focus [Leu+94]) (siehe Abb. 32).

Nicht-lineare Projektionen haben unter bestimmten Umständen gewisse ‘Unschönheiten’. Die Verwendung von Mehrfachfoki erlaubt es zwar, mehrere Bereiche detailliert darzustellen. Wenn jedoch ein bereits detailliert dargestellter Bereich weiter verfeinert werden soll – dieser Fall wird hier ‘Fokus auf Fokus’ genannt – treten bedingt durch Stellen mit Unstetigkeiten und/oder hoher Steigung unschöne Anomalien auf (siehe Abb. 33), die eine übermässig verzerrte Darstellung des Originals zur Folge haben [Leun89].

Die Anomalie kann zur Folge haben, dass ein Bereich, der eigentlich näher am Fokus liegt, entweder stark verzerrt oder wider Erwarten nicht vergrössert dargestellt wird, bzw. nicht vergrös-





Beispiel für Multi-Level Magnification [Keah98]

Abbildung 33: *Anomalie bei Fokus auf Fokus. Stellen mit Unstetigkeiten und/oder (zu) hoher Steigung führen dazu, dass bestimmte Bereiche, die näher am Fokus liegen als andere und somit vergrössert dargestellt werden müssten, entweder verzerrt oder verkleinert dargestellt werden.*

sert dargestellt werden kann (siehe Abb. 33). Die generierte Sicht ist im Sinne von ‘wenn etwas näher am Fokus liegt, wird es vergrössert dargestellt’ kontra-intuitiv und somit nicht immer einfach zu verstehen, da sie eben nicht mehr so beschaffen ist, wie es der Betrachter ‘normalerweise’ erwarten würde.

Insbesondere bei der Darstellung hierarchischer Strukturen (siehe auch Kapitel 6.3) ist die weitere Detaillierung eines bereits detailliert dargestellten Bereiches meist der Normalfall und nicht die Ausnahme. Mit wachsender Tiefe der Struktur wächst bei der Verwendung klassischer nicht-linearer Projektionstechniken (siehe Eigenschaften S. 56) und entsprechender Verfeinerung die Wahrscheinlichkeit, dass o.g. Anomalien auftreten und dadurch die Sicht zunehmend unverständlich wird.

Grundsätzlich lassen sich diese Anomalien ohne einen Verzicht auf die Eigenschaft der In-Situ-Vergrösserung auch nicht auflösen. Selbst wenn die Unstetigkeitsstellen auf dem Projektionskörper entsprechend geglättet sind, treten früher oder später Stellen mit so hoher Steigung auf, dass die Darstellung dementsprechend verzerrt wird. Ursächlich sind sie darin begründet, dass die darzustellende Information die maximal mögliche Informationsdichte des Anzeigegeräts – in bestimmten Bereichen oder ingesamt – überschreiten kann, wenn an der Eigenschaft der In-Situ-Vergrösserung festgehalten wird.

6.2.3 Hybride Techniken

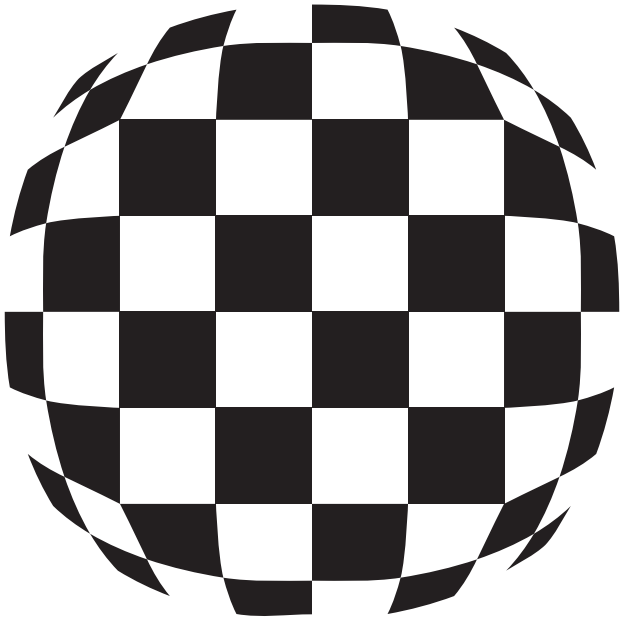


Abbildung 34: *Hybride Projektionstechnik mit einem Bereich in der Mitte, welcher linear projiziert wird, während an den Rändern nicht-linear gearbeitet wird.*
In diesem Beispiel ist die In-Situ-Vergrösserung weiterhin gegeben, was bei 'Fokus-auf-Fokus' zu den gleichen Problemen führt, wie bei den nicht-linearen Verfahren.

Hybride Techniken – in [Keah97] auch als *combined linear/non-linear transformations* und in [Leu+94] als *non-continuous magnification* bezeichnet – sind eine Kombination linearer und nicht-linearer Projektionstechniken mit dem Ziel, möglichst viele vorteilhafte Eigenschaften beider Verfahren zu vereinen. Das eigentlich Interessante an hybriden Techniken ist in diesem Zusammenhang weniger die Tatsache, dass (und wie) lineare und nicht-lineare Projektionstechniken kombiniert werden, sondern das Miteinander von perspektivloser und perspektivbehafteter Betrachtungsweise. Die Gesamtbetrachtungsweise ist zwar strenggenommen immer noch perspektivbehaftet aber partiell perspektivlos.

Bedingt dadurch, dass teilweise lineare Techniken Verwendung finden, werden bestimmte Bereiche verzerrungsfrei projiziert. Dies ist u.a. wenn Text dargestellt werden soll, eine wichtige Eigenschaft. Somit finden hybride Techniken beispielsweise häufig Verwendung, wenn (lesbarer) Text mitdargestellt werden soll (vgl. [Furn86] [Keah97] [Koik95] [Mös+97]).

Technisch gesehen werden bei hybriden Techniken in einer Sicht bestimmte Bereiche zwar immer noch unterschiedlich detailliert dargestellt, jedoch mindestens ein Bereich davon wird linear projiziert. Abb. 34 und 35 illustrieren hybride Projektionstechniken – wie üblich am Beispiel des Schachbrettmusters (und richtig, auch in Abb. 32 ist eigentlich hybrid projiziert).

Abb. 34 zeigt eine hybride Technik mit In-Situ-Vergrösserung und einem einzigen Bereich, wo linear projiziert wird. Abb. 35 zeigt eine hybride Technik, die bei einer Verfeinerung zusätzlichen Platz benötigt und somit die Eigenschaft der In-Situ-Vergrösserung nicht mehr gegeben ist. Der zusätzliche Platzbedarf bei Verfeinerung ist bei hybriden Techniken zwar bei weitem

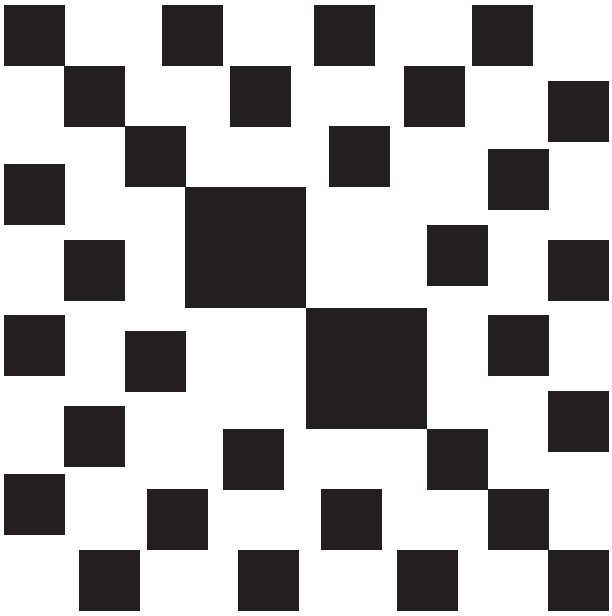


Abbildung 35: *Beispiel für eine hybride Projektionstechnik, bei welcher die In-Situ-Vergrösserung nicht mehr gegeben ist.*

nicht so hoch wie bei linearen Techniken – sonst würden sie wenig Vorteil gegenüber linearen Techniken bieten – aber es wird zusätzlicher Platz benötigt (vgl. Abb. 31, 32 und 35).

Der Verzicht auf eine In-Situ-Vergrösserung und der hierdurch bedingte zusätzliche Platzbedarf hat zwei Hauptkonsequenzen:

- Einerseits müssen – bedingt durch den zusätzlichen Platzbedarf bzw. den Verzicht auf die In-Situ-Vergrösserung – zusätzliche Navigationsarten bereitgestellt werden. Zumindest dann, wenn die Sicht so gross wird, dass nicht mehr genügend Platz auf dem Ausgabegerät vorhanden ist. Hierauf wird in Kapitel 6.3.2 eingegangen.
- Andererseits kann aber eben der Verzicht auf eine In-Situ-Vergrösserung dazu genutzt werden, den zusätzlichen Platz zu schaffen, welcher zwingend notwendig ist, um die bereits angesprochenen Anomalien aufzulösen.

Dies macht hybride Techniken insbesondere dann interessant, wenn es um eine möglichst anomalienfreie Generierung von Sichten für hierarchisch verschachtelte Strukturen geht, die nicht zuviel Detail erzeugen und gleichzeitig möglichst ähnlich zum Layout der alten Sicht sind, also viel Grösse, Anordnung, Proportionen, etc. der alten Sicht erhalten (vgl. Abb. 30 und 35). Natürlich wären auch klassische, nicht-lineare Techniken ohne In-Situ-Vergrösserung denkbar. Diese wären ebenso geeignet, um hierarchische Strukturen anomalienfrei darzustellen. Dass dies bisher nicht getan wurde, hat u.a. zwei Gründe. (1) Vergröberte und verfeinerte Sichten bleiben sich ähnlicher, wenn partiell linear projiziert wird, was dem Benutzer die Orientierung erleichtert. (2) Die Berechnung des Projektionskörpers sowie die eigentliche Projektion wird recht aufwendig und dies ganz besonders dann, wenn zusätzlich nicht nur ein Einfachfokus, sondern Mehrfachfoki möglich sein sollen (vgl. [Leu+94][Keah98]).

6.3 Hierarchische Modelle

Im Hinblick auf die Visualisierung hierarchischer Modelle oder Strukturen gibt es zwei Bereiche, welche für das Verständnis dieser Arbeit wichtig sind:

- Erstens, wie kann man zentrale Eigenschaften verschiedener Visualisierungstechniken bzw. -konzepte möglichst einfach festhalten, um so anschliessend einen Vergleich dieser Konzepte im Hinblick auf die Visualisierung von hierarchischen Objektmodellen im allgemeinen und ADORA-Modellen im besonderen zu ermöglichen? Hierauf wird in Kapitel 6.3.1 eingegangen.
- Zweitens, welche unterschiedlichen Arten von Navigation gibt es in hierarchischen Modellen? Hierauf wird in Kapitel 6.3.2 eingegangen.

6.3.1 Visualisierung

Hierarchische Modelle und deren Visualisierung – also auch ADORA-Modelle – haben viel mit Bäumen[3] bzw. Baumstrukturen zu tun. Offensichtlich kann mittels einer Baumstruktur die Teil-Ganzes-Hierarchie (siehe Abb. 36) wie auch die Beziehungs-Hierarchie für ADORA-Modelle beschrieben werden.

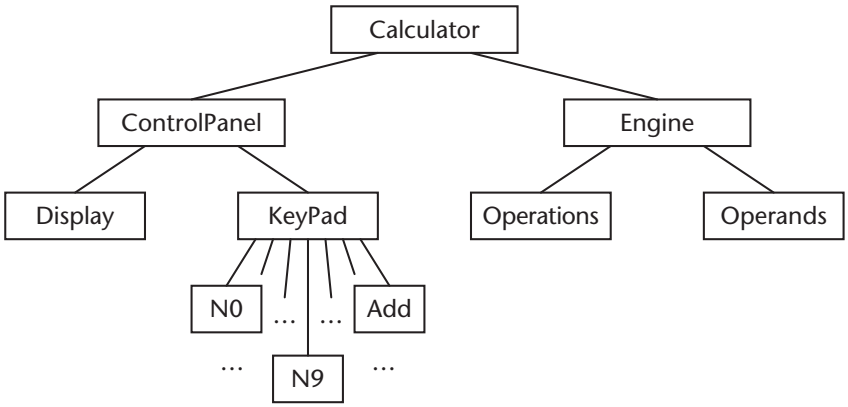


Abbildung 36: *Basisstruktur/Teil-Ganzes-Hierarchie des Taschenrechners (Calculator) dargestellt als Baumstruktur.*

Konzepte zur Visualisierung hierarchischer Strukturen unterscheiden sich insbesondere darin, auf welche Art und Weise bestimmte Teile der hierarchischen Struktur dargestellt werden; oder mit anderen Worten: welche Elemente wie in einer Sicht dargestellt werden. Die Interpretation oder Darstellung eines ADORA-Modells bzw. der Teil-Ganzes-Hierarchie als Baum bildet die Voraussetzung für eine sinnvolle Anwendung der beiden Parameter ‘Darstellungsebene’ und ‘Darstellungstiefe’ (vgl. [Kaen96]).

---

[3] Für diese Arbeit werden Kenntnis und Eigenschaften von Baumstrukturen als bekannt vorausgesetzt; d.h. Begriffe wie Knoten, Kante, Blatt, Wurzel, Höhe, Stufe etc. und deren Definition sollten bekannt sein; ggf. sei auf [Sedg92] verwiesen.

Die Parameter ‘Darstellungsebene’ und ‘Darstellungstiefe’ dienen in erster Linie dazu, strukturiert angeben zu können, welche Elemente eines Modells in einer bestimmten Sicht dargestellt werden und welche nicht. Dies ermöglicht u.a. Aussagen darüber, wie verschiedene Sichten zusammenhängen oder allgemeiner ausgedrückt, auf welche Art und Weise welche Teile einer hierarchischen Struktur dargestellt werden.

Auf der Basis dieser Parameter können später die speziellen Eigenschaften der verschiedenen Konzepte bei der Verfeinerung oder Vergröberung von Sichten relativ einfach und unkompliziert beschrieben werden. Dies vereinfacht die eigentliche Diskussion der unterschiedlichen Visualisierungskonzepte.

6.3.1.1 Darstellungsebene und Darstellungstiefe

Mit den Parametern *Darstellungsebene* (kurz DE) und *Darstellungstiefe* (kurz DT) können Teilstrukturen für eine gegebene Baumstruktur beschrieben werden. Interpretiert man eine Sicht als Teilstruktur einer durch ein hierarchisches Modell gegebenen Baumstruktur (vgl. Kapitel 6.1), so lässt sich mittels DE und DT angeben, welcher Modellauschnitt in einer Sicht repräsentiert wird.

Darstellungsebene

Ausgehend von der Wurzel einer Baumstruktur gibt die *Darstellungsebene* DE die Stufe s an, auf welcher diese Baumstruktur b betrachtet wird.

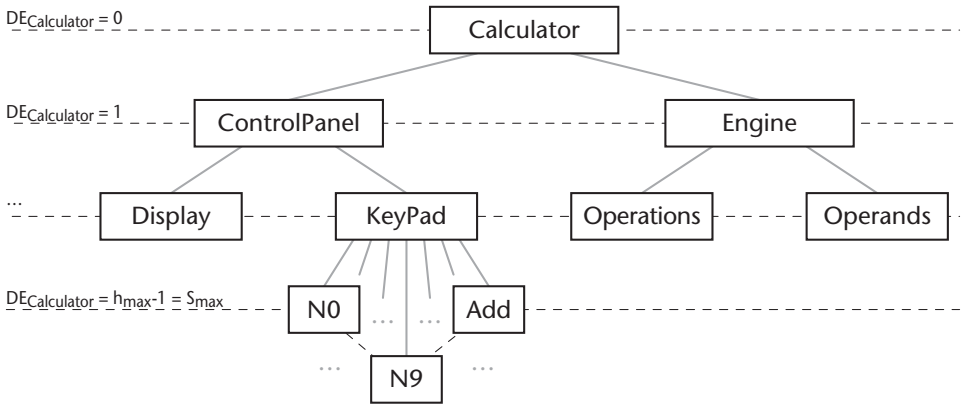


Abbildung 37: *Darstellungsebene am Beispiel des bekannten Taschenrechners. Ist die Darstellungsebene $DE_{Calculator} = 1$, so werden die Knoten unter der Wurzel, also die Komponenten ControlPanel und Engine betrachtet.*

Ist beispielsweise $DE = 0$, dann wird in dem entsprechenden Baum (initial) nur die Wurzel betrachtet. Ist $DE = 1$, dann werden die Knoten auf Stufe 1, also die Knoten unmittelbar unter der Wurzel betrachtet (siehe Abb. 37). Grundsätzlich gilt: $0 \leq DE \leq s_{max}$.

Anmerkung: Die Darstellungsebene wird u.a. auch als Detaillierungsgrad bezeichnet (vgl. [Kaen96]). Geht man davon aus, dass in einem Baum das abstrakteste Element als Wurzel platziert wird, bzw. die Anzahl der Elemente in einem Baum mit steigender Höhe zunimmt, kann

die Darstellungsebene als Parameter für den (initialen) Detailreichtum der entsprechenden Darstellung verwendet werden; daher auch die alternative Bezeichnung Detaillierungsgrad. Je weiter weg man von der Wurzelebene kommt, desto detaillierter ist die entsprechende Darstellung und desto höher wird der entsprechende Detaillierungsgrad.

Darstellungstiefe

Die Darstellungstiefe DT gibt für einen gegebenen Knoten oder eine gegebene Darstellungsebene an, wie tief die Teilbaumstruktur ist, welche von dort aus betrachtet werden soll. In Übereinstimmung mit [Kaen96] kann die Darstellungstiefe hier auf zwei verschiedene Arten angegeben werden:

- $DT_k = h_1$: Ausgehend vom Knoten k wird der dortige (Teil-)Baum der Höhe h_1 betrachtet (siehe Beispiel in Abb. 38a). Hier gilt u.a. $0 \leq DT_k < h - S_k$ wobei S_k als die Stufe s des Knotens k definiert sei.

  - $DT_k = 0$: Der dortige (Teil-)Baum hat die Höhe 0 und ist somit leer.
  - $DT_k = 1$: Nur der Knoten K ist Element des (Teil-)Baums.

- $DT_{DE} = h_2$: Es werden die Teilbäume all derjenigen Knoten, die sich auf Darstellungsebene DE befinden, bis zur Höhe h_2 betrachtet (siehe Beispiel in Abb. 38b). Dies ist eine Abkürzung. Anstatt jeden Knoten und die Höhe jeder Teilbaumstruktur einzeln anzugeben, kann dies auf diese Weise direkt gemacht werden. Analog gilt hier: $0 \leq DT_{DE} < h\text{-}DE$.

6.3.1.2 Sichten in hierarchischen Modellen

Wie bereits in Kapitel 6.1 angesprochen, ist eine Sicht ein Teilmodell eines Originalmodells. Ist das Original ein hierarchisch gegliedertes Modell, d.h. lässt es sich durch eine entsprechende Baumstruktur beschreiben, so lassen sich anhand von Darstellungsebene und Darstellungstiefe Begriffe wie ‘Verfeinerung’ und ‘Vergröberung’ von Sichten sowie Begriffe wie ‘Detailsicht’, ‘Vollsicht’, etc. einfach und präzise definieren.

Vergröberte Sicht

Eine Sicht S_1 heisst vergröbert gegenüber einer anderer Sicht S_2, wenn:

- entweder bezogen auf die Darstellungstiefe vergröbert wird; d.h. die Darstellungsebene der Sicht S_1 ist gleich der Darstellungsebene von S_2, aber die Darstellungstiefe für mindestens ein Element auf Darstellungsebene DE_{S1} ist kleiner als die Darstellungstiefe des entsprechenden Elements in S_2. Zusätzlich muss natürlich gelten, dass bei keinem Element auf DE_{S1} die Darstellungstiefe grösser ist als die Darstellungstiefe des entsprechenden Elements in S_2. Etwas formaler ausgedrückt gilt hier:

$$DE_{S1} = DE_{S2}$$
$$\wedge\ \exists\ e_1 \in S_1 \cap S_2 \bullet DT(S_1)_{e1} < DT(S_2)_{e1}$$
$$\wedge\ \neg\ \exists\ e_2 \in S_1 \cap S_2 \bullet DT(S_1)_{e2} > DT(S_2)_{e2}$$

Spezialfall: Für alle Elemente wird die Darstellungstiefe kleiner:

$$DE_{S1} = DE_{S2} \wedge \forall e \in S_1 \cap S_2 \bullet DT(S_1)_e < DT(S_2)_e$$
bzw. etwas vereinfacht: $DE_{S1} = DE_{S2} \wedge DT_{S1} < DT_{S2}$

- oder wenn bezogen auf die Darstellungsebene vergröbert wird; d.h. die Darstellungsebene der Sicht S_1 ist kleiner als die Darstellungsebene von S_2, wobei die Darstellungstiefe von S_1 gleich der Darstellungstiefe von S_2 ist. Hier gilt:

$$DE_{S1} < DE_{S2} \wedge \forall e_1 \in S_1, \forall e_2 \in S_2 \bullet DE(S_1)_{e1} = DE_{S1} \wedge DE(S_2)_{e2} = DE_{S2} \rightarrow DT(S_1)_{e1} = DT(S_2)_{e2}$$
bzw. einfacher notiert: $DE_{S1} < DE_{S2} \wedge DT_{S1} = DT_{S2}$

Anmerkung: Hier wird absichtlich der Begriff ‘Vergröberung’ gebraucht und nicht der Begriff ‘Abstraktion’. Ob eine Vergröberung auch gleichzeitig eine Abstraktion bildet, ist abhängig von der Bedeutung der Elemente eines Modells. Wenn davon ausgegangen werden kann, dass in der entsprechenden Baumstruktur ein Vaterknoten immer auch eine Abstraktion seiner Söhne darstellt, also die Merkmale der Söhne direkt oder in zusammengefasster Form abbildet, dann erzeugt die Vergröberung eine abstraktere Sicht auf die Struktur, andernfalls ist und bleibt es eben nur eine Vergröberung (vgl. [Joos99] [Stac73]).

Verfeinerte Sicht

Entsprechend der Vergröberung ist auch die Verfeinerung definiert. Eine Sicht S_3 heisst also verfeinert (detailliert) bezogen auf eine andere Sicht S_4, wenn sie:

- entweder bezüglich der Darstellungstiefe verfeinert ist; d.h. die Darstellungsebene der Sicht S_1 ist gleich der Darstellungsebene von S_2, aber die Darstellungstiefe für mindestens ein Element auf Darstellungsebene DE_{S1} ist grösser als die Darstellungstiefe des entsprechenden Elements in S_2. Es muss natürlich auch gelten, dass bei keinem Element der Darstellungsebene DE_{S1} die Darstellungstiefe kleiner ist als die Darstellungstiefe des entsprechenden Elements in S_2. Hier gilt:

$$DE_{S1} = DE_{S2} \wedge \exists e_1 \in S_1 \cap S_2 \bullet DT(S_1)_{e1} > DT(S_2)_{e1} \wedge \neg \exists e_2 \in S_1 \cap S_2 \bullet DT(S_1)_{e2} < DT(S_2)_{e2}$$

  Spezialfall: Für alle Elemente wird die Darstellungstiefe grösser:

$$DE_{S1} = DE_{S2} \wedge \forall e \in S_1 \cap S_2 \bullet DT(S_1)_e > DT(S_2)_e$$
  bzw. etwas vereinfacht: $DE_{S1} = DE_{S2} \wedge DT_{S1} > DT_{S2}$

- oder wenn ebenenweise verfeinert wird, d.h. die Darstellungsebene der Sicht S_1 ist grösser als die Darstellungsebene von S_2 und die Darstellungstiefe von S_1 ist gleich der Darstellungstiefe von S_2.

$$DE_{S1} < DE_{S2} \wedge \forall e_1 \in S_1, \forall e_2 \in S_2 \bullet DE(S_1)_{e1} = DE_{S1} \wedge DE(S_2)_{e2} = DE_{S2} \rightarrow DT(S_1)_{e1} = DT(S_2)_{e2}$$
bzw. einfacher notiert: $DE_{S1} < DE_{S2} \wedge DT_{S1} = DT_{S2}$

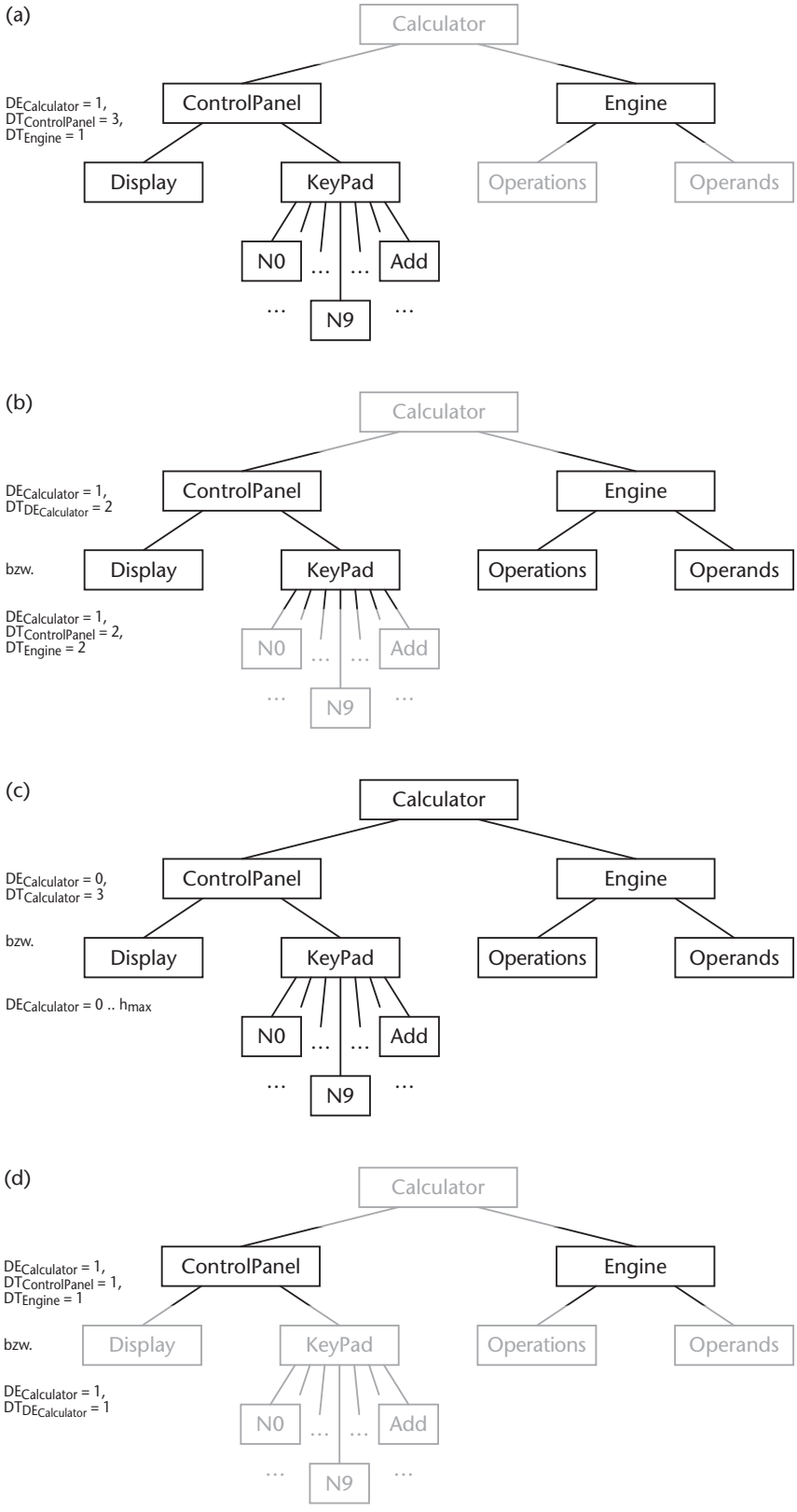


Abbildung 38: *Beispiele für die Parameter 'Darstellungsebene' und 'Darstellungstiefe' sowie für Vergröberung und Verfeinerung von Sichten.*
(a) ist eine Verfeinerung von (b) bzw. (d) ist eine Vergröberung von (a), (c) ist eine Totalsicht.

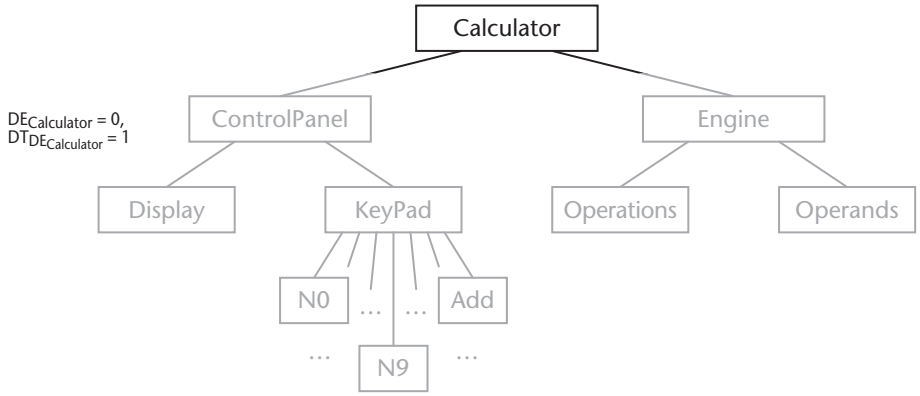


Abbildung 39: *Nicht kontexterhaltende Vergröberung von Abb. 38b*

kontexterhaltend vs. nicht-kontexterhaltend

Wird ausschliesslich bezüglich der Darstellungstiefe verfeinert, so ist die neue Sicht generell *kontexterhaltend*, d.h. sie zeigt die zusätzlichen Details im Kontext der alten Sicht.

Wird bezüglich der Darstellungsebene verfeinert, so ist die neue Sicht *nicht kontexterhaltend*. Die Details werden ohne direkten Bezug zur alten Sicht repräsentiert.

Abb. 38 zeigt Beispiele für vergröberte und verfeinerte Sichten. Abb. 38a ist eine kontexterhaltende Verfeinerung von Abb. 38d. Hier vergrössert sich ausschliesslich die Darstellungstiefe, während die Darstellungsebene gleich bleibt. Gleiches gilt für Abb. 38b und Abb. 38d. Hier wird für alle Elemente die Darstellungsebene grösser.

Abb. 38a ist weder eine Verfeinerung noch eine Vergröberung von Abb. 38b und umgekehrt. Hier vergrössert sich zwar die $DT_{ControlPanel}$ aber gleichzeitig verringert sich DT_{Engine}.

Abb. 39, mit $DE_{Calculator} = 0$, $DT_{DE\ Calculator} = 1$ ist eine Vergröberung von Abb. 38b. Hier wird bezüglich der Darstellungsebene vergröbert. Umgekehrt ist Abb. 38b eine nicht kontexterhaltende Verfeinerung von Abb. 39.

Totalsicht (siehe Abb. 38c)

Eine Sicht des ganzen Modells, also eine Sicht, bei welcher alle Elemente dargestellt werden, wird *Totalsicht* genannt und ist definiert durch: $DE = 0 \wedge DT_{DE} = h\text{-}1$. Somit ist eine Totalsicht eine Sicht, die sich nicht mehr weiter verfeinern lässt.

Detailsicht (siehe Abb. 38b)

Eine *Detailsicht* eines Modells, ist eine Sicht mit $DE > 0$.

Eine Detailsicht S_{detail} einer Sicht (bzw. eines Teilmodells) S ist eine Verfeinerung dieser Sicht. Mit dem Begriff Detailsicht wird üblicherweise eine Vergrösserung der Darstellungsebene – also ($DE_{S\ detail} > DE_S$) bei gleichbleibender Darstellungstiefe – verbunden. Dies muss jedoch

nicht zwingend so sein, d.h. eine Detailsicht kann durchaus auch über die Vergrösserung der Darstellungstiefe zustande kommen.

Beispiele: Abb. 38b ist eine typische Detailsicht der Sicht S_5, die über eine Erhöhung der Darstellungsebene erzeugt wird. Abb. 38a und 38b sind beides Detailsichten von 38d, die über eine Vergrösserung der Darstellungstiefe erzeugt werden. In Abb. 38a wird die Darstellungstiefe für einen Knoten schrittweise erhöht, während in Abb. 38b eine komplette Darstellungsebene hinzukommt.

Anmerkung: Ganz offensichtlich – eine Totalsicht ist nicht weiter verfeinerbar – kann eine Detailsicht keine Verfeinerung der Totalsicht des entsprechenden Modells sein. Eine Detailsicht ist aber auch keine Vergröberung einer Totalsicht.

Nullsicht

Die leere Sicht, also eine Sicht, in welcher überhaupt kein Element dargestellt ist, wird auch Nullsicht genannt. Eine *Nullsicht* ist eine Sicht mit $DE = 0$, $DT_{DE} = 0$.

Initialsicht

Um die Diskussion von Zoom-Verfahren (siehe Kapitel 6.3.2.1 sowie Kapitel 10) und Visualisierungskonzepten knapper und gleichzeitig weniger formal halten zu können, wird der Begriff der Initialsicht eingeführt.

Eine *Initialsicht* zeigt genau die Wurzel und ist somit eine Sicht mit $DE = 0$ und $DT_{DE} = 1$ (oder anders: sie ist kontexterhaltende Verfeinerung der Nullsicht um eine Stufe. Sie ist ferner eine Sicht, welche sich höchstens zur Nullsicht, also sinnvoll nicht weiter vergröbern lässt).

6.3.2 Navigation

Wie bereits in Kapitel 6.1 angesprochen, wird Navigation dann notwendig, wenn zu wenig Platz auf dem Ausgabe- bzw. Anzeigegerät vorhanden ist. Bei der Visualisierung von hierarchisch strukturierten Modellen ist Navigation jedoch nicht alleine wegen der Limitationen von Anzeigegeräten notwendig. Navigation kann auch dazu dienen, zu bestimmen, welche Elemente der Hierarchie angezeigt werden. Derartige Navigation unterstützt den Betrachter beim Verstehen eines Modells, indem ihm grundsätzlich die Möglichkeit geboten wird, abstrahierende Sichten zu erzeugen. Hierzu muss automatisch oder manuell bestimmt werden, welche Elemente der Hierarchie in einer Sicht darzustellen sind.

Bei der Visualisierung hierarchisch strukturierter Modelle wird somit zwischen zwei Arten von Navigation unterschieden: Der *physikalischen Navigation* und der *logischen Navigation*.

Physikalische Navigation

Physikalische Navigation wird notwendig, wenn die aktuelle Sicht – aus welchen Gründen auch immer – mehr Platz benötigt, als eigentlich auf dem Ausgabe- bzw. Anzeigegerät zur Verfügung steht.

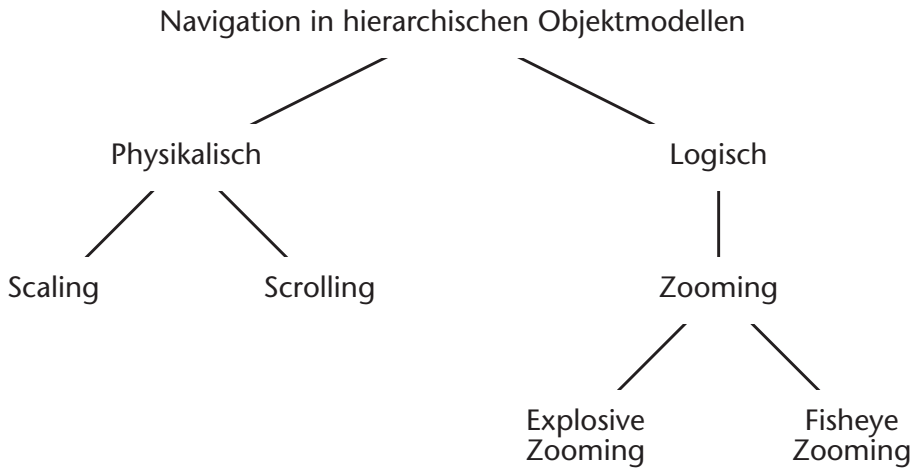


Abbildung 40: *Navigationsarten*

Soll die aktuelle Sicht auf dem zur Verfügung stehenden Platz dargestellt werden, muss die Grösse und/oder die Proportionierung der Sicht verändert werden (siehe hierzu Kapitel 6.2, Projektionstechniken). Soll die Grösse und die Proportionierung nicht verändert werden, so bleibt nichts anderes übrig, als lediglich einen Ausschnitt dazustellen. Zusätzlich benötigt es Bedienelemente, um den nicht dargestellten Teil des Modells sichtbar zu machen. Rollbalken (scroll bars) oder sog. Roam-Bars sind solche Bedienelemente. Mittels dieser Elemente wird der aktuell sichtbare Ausschnitt verändert oder verschoben [Bea+90] [Kaen96]. Diese Art der Navigation ist gut bekannt, da sie die normale Art und Weise beschreibt, wie in flachen Modellen navigiert wird. Daher wird hier nicht weiter auf physikalische Navigation eingegangen.

Logische Navigation

Bei hierarchischen Modellen kommt eine Dimension hinzu. Da das Modell eine hierarchische Struktur hat und die Art der Visualisierung desselben es ermöglichen sollte, entweder das ganze Modell oder auch Teile davon auf verschiedenen Stufen zu betrachten, muss[4] es eine Möglichkeit geben, durch die Hierarchie des Modells zu navigieren.

Logische Navigation in einer hierarchischen Struktur heisst zum einen, die aktuelle Position eines Elements im globalen Kontext der Hierarchie zu finden und zum anderen, den Fokus zu bestimmen oder zu verändern, so dass die gewünschten Elemente sichtbar werden. Hierfür wird unter *Ausnützung der Struktur des Modells* verfeinert oder vergröbert.

Verfeinern wird auch als zoom(ing)-in bezeichnet, vergröbern auch als zoom(ing)-out. Auf verschiedene Zoom-Verfahren[5] wird im folgenden Kapitel 6.3.2.1 eingegangen.

---

4 Eine etwas saloppe Anmerkung für den kritischen Leser. Natürlich könnte man das Modell auch ‘flach klopfen’ und dann konventionell visualisieren und physikalisch navigieren, aber wozu dann der ganze Aufwand mit der hierarchischen Strukturierung?

5 Hier wurde der englische Ausdruck beibehalten. Ebenso wurden die englischen Ausdrücke *full zoom* und *selective zoom* beibehalten.

6.3.2.1 Zooming als logische Navigation

Zooming-in bedeutet, dass mehr Details, d.h. die Elemente einer tieferen hierarchischen Stufe sichtbar werden. Zooming-out bedeutet, dass eine abstraktere Sicht auf die gerade sichtbaren Elemente produziert wird. Hier sollen nun die wichtigsten Zoom-Verfahren und deren Eigenschaften kurz diskutiert werden.

Explosive Zoom **(siehe Abb. 41)**

Explosive Zooming heisst, dass bei einem Zoom-Schritt der zu verfeinernde Knoten wie auch seine Nachfolger die bestehende Sicht komplett ersetzen; d.h. die Verfeinerung ersetzt die bestehende Sicht und wird im bestehenden Fenster explodiert. Alternativ kann auch ohne direkten Bezug zur alten Sicht ein neues Fenster geöffnet werden. Beschreibungen derartiger Zoom-Funktionen finden sich in [Bea+90] [Sc+93a].

Explosive Zoom ändert die Darstellungsebene und zeigt die Söhne der zu explodierenden Knoten mit Darstellungstiefe $DT = 1$ (bzw. $DT = c$ und $c << h$) an. Hier wird ebenenweise verfeinert bzw. vergröbert (siehe Kapitel 6.3.1.1), wobei aber zumindest im Fall der Verfeinerung nicht grundsätzlich alle Knoten der Darstellungsebene repräsentiert sein müssen.

Es wird nicht kontexterhaltend verfeinert – zumindest nicht solange die Darstellungstiefe konstant und deutlich kleiner als die Höhe der Struktur ist ($DT = c$ und $c << h$).

Entweder wird der globale Kontext ohne lokale Detailinformation betrachtet – *global context without local detail* – oder lokales Detail ohne Bezug zum globalen Kontext – *local detail without global context*. Explosive Zooming unterliegt somit eine perspektivlose Betrachtungsweise. Jedes Detail ist gleich wichtig und die Details werden entweder komplett, also für alle Elemente gleich oder gar nicht dargestellt. Zur Realisierung finden lineare Projektionstechniken (siehe Kapitel 6.2.1) Verwendung.

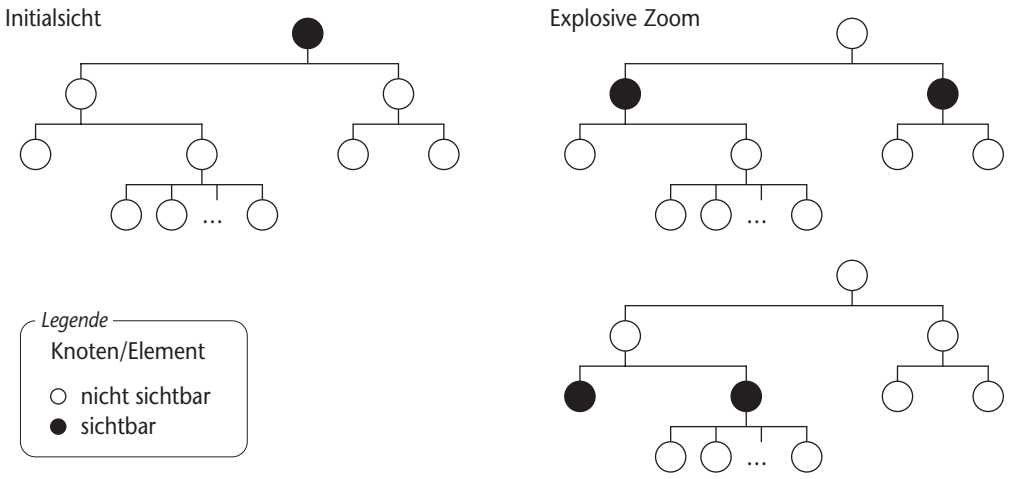


Abbildung 41: *Explosive Zoom am Beispiel der (aus Platzgründen schematisierten) Basisstruktur des Taschenrechners aus Kapitel 5.5.2.*

Anmerkung: Durch die Verwendung von Navigationsfenstern, Roam Bars, etc. kann der Kontextverlust bei explosive zooming zwar etwas gelindert werden, aber im Grundsatz ist es lediglich eine Frage der Anzahl an Verfeinerungsschritten ($DT << h$), bis der globale Kontext verloren geht.

Full Zoom **(siehe Abb. 42)**

Beim Full Zoom wird die Darstellungstiefe für die gegebene Darstellungsebene um eins bzw. um eine gegebene Konstante erhöht. Full Zoom stellt somit bis zur gegebenen Darstellungstiefe einfach alles dar – *full detail with full context*.

Full Zoom ist kontexterhaltend, betrachtet jedoch perspektivlos. Es werden entweder die Details aller Knoten oder gar keine angezeigt – wie gesagt: *full detail with full context*. Zur Realisierung finden lineare Projektionstechniken Verwendung.

Konzeptionell betrachtet besteht in hierarchischen Strukturen der Unterschied zwischen Explosive Zoom-Techniken und Full Zoom-Techniken darin, dass im Fall von Full Zoom-Techniken bei gleichbleibender Darstellungsebene die Darstellungstiefe verändert wird, während bei Explosive Zoom-Techniken bei konstant gehaltener Darstellungstiefe die Darstellungsebene verändert wird. Dies bewirkt den Kontextverlust bei Explosive Zoom-Techniken (vgl. Kapitel 6.3.1.1).

Selective Zoom/Fisheye Zoom **(siehe Abb. 43)**

Fisheye oder Selective Zooming dient dazu, Sichten zu erzeugen, bei welchen in einer einzigen Sicht die Elemente des Modells unterschiedlich detailliert dargestellt werden. Fisheye Zooming ist somit eine perspektivbasierte Betrachtungsweise, wo bei einem Zoom-Schritt bezüglich der Darstellungstiefe verfeinert oder vergröbert wird. Bei der Realisierung finden nicht-lineare Projektionstechniken (siehe Kapitel 6.2.2) Verwendung.

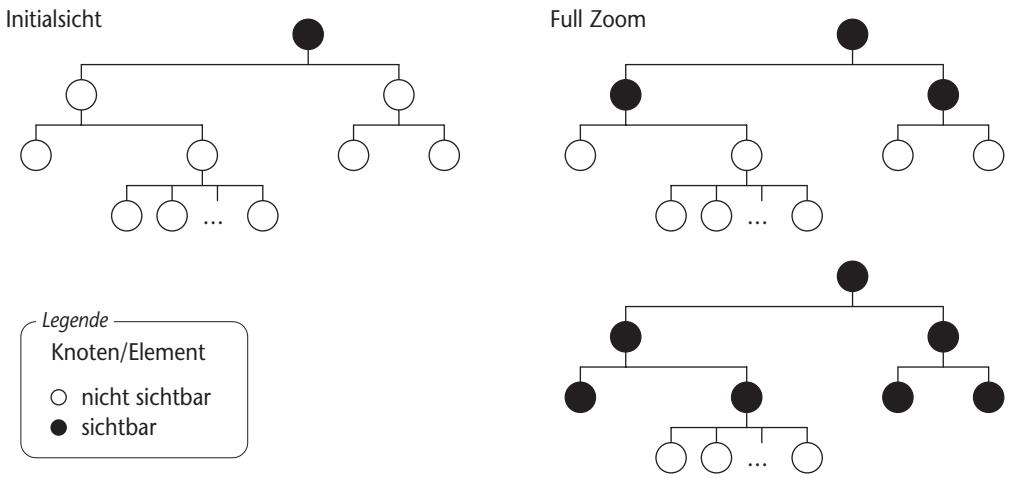


Abbildung 42: *Full Zoom; hier wird die Darstellungsebene beibehalten, während sich die Darstellungstiefe ändert.*

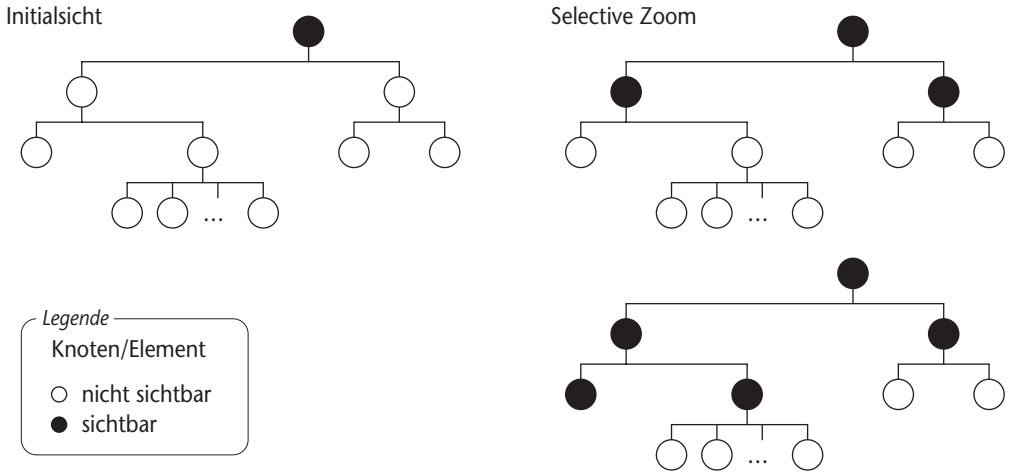


Abbildung 43: *Selective Zoom*

Beim Selective oder Fisheye Zoom wird davon ausgegangen, dass der Betrachter zwar die Übersicht behalten will, ihm aber zu einem bestimmten Zeitpunkt nicht alle Elemente gleich wichtig sind und dass es daher auch nicht wünschenswert ist, alle Elemente gleich detailliert darzustellen. Elemente, die genauer betrachten werden müssen, können detailliert dargestellt werden, während solche, die nur bedingt von Interesse sind, vergröbert dargestellt werden (*local detail and global context*).

Tab. 1 zeigt zusammengefasst die wichtigsten Eigenschaften der in diesem Kapitel vorgestellten Zoom-Funktionen.

Tabelle 1: *Eigenschaften der vorgestellten Zoom-Funktionen.*

	selektiv	**nicht selektiv**
kontexterhaltend	Fisheye Zoom, Selective Zoom	Full Zoom
nicht kontexterhaltend	Explosive Zoom	Explosive Zoom

7 VISUALISIERUNGSKONZEPT

Ein Visualisierungskonzept (vgl. [Kaen96]) beschreibt die grundlegenden Elemente zur Repräsentation eines Modells auf einem Anzeigegerät sowie die Integration dieser Elemente. Demnach soll ein Visualisierungskonzept für ADORA-Modelle beschreiben, welche Sichten auf das Objektmodell grundsätzlich bereitgestellt werden, wie diese Sichten miteinander zusammenhängen und wozu in diesen Sichten auf welche Art und Weise navigiert wird.

Für jedes der hier vorgestellten Konzepte werden drei Aspekte diskutiert:

- Präsentationsform
- Visualisierung
- Navigation

Präsentationsform

Die Präsentationsform gibt die *Art der Sichtweise* sowie die *äussere Form* eines Visualisierungskonzepts an.

Zur Beschreibung der Art der Sichtweise, wird u.a. angegeben, wie viele Dimensionen (1D, 2D, 3D, etc.) effektiv für die Präsentation verwendet werden und ob mit der entsprechenden Projektion generell perspektivlos oder perspektivbehaftet betrachtet wird.

Für diese Arbeit weitgehend uninteressant sind sämtliche 3D-Konzepte wie z.B. Cone-Trees[6]. Betrachtet werden ausschliesslich 2D-Konzepte. Daher wird für alle hier vorgestellten Konzepte die Anzahl Dimensionen nicht extra angegeben. Einer der Gründe hierfür liegt in der für Spezifikationsmodelle durchaus sinnvollen Anforderung (siehe R7 auf S. 103), dass Sichten so beschaffen sind, dass sie grundsätzlich auch als Skizze von Hand auf Papier gezeichnet werden können.

Die äussere Form gibt an, wie eine Projektion auf dem Anzeigegerät dargestellt wird – oder mit anderen Worten: Welche Sichten werden angeboten und wie stellt sich eine Sicht dem Betrachter dar, nachdem sie generiert ist. Bei der äusseren Form kann es sich um Ein-, oder Zwei- bzw. Mehrsichtformen handeln.

Da – zumindest für die Anwendung zur Visualisierung von ADORA-Modellen – Formen, die simultan mehr als zwei konzeptionell unterschiedliche Sichten anbieten, grundsätzlich von 2-Fensterkonzepten abgeleitet werden können, wird auf diese (3-Fensterkonzepte, 4-Fensterkonzepte, etc.) nicht weiter eingegangen.

---

6 Cone-Trees [Rob+91] sind ein Visualisierungskonzept, um hierarchisch strukturierte Information dreidimensional, perspektivbehaftet und in einem Fenster zu visualisieren. Es werden u.a. 3D-Effekte mit Animation kombiniert (siehe auch Kapitel 7.4.1).

Visualisierung

Unter dem Aspekt der Visualisierung wird für eine gegebene Präsentationsform beschrieben, welcher Modellausschnitt wie betrachtet wird und wie dieser Modellausschnitt in welcher Sicht dargestellt wird.

Eine Sicht ist ein spezielles Modell und stellt i.d.R. horizontal oder vertikal verkürzt ein anderes Modell dar. Eine Sicht bildet somit ein Teilmodell (siehe Kapitel 4.3, 6.1, 6.3). Die Verkürzung bringt für den Betrachter Orientierungsprobleme mit sich (vgl. [Bea+90] [Leu+94]). Für den Fall, dass konzeptionell unterschiedliche Sichten angeboten werden, wird in der Beschreibung der Visualisierung speziell darauf eingegangen, *was* in welcher Sicht dargestellt wird, um so die Orientierungsmöglichkeiten offenzulegen.

Neben den Orientierungsmöglichkeiten beschreibt die Visualisierung auch das *wie* der Projektion, d.h. die unterliegenden Projektionstechniken (siehe Kapitel 6.2), welche zur Generierung einer bestimmten Sicht verwendet werden. Beispielsweise mit welcher nicht-linearen Technik eine perspektivbasierte Betrachtung realisiert wird. Grundsätzlich muss die Projektionstechnik so beschaffen sein, dass sie in der Lage ist, der Art der Sichtweise gerecht zu werden. Pathologische Kombinationen – wie beispielsweise perspektivlos mit nicht-linearen Projektionen – werden nicht betrachtet.

Navigation

Der Punkt Navigation beschreibt die Navigationsinstrumente, welche zur Verfügung stehen sowie deren Zusammenhang mit Präsentation und Visualisierung. Eingegangen wird hierbei jeweils auf die verschiedenen Möglichkeiten, die in einem Visualisierungskonzept zur physikalischen und logischen Navigation vorgesehen sind.

Da es in dieser Arbeit um die Visualisierung von Hierarchien geht, wird davon ausgegangen, dass stets die Möglichkeit zur logischen Navigation besteht. Konzepte ohne die Möglichkeit zur logischen Navigation machen für die Visualisierung von Hierarchien nur eingeschränkt Sinn und werden daher im Rahmen dieser Arbeit auch nicht betrachtet.

Anmerkung: Ein Grossteil der Navigationsinstrumente – wie beispielsweise Rollbalken oder Scrollbars, Roam Bars, etc. – wird in dieser Beschreibung als bereits bekannt vorausgesetzt und daher nicht weiter erläutert; ggf. sei hierfür auf [Ilg+87] [Ilg+88] [Bea+90] [Herc94] oder [Kaen96] verwiesen.

7.1 Existierende Konzepte und Techniken

In [Kaen96] wurden bestehende Visualisierungskonzepte bezogen auf ihre Anwendbarkeit für ADORA-Modelle evaluiert. Hierbei wurden stellvertretend für viele ähnliche Ansätze drei typische Konzepte untersucht (siehe Tab. 1). Die Betrachtung dieser drei Konzepte ist im Kontext dieser Arbeit ausreichend. Andere Konzepte lassen sich mehr oder weniger direkt aus einem der drei hier vorgestellten Konzepte ableiten bzw. bilden eine Variante. Auf diese Ableitungen oder Varianten wird bei der Vorstellung des jeweiligen Konzepts speziell eingegangen.

Tabelle 2: *Übersicht über die in [Kaen96] evaluierten Visualisierungskonzepte.*

		(klassisches) 1-Fensterkonzept	**2-Fensterkonzept**	**Fischaugenkonzept**
Präsentationsform	**Art der Sichtweise**	perspektivlos mit einer Sichtart	perspektivlos (bzw. bispektiv) mit untersch. Sichtarten	perspektivbehaftet
	äussere Form	Arbeitsfenster mit benutzergenerierter Sicht	Übersichtsfenster mit Totalsicht + Arbeitsfenster mit benutzergenerierter Sicht	Arbeitsfenster mit benutzergenerierter Sicht
Visualisierung	**Orientierungsmöglichkeiten**	lokales Detail	lokales Detail + separate Totalsicht	lokales Detail und globaler Kontext
	Sichtengenerierung	lineare Projektion	lineare Projektion	nicht-lineare Projektion mit oder ohne In-Situ-Vergrösserung
Navigation	**physikalisch**	Rollbalken + Zoom	Rollbalken, Roam	–
	logisch	Explosive Zoom + Full Zoom	Explosive Zoom	Selektive Zoom/ Fisheye Zoom

Zum weiteren Vorgehen: Im Anschluss, also in Kapitel 7.2 - 7.4 werden die in der von Kaenel'schen Evaluierung untersuchten Visualisierungskonzepte beispielhaft und kurz vorgestellt und dann ihre Vor- und Nachteile zur Visualisierung von ADORA-Modellen schrittweise diskutiert.

Die Vorstellung dieser drei Konzepte erfolgt bewusst bevor die eigentlichen Anforderungen an ein Visualisierungskonzept für ADORA-Modelle aufgeführt werden, da sich ein Teil der Anforderungen aus der Diskussion der drei Konzepte ableiten lässt.

Anmerkung: In dieser Arbeit (und auch in [Kaen96]) nicht betrachtet wurden Ansätze, welche die Anordnung der Elemente in einer Sicht vollständig automatisch generieren – wie beispielsweise Graphlayouter, Treemaps [Joh+91], etc. – also Konzepte, die dem Benutzer keine oder nur sehr wenig Möglichkeiten bieten, auf die Anordnung der Elemente in einer Sicht irgendeinen Einfluss zu nehmen. Begründet ist dies durch die fehlende Sekundärnotation; siehe hierzu Kapitel 9.1 und [Petr95]. Ferner wird auch nicht darauf eingegangen, wie mittels kleinerer oder grösserer Animation(en) das Verständnis von ADORA-Modellen verbessert werden kann. Aus unterschiedlichsten Gründen nicht behandelt werden ausserdem Ansätze, wo visuell in einer dritten Dimension gearbeitet wird (vgl. hierfür auch Kapitel 9.1).

7.2 1-Fensterkonzepte

… oder auch perspektivlose 1-Sichtkonzepte

Präsentationsform

Wie der Name schon andeutet, bietet das (klassische) 1-Fensterkonzept (siehe Abb. 44) ein einziges Fenster zur Aufnahme einer einzigen Sicht an. Neben der Arbeit von [Kaen96] werden 1-Fensterkonzepte u.a. von [Bea+90] [Leun89] [Hol+89] beschrieben. Das 1-Fensterkonzept wird hier stellvertretend für Konzepte angeführt, welche genau eine Art von perspektivloser Sicht bereitstellen.

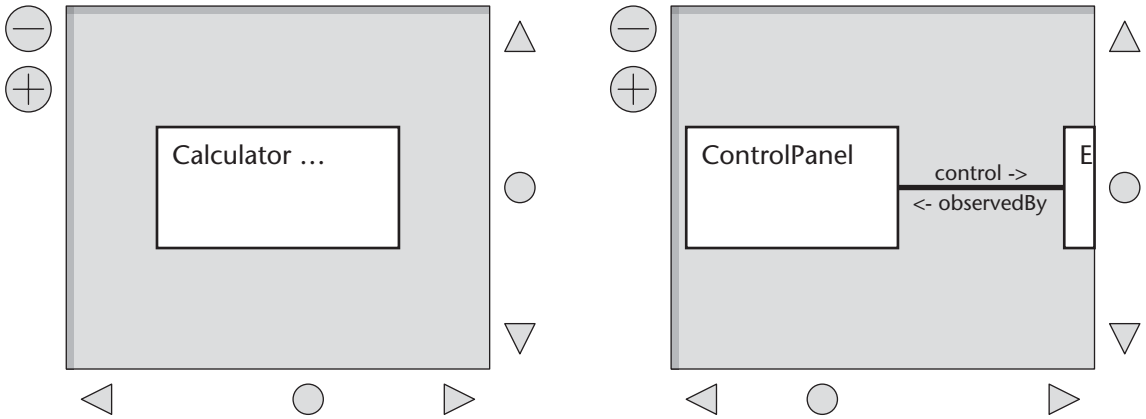


Abbildung 44: *Beispiel für ein 1-Fensterkonzept mit Explosive Zoom angewandt zur Visualisierung des Taschenrechners (zur Erinnerung: Taschenrechnerbeispiel in Kapitel 5.2.2 eingeführt). Auf der linken Seite ist die Initialsicht des Taschenrechners dargestellt. Die Detailsicht auf der rechten Seite wird durch Verfeinerung des Objekts Calculator erzeugt.*

Visualisierung

Zu einem Zeitpunkt kann nur eine Sicht bereitgestellt werden. Zur Erzeugung dieser Sicht werden lineare Projektionen verwendet. Lediglich der gerade in der aktuellen Sicht gezeigte Ausschnitt des Modells kann zur Orientierung benutzt werden. Es ist somit nicht unmittelbar offensichtlich, welcher Teil des Modells dargestellt ist.

Navigation

Physikalisch wird über Rollbalken/Scrollbars navigiert. Logisch navigiert wird über einen Explosive Zoom oder Full Zoom. Die über diese Navigation qualifizierten Elemente dienen als Basis für die Generierung der Sicht.

Beispiele für Zoom-Funktionen finden sich u.a. in [Bea+90] [Sc+93a]. Die dort beschriebenen Funktionen sind Explosive Zoom-Funktionen, welche nicht unbedingt für hierarchische Objektmodelle geeignet sind, da die Darstellungstiefe immer DT=1 beträgt. Zoom-Funktionen mit DT=1 führen zwar zu überschaubaren Sichten, blenden aber den Kontext vollständig aus. Es werden wie in Abb. 44 immer nur Objekte ohne ihre Komponenten dargestellt.

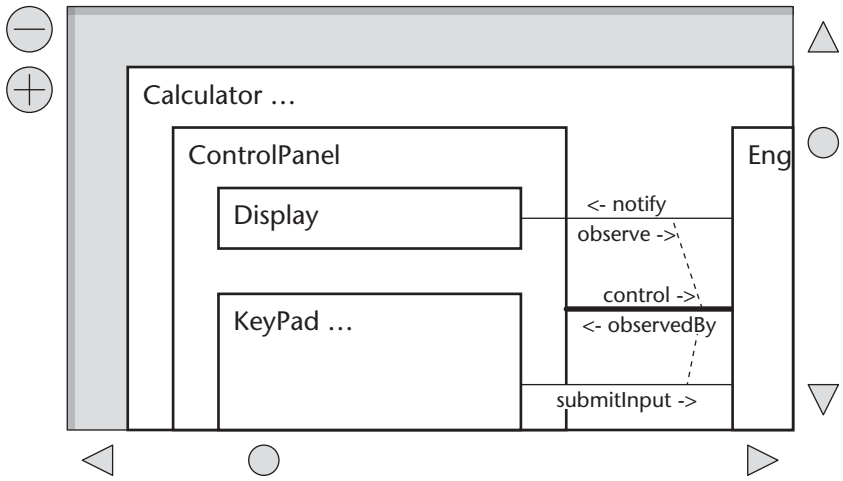


Abbildung 45: *Beispiel für ein 1-Fensterkonzept und eine Zoomfunktion mit min. DT=3.*

Besser geeignet für hierarchische Strukturen sind Zoom-Funktionen mit Darstellungstiefe DT≥2 wie sie in Abb. 45 zu sehen sind. Die Sichten, die auf Basis dieser Funktionen entstehen, bieten mehr Kontext und sind übersichtlich, solange die Darstellungstiefe nicht allzu gross ist.

Grundsätzlich kann sich der Benutzer des 1-Fensterkonzepts nur über eine Sicht orientieren. Diese Sicht ist horizontal und vertikal verkürzt. Somit muss bereits zur Lösung des 'Wo-bin-ich-Problems' (siehe [Leun89]) navigiert werden. Der Benutzer muss Kenntnis des Modells haben, um gezielt physikalisch oder logisch navigieren zu können oder er muss navigieren, um sich dieses Wissen überhaupt erst zu verschaffen. Findet Explosive Zoom Verwendung, verschärft sich diese Problematik zusätzlich. Nun muss zuerst vergröbert werden, um dadurch die Sicht zu erzeugen, die eben Voraussetzung dafür ist, um im Anschluss an der gewünschten Stelle logisch navigieren, sprich verfeinern, zu können. Ganz abgesehen davon, dass damit der Zoom-Funktion eine verwirrende Doppelbedeutung zukommt – de facto muss über ein logisches Navigationsinstrument physikalisch navigiert werden (quasi physikalische Navigation über ein logisches Navigationsinstrument) – hat dies erheblichen kognitiven Ballast bei der Navigation zur Konsequenz.

Wie erwähnt (siehe Präsentationsform), besteht das zentrale Merkmal des 1-Fensterkonzepts darin, dass nur eine perspektivlose Sicht, oder etwas allgemeiner, nur eine Art von perspektivloser Sicht existiert. Wird an diesem Merkmal festgehalten, so kann das o.g. Problem auf zwei Arten angegangen werden: Es kann (1) eine Zoom-Funktion gewählt werden, die kontexterhaltend arbeitet und es können (2) zusätzliche Sichten angeboten werden – wohlgemerkt zusätzliche gleichartige Sichten und nicht andere Arten von Sichten. Auf (1) wird nachfolgend eingegangen, auf (2) im Kapitel 7.2.1.

Beständig kontexterhaltende Full-Zoom-Funktionen – also Zoom-Funktionen, bei welchen bei gleichbleibender Darstellungsebene die Darstellungstiefe erhöht wird – weisen das oben diskutierte Problem der 'Orientierung im Modell bei Kontextverlust' nicht auf. Dafür haben Full-

Zoom-Funktionen zwei andere Probleme. Das eine Problem ist der durch die lineare Projektion bedingte immense Platzbedarf bei grosser Darstellungstiefe (vgl. hierzu Kapitel 6.2.1), welcher in zunehmendem Mass physikalische Navigation notwendig macht (vgl. hierzu auch den Abschnitt ‘Navigation’ auf S. 87). Das andere Problem ist die im Zuge einer sukzessiven Verfeinerung erzeugte ‘Detailflut’, die dazu führt, dass eine Sicht mit wachsender Darstellungstiefe immer unübersichtlicher wird. Full-Zoom Funktionen sind daher nur dann eine effektive Lösung des Problems, wenn die Darstellungstiefe einer Sicht modell- oder sprachbedingt nicht allzu gross werden kann.

7.2.1 Varianten

Weicht man das Kriterium ‘nur eine perspektivlose Sicht’ in Richtung ‘nur eine *Art* von Sicht’ etwas auf, so lassen sich vom hier vorgestellten 1-Fensterkonzept bestimmte Mehrsicht- oder Mehrfensterkonzepte ableiten, in welchen bei jedem Zoom-Schritt eine zusätzliche (Elementar-)Sicht geöffnet wird. Die Gesamtsicht setzt sich dann aus diesen gleichartigen Elementarsichten zusammen. Solange auf dem Ausgabegerät genügend Platz vorhanden ist, entschärft sich mit dieser Variante das o.g. Problem ‘Orientierung im Modell bei Kontextverlust’, da es keinen Kontextverlust im eigentlichen Sinn mehr gibt. Allerdings werden – ähnlich wie beim in Kapitel 7.3 vorgestellten 2-Fensterkonzept – erhöhte kognitive Anforderungen gestellt, da der Betrachter nun Integrationsarbeit für die Elementarsichten zu leisten hat (vgl. [Leun89]).

Steht nicht genügend Platz für die zusätzlichen Sichten zur Verfügung, so kann eine der drei folgenden Strategien zur Anwendung kommen:

(1) Es wird zugelassen, dass sich Sichten überdecken.

(2) Es kann virtueller Platz für die zusätzlichen Sichten geschaffen werden.

(3) Es können bestimmte Sichten automatisch geschlossen werden.

Mit Strategie 1 oder 2 wären prinzipiell beliebig viele (gleichartige) Sichten möglich und mit Strategie 3 zumindest eine gewisse Auswahl. Dies bietet den Vorteil, dass Sichten, welche wechselseitig benötigt werden, nicht beständig durch logische Navigation neu erzeugt werden müssen, was bei der ‘normalen’ 1-Sicht-Variante der Fall war. Dafür muss bei Strategie 1 oder 2 zusätzlich physikalisch navigiert werden; entweder durch die sich überdeckenden Sichten oder durch den zusätzlichen virtuellen Platz. Verliert der Benutzer bei Strategie 1 die Übersicht über die sich überdeckenden Sichten, bringen sie keinen Nutzen und haben den selben Kontextverlust wie die ‘normale’ 1-Sicht-Variante zur Folge. Strategie 2 lässt sich im Prinzip durch ein 1-Sicht-Konzept mit Full-Zoom ersetzen (vgl. S. 77), für welches die kognitiven Anforderungen geringer sind, da der Betrachter weniger Integrationsarbeit leisten muss. Strategie 3 lässt sich durch ein 1-Sicht-Konzept mit Explosive Zoom und entsprechender Darstellungstiefe ersetzen, welches ebenfalls geringere kognitive Anforderungen stellt (vgl. S. 76).

Da bedingt durch die Strategie der Sichtengenerierung echte Übersichten nicht möglich sind, lässt sich das (nachfolgend in Kapitel 7.3 vorgestellte) 2-Fensterkonzept nicht vom 1-Fensterkonzept oder einer der Varianten ableiten.

7.3 2-Fensterkonzepte

… oder auch perspektivlose 2-Sichtkonzepte

Präsentationsform

Im 2-Fensterkonzept wird der Kontextverlust, das Hauptproblem des 1-Fensterkonzepts, angegangen, indem spezialisierte Sichten geschaffen werden. Hierzu werden simultan mindestens zwei Sichten mit *unterschiedlichen* Sichtarten angeboten. Meistens sind diese Sichten auf zwei Fenster verteilt – daher die Bezeichnung 2-Fensterkonzept. Das eine Fenster bietet eine Übersicht, möglichst in Form einer Totalsicht (siehe Abb. 46a oder Abb. 46b), um so den kognitiven Ballast zur Lösung des 'Wo-bin-ich-Problems' gering zu halten. Das andere Fenster bietet eine Arbeits- oder Hauptsicht in Form einer Detailsicht, in welcher die eigentliche Arbeit am Modell verrichtet wird (siehe Abb. 46c, vgl. auch [Bea+90] [Leun89]).

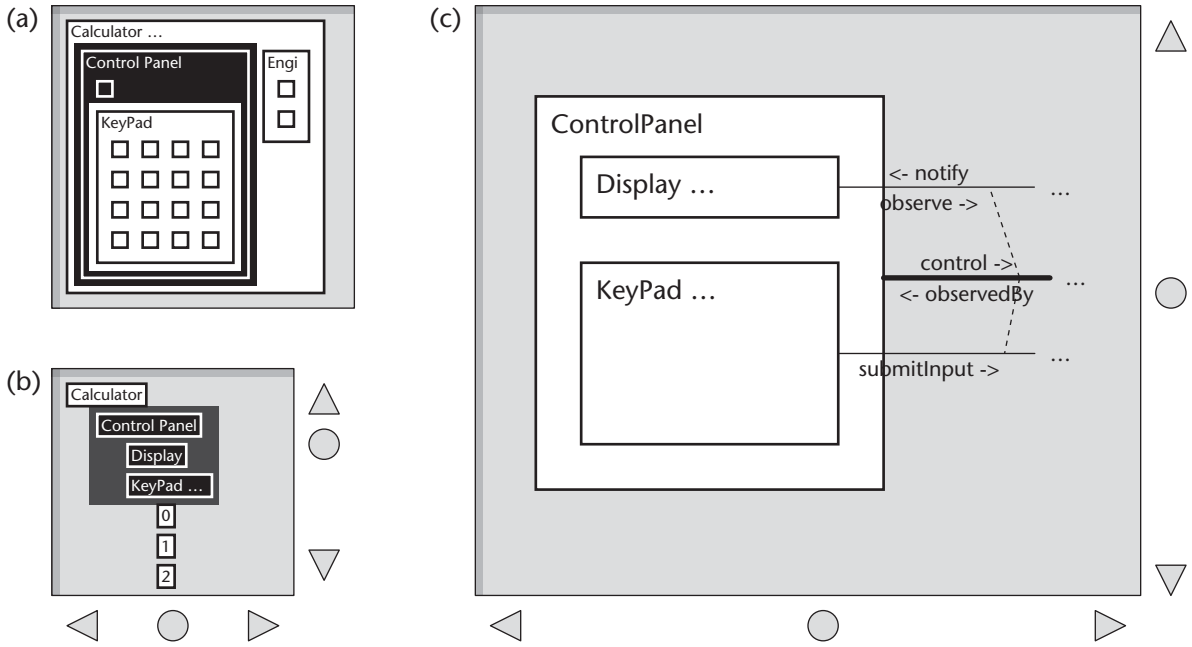


Abbildung 46: *Beispiel für ein 2-Fensterkonzept; (a) und (b) zeigen zwei Möglichkeiten für Übersichten, (c) zeigt die Haupt- bzw. Arbeitssicht.*
Die Arbeitssicht zeigt hierbei ein Objekt mit DT=2 an. Welches Objekt in der Arbeitssicht angezeigt wird, bestimmt die Auswahl in der Übersicht. Die in der Haupt- bzw. Arbeitssicht aktuell dargestellten Objekte (ControlPanel, Display und KeyPad wg. DE=1, $DT_{ControlPanel}$=2) sind in der Übersicht invertiert dargestellt.

Visualisierung

Zur Erzeugung beider Sichten werden lineare Projektionen verwendet. Die einzelne Sicht ist für sich betrachtet perspektivlos. Nimmt man beide Sichten zusammen, wird das Modell aus zwei fixen Blickwinkeln, also bispektiv betrachtet.

Um die Probleme des 1-Fensterkonzepts mit dem Platzbedarf für mehrere Sichten zu vermeiden (vgl. Kapitel 7.2.1), wird das Modell in einer Übersicht stark verkleinert bzw. komprimiert

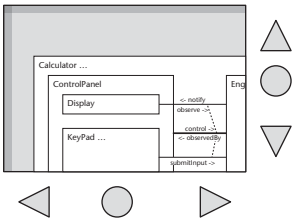


Abbildung 47: *Übersicht mittels geometrischer Projektion.*

dargestellt. Da die Übersicht nur der Orientierung dienen muss, ist dies unproblematisch. Die Übersicht zeigt hierbei eine stark verkleinerte Totalsicht oder zumindest eine Sicht mit hoher Darstellungstiefe und geringer horizontaler sowie vertikaler Verkürzung des darzustellenden Modells. Liefert eine rein geometrische Projektion à la Abb. 47 nicht das gewünschte Resultat, was bei tiefen Hierarchien der Regelfall ist, so wird ein auf Schemaebene stark horizontal verkürztes Modell gezeigt (siehe ggf. Stichwort *Sicht* in Kapitel 6.1).

Da sich der Benutzer über die Übersicht orientieren können sollte, ist in der Haupt- bzw. Arbeitssicht nur noch eine Sicht mit geringer Darstellungstiefe notwendig, z.B. DT=2, welche typisch über einen Explosive Zoom erzeugt wird. Abb. 46c zeigt eine solche Sicht. Hier wird eine weitere Facette des Problems ‘Kontextverlust’ offensichtlich. Wenn nur selektiv ein Objekt und dessen Komponenten angezeigt werden, wohin führen dann die Beziehungen, die von diesem Objekt (hier ControlPanel) ausgehen? Im Beispiel in Abb. 46c wäre es zwar noch gut vorstellbar, das korrespondierende Objekt (Engine) auch darzustellen. Aber was tun, wenn die Darstellungstiefe der Sicht kleiner ist als die Hierarchiestufen, über welche eine Beziehung geht? Beispielsweise wenn eine Beziehung von einer der Tasten (siehe Abb. 17 auf Seite 37) der Tastatur zum Rechenwerk (Engine) existieren würde.

Orientieren kann sich der Betrachter an beiden Sichten. Allerdings obliegt ihm hierbei die Integrationsarbeit vollständig, oder um es mit Furnas Worten auszudrücken:

> *Where does the small view fit in the big picture and what visible features correspond? [Furn81]*

Um herauszufinden, wie Übersicht und Detailsicht zusammenhängen, müssen die Sichten wechselseitig betrachtet werden. Ist die Übersicht zu abstrakt, kann der Betrachter den Zusammenhang zur Arbeitssicht nicht mehr ohne weiteres herstellen. Zeigt die Übersicht zuviel Detail, wird sie unübersichtlich, was ihren Nutzen als Übersicht entsprechend einschränkt. Diese Problematik verschärft sich bei grossen Modellen, da das Abstraktionsniveau von Übersicht und Detailsicht immer weiter auseinanderläuft bzw. laufen muss, wenn noch von einer Übersicht die Rede sein soll.

Navigation

Physikalisch wird auch beim 2-Fensterkonzept über Scrollbars/Rollbalken navigiert, zusätzlich ist Roam Navigation [Bea+90] über das Übersichtsfenster möglich. Ansonsten ist die Navigation in der Haupt- bzw. Arbeitssicht analog zum 1-Fensterkonzept (siehe Kapitel 7.2).

In der Übersicht selbst muss nicht physikalisch navigiert werden, solange genügend Platz vorhanden ist, um eine Totalsicht des Modells darzustellen (siehe Abb. 46a). Ist dies nicht der Fall (siehe Abb. 46b und Abb. 47), so muss es entweder möglich sein, die Grösse der Übersicht zu verändern oder es müssen auch hier entsprechende Navigationsinstrumente zur physikalischen Navigation vorhanden sein. Alternativ ist es auch denkbar, eine Übersicht anzubieten, welche rekursiv arbeitet, d.h. keine Totalsicht darstellt, sondern in Abhängigkeit von der Darstellungsebene der Arbeitssicht arbeitet (siehe Kapitel 7.2.1, Strategie 3; vgl. auch [Bea+90]). Bei geeigneter Wahl der Darstellungstiefe wäre hier keine physikalische Navigation notwendig, jedoch um den Preis, dass u.U. analog zur Navigation im 1-Fensterkonzept (siehe Kapitel 7.2) stattdessen logisch navigiert werden muss, um ein physikalisches Navigationsziel zu erreichen.

Die logische Navigation findet beim 2-Fensterkonzept über einen Explosive Zoom statt. Die Darstellungsebene bzw. die darzustellenden Elemente können hierfür in der Übersicht ausgewählt werden. Auf Basis dieser Auswahl wird dann die in der Arbeitssicht darzustellende Sicht generiert. Die Übersicht bleibt möglichst unverändert.

Auch für das 2-Fensterkonzept ist ein Full Zoom denkbar. Er hat die selben Nachteile wie bereits erwähnt (vgl. vorheriges Kapitel 7.2), bringt nur nicht mehr den gleichen Nutzen wie beim 1-Fensterkonzept, da beim 2-Fensterkonzept im Übersichtsfenster naturgemäss bereits der Kontext der Arbeitssicht dargestellt wird. Wesentlich eleganter ist hier eine Variante, die flexibel bezüglich Darstellungsebene und -tiefe arbeitet und es dem Benutzer erlaubt, in der Übersicht mehrere Elemente gleichzeitig anzuwählen. Das System determiniert dann in

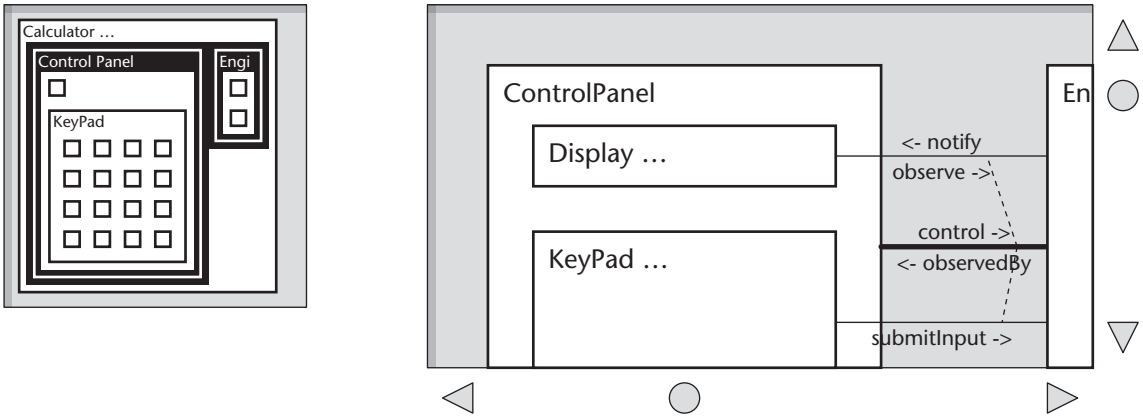


Abbildung 48: *Beispiel für eine ROAM-Navigation, die es erlaubt, mehrere Elemente gleichzeitig auszuwählen. Die Auswahl von ControlPanel und Engine führen zu einer Sicht mit DE=1, DT = 2. Würde nun in der Übersicht lediglich eines der Objekte Display, KeyPad, Operands oder Operations angewählt, würde sich die Darstellungstiefe der Sicht auf DT=3 erhöhen, und all diese Objekte würden dargestellt (auf diese Darstellung wurde aus Platzgründen verzichtet).*

Abhängigkeit von dieser Auswahl die entsprechende, minimale Darstellstellungsebene sowie die maximale Darstellungstiefe für eine entsprechende Detailsicht und stellt diese als neue Arbeitssicht dar. Werden Elemente selektiert, die in der Objekthierarchie weit auseinanderliegen, degeneriert die Arbeitssicht fast zur Totalsicht, was die Übersicht eigentlich überflüssig macht und die bereits bekannten Probleme schafft, die ein Full-Zoom mit sich bringt. Abhilfe kann hier durch zusätzliche Arbeitssichten geschaffen werden, so dass nicht mehr zwingend *eine* Sicht mit minimaler Darstellungsebene und maximaler Darstellungstiefe erzeugt werden muss. Hierauf wird im Kapitel 7.3.1 eingegangen.

2-Fensterkonzepte lösen viele der durch Kontextverlust entstehenden Probleme, die bei 1-Fensterkonzepten früher oder später auftreten. Potentiell Probleme gibt es mit 2-Fensterkonzepten bei der Darstellung bestimmter, kooperierender Objekte und bei der zu Orientierungszwecken durchzuführenden Integration von Übersicht und Arbeitssicht durch den Betrachter.

7.3.1 Varianten

Auch beim 2-Fensterkonzept sind diverse Varianten denkbar. So können analog zu den Varianten beim 1-Fensterkonzept zusätzliche Sichten angeboten werden. Während es nur selten sinnvoll ist, zusätzliche Übersichten anzubieten – hier sollte eher die Sichtart variiert werden – kann durch zusätzliche Arbeitssichten die Darstellung kooperierender Objekte verbessert werden. Führt die oben beschriebene Variante der Mehrfachselektion in der Übersicht nicht zu zufriedenstellenden Ergebnissen, weil damit gerechnet werden muss, dass Objekte entweder räumlich oder in der Objekthierarchie zu weit auseinanderliegen, können zusätzliche Arbeitssichten angeboten werden. Dies verhindert, dass die Arbeitssicht zur Totalsicht degeneriert, bringt aber zusätzlichen, kognitiven Ballast mit sich. Oft sind nur zwei der drei Sichten durch den Benutzer direkt nutzbar (vgl. [Leun89]). Kooperieren Objekte nicht direkt, sondern indirekt über andere Objekte, so ist dies nicht ohne weiteres erkennbar und erfordert zusätzlich Navigation.

7.4 Fischaugenkonzepte

... oder auch perspektivbasierte 1-Sichtkonzepte

Präsentationsform

Das Fischaugenkonzept bietet ein Fenster mit einer Fischaugensicht. Diese Sicht zeigt eine vertikale Vergröberung des Modells mit einer Balance zwischen lokalem Detail und globalem Kontext an (siehe Abb. 49).

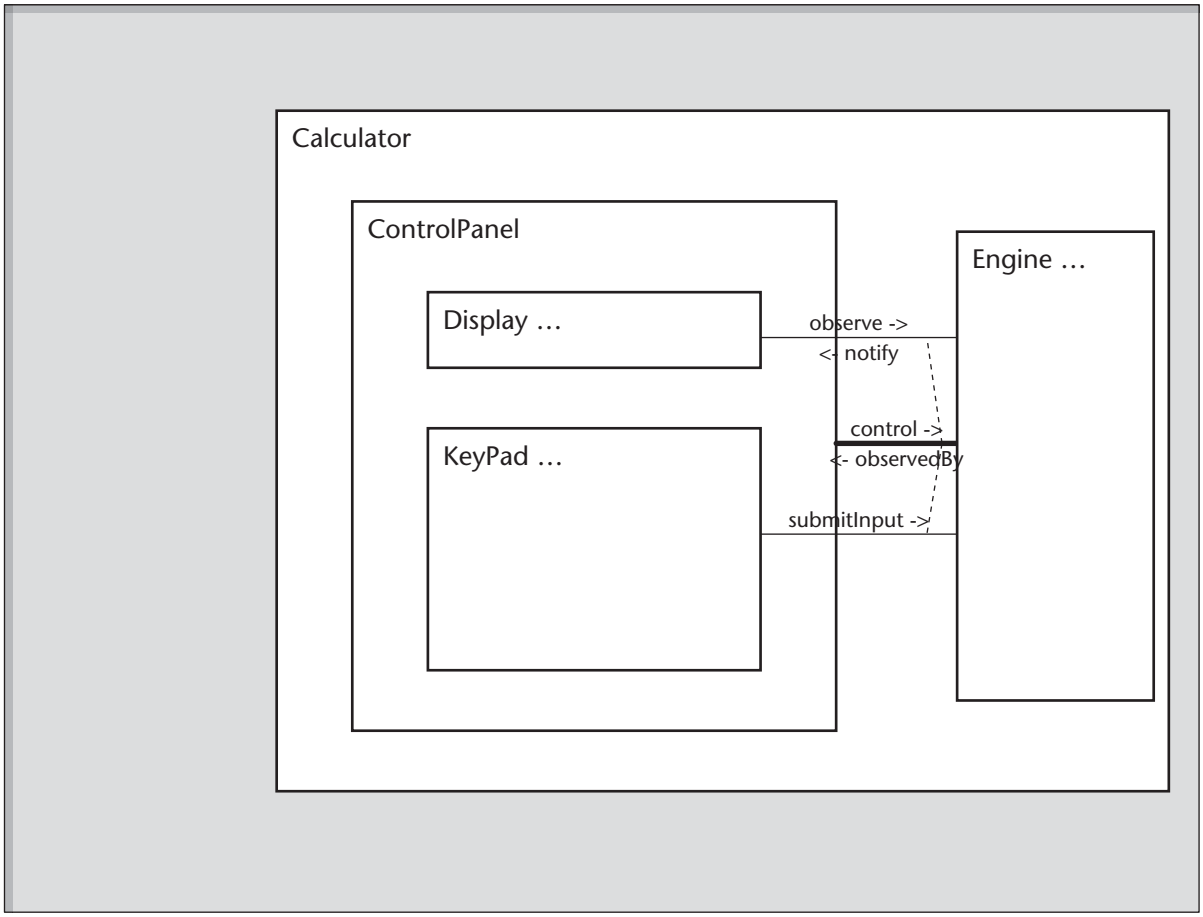


Abbildung 49: *Beispiel für eine Fischaugensicht des Taschenrechners mit Fokus auf dem Objekt ControlPanel, welche aufgrund des in Abb. 50 gezeigten Projektionskörper generiert wurde.*

Visualisierung

Zur Erzeugung von Fischaugensichten werden meist klassische, nicht-lineare Projektionen verwendet. Bisweilen kommen auch hybride Techniken zum Einsatz. Somit wird hier perspektivbasiert betrachtet.

Bei Fischaugensichten wird der Bereich um den aktuellen Fokus detailliert(er) dargestellt, während mit zunehmender Entfernung vom Fokus vergröbert dargestellt wird.

Zur Orientierung steht nur eine Sicht zur Verfügung, diese zeigt allerdings das (lokale) Detail eines Elements im (globalen) Kontext des gesamten Modells an. Die Sicht hat immer Darstel-

lungsebene DE = 0 und verfeinert bezüglich der Darstellungstiefe. Fischaugensichten sind somit kontexterhaltend und es ist unmittelbar ersichtlich, welcher Modellausschnitt gerade dargestellt wird. Unterstellt man, dass In-Situ-Vergrösserung gegeben ist, muss der Benutzer nicht zuerst physikalisch navigieren (vgl. Kapitel 7.2) und/oder verschiedene Sichten integrieren (vgl. Kapitel 7.3), um sich zu orientieren.

Die Art und Weise wie Sichten generiert werden bzw. wann welche Elemente wie dargestellt werden, ist jedoch speziell, da bei Fischaugenkonzepten implizit Annahmen darüber gemacht werden, an welchen Elementen der Betrachter interessiert ist (siehe speziell [Furn86]). Da diese Annahmen bisweilen falsch sind, entstehen mitunter Sichten, die nicht der normalen Benutzererfahrung entsprechen.

Um einige der nachfolgend angeführten Probleme mit Fischaugensichten besser verstehen zu können, ist es notwendig, ein wenig darüber auszuholen, wie die Sichten technisch realisiert sind. Fischaugensichten können technisch unterschiedlich realisiert werden (vgl. [Sch+96]).

- Zum einen können die Elemente eines Modells auf einen Körper oder ein Feld projiziert und somit die Sicht grafisch verzerrt werden. Elemente schrumpfen bzw. werden vergröbert dargestellt, je weiter sie auf dem Projektionskörper vom Fokus entfernt sind, z.B. [Mac+91] [Sar+92] [Keah97].
- Zum anderen können Sichten durch Filterung generiert werden. Hier bestimmt eine explizit formulierte Funktion die aktuelle Relevanz eines Elements. Der Vergleich mit einem Schwellwert entscheidet anschliessend, ob ein Element dargestellt wird oder nicht, z.B. [Furn86].

Beide Ansätze sind im Prinzip ineinander überführbar (vgl. [Leu+94] [Keah97]), entweder durch geeignete Wahl des Projektionskörpers oder durch geeignete Wahl der Relevanzfunktion. Die Überführung trägt hier jedoch wenig zum besseren Problemverständnis bei. Exemplarisch sollen beide Strategien kurz vorgestellt werden.

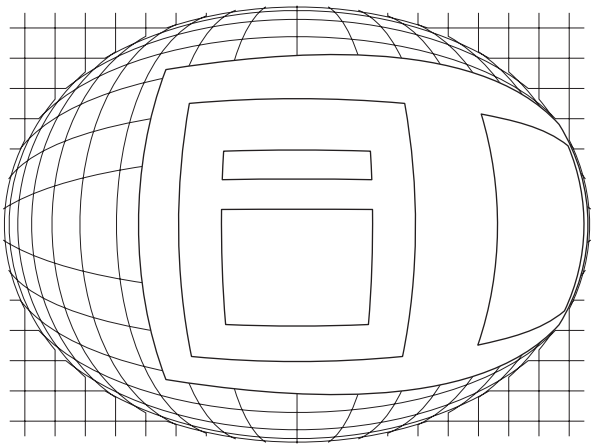


Abbildung 50: *Projektionskörper für die in Abb. 49 gezeigte Fischaugensicht.*
Stilisiert dargestellt ist die Basisstruktur des Taschenrechners. Der Fokus befindet sich in der Mitte des Projektionskörpers, ist also statisch in der Sicht und nicht auf den Elementen verankert. Das Bedienfeld ControlPanel wird durch die Projektion soweit vergrössert, dass die Komponenten Display und KeyPad dargestellt werden können.

Geometrische Projektion

Abb. 49 zeigt ein Beispiel einer Fischaugensicht des Taschenrechnerbeispiels, für deren Generierung der in Abb. 50 gezeigte Projektionskörper verwendet wurde. Die Komponenten eines Objekts werden hier genau dann angezeigt, wenn durch die Projektion genügend Platz für die Darstellung geschaffen werden konnte (vgl. ControlPanel und Engine).

Logische Projektion

Filterungsbasierte Ansätze – wie beispielsweise [Furn86] – benutzen eine Funktion, die sogenannte 'Degree-Of-Interest'-Funktion DOI, um die aktuelle Relevanz eines Elements für den Betrachter zu bestimmen.

> *… this basic strategy uses a 'Degree of Interest' (DOI) function which assigns to each point in the structure, a number telling how interested the user is in seeing that point, given the current task … [Furn86]*

Für Baumstrukturen wird angenommen, dass Punkte, die näher an der Wurzel sind, mehr Relevanz haben, als solche, die weiter entfernt sind. Angezeigt wird ein Element, wenn die durch die DOI-Funktion für ein Element berechnete aktuelle Relevanz einen vorgegebenen Schwellwert s erreicht oder überschreitet (DOI ≥ s). Über einen Ordnungswert o kann Einfluss auf den Schwellwert genommen werden, um die Darstellungstiefe der anzuzeigenden Elemente zu verändern. Bleibt die aktuelle Relevanz unter dem Schwellwert, wird das Element nicht angezeigt.

Die aktuelle Relevanz ist hierbei von zwei Faktoren abhängig: (1) Wie wichtig ein Element a priori ist – die sog. *a priory importance* (kurz API) – und (2) wie weit ein Element vom just fokussierten Element (D) entfernt liegt. Beide Werte werden über entsprechende Funktionen bestimmt. Die aktuelle Relevanz (DOI) eines Elements steigt mit dessen 'a priori Wichtigkeit' und sinkt mit wachsender Distanz vom aktuell fokussierten Element.

DOI(e) = API(e) - D(e, f)

wobei e Element, für welches die aktuelle Relevanz berechnet werden soll
f aktuell fokussiertes Element

Für eine Baumstruktur – wie zum Beispiel die Basisstruktur eines ADORA-Modells – können nach [Furn86] die 'a priori-Wichtigkeit' (API) und die Distanz (D) wie folgt definiert werden:

D(e, f) = d(t, e, f) also als die Pfadlänge zwischen Knoten e und f in der Baumstruktur t
API(e) = d(t, e, root(t)) also als Höhendifferenz zwischen Wurzel und Knoten e

wobei d(t, e, f) für eine Baumstruktur t die Pfadlänge zwischen Knoten e und f liefert und
root(t) für eine Baumstruktur t den Wurzelknoten liefert

Abb. 51 zeigt am bekannten Taschenrechner exemplarisch die Werte für a priori-Wichtigkeit (Abb. 51a) und Distanz (Abb. 51b). Für die Distanzwerte wird hierbei angenommen, dass die Tastatur des Taschenrechners, also das Objekt KeyPad fokussiert ist. Aus der a priori-Wichtigkeit und der Distanz berechnet sich die aktuelle Relevanz eines Elements (Abb. 51c).

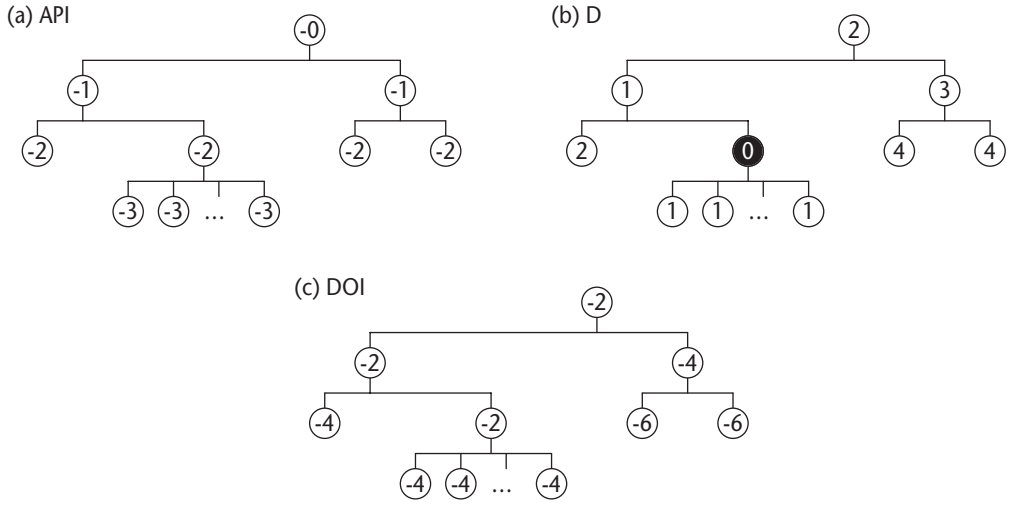


Abbildung 51: *Beispiel für die Werte für a priori Wichtigkeit API, Distanz D und die hieraus berechnete aktuelle Relevanz DOI bei filterbasierten Strategien wie z.B. [Furn86].*
Für die (Baum-)Struktur des Taschenrechners zeigt (a) die Werte für die a priori Wichtigkeit API und (b) die Werte für die Distanz(en) bei fokussiertem KeyPad. (c) zeigt die Werte für die aus API und der jeweiligen Distanz D für jedes Element berechnete aktuelle Relevanz DOI.

Wie bereits erwähnt, wird ein Element genau dann angezeigt, wenn dessen aktuelle Relevanz DOI einen gegebenen Schwellwert s erreicht oder überschreitet ($DOI \geq s$). Der Schwellwert s berechnet sich für eine Baumstruktur t und fokussiertem Element f folgendermassen:

$$s = -(d(t, root(t), f) + 2 \cdot o)$$

wobei o die Ordnung der Fischaugensicht angibt

Mittels des Ordnungswertes o wird für alle dargestellten Elemente die Darstellungstiefe angegeben. Ist die Ordnung $o = 0$, so spricht Furnas [Furn86] von einer Fischaugensicht null-ter Ordnung. Hier werden lediglich die Elemente auf dem Weg von der Wurzel bis hin zum aktuell fokussierten Element angezeigt (siehe Abb. 52d).

Ist die Ordnung $o = 1$, so ist es nach Furnas [Furn86] eine Fischaugensicht erster Ordnung. Die Elemente von der Wurzel bis hin zum aktuell fokussierten Element mit ihren unmittelbaren Nachfolgern werden angezeigt (siehe Abb. 52e).

Sichten mit höheren Ordnungen ($o > 1$) werden entsprechend gebildet.

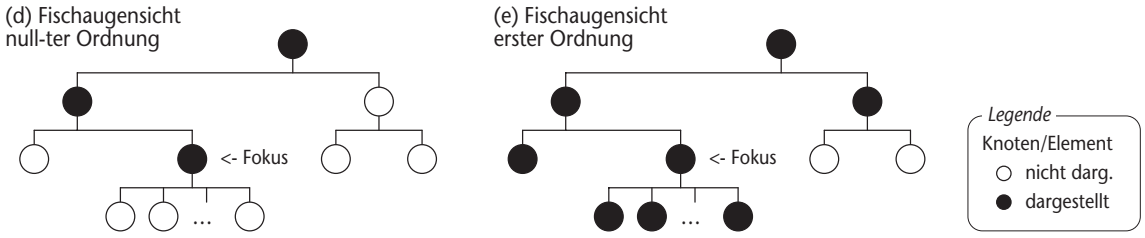


Abbildung 52: *Beispiel für die dargestellten Elemente in einer Fischaugensicht null-ter (d) und erster (e) Ordnung.*

Navigation

Bei gegebener Okklusionsfreiheit und In-Situ-Vergrösserung der unterliegenden Projektionstechnik muss beim ‘klassischen’ Fischaugenkonzept (siehe Kapitel 6.2.2) nicht physikalisch navigiert werden[7].

Logisch navigiert wird beim Fischaugenkonzept durch einen Fisheye Zoom/Selective Zoom. Hierfür wird der Fokus durch Auswahl eines spezifischen Elements oder Bereichs bestimmt. Anhand des Fokus’ können dann die darzustellenden Elemente bestimmt sowie die Sicht entsprechend der zugrundeliegenden nicht-linearen Projektionstechnik generiert werden.

In Hollands [Hol+89] wird die Navigation im Fischaugenkonzept (Bestimmung eines neuen Fokus’, Zoom-In) mit der Navigation mittels Scroll Bars[8] im 1-Fensterkonzept verglichen. Hollands kommt dabei zu dem Schluss, dass die Navigation im Fischaugenkonzept der Navigation mit Scroll Bars im 1-Fensterkonzept überlegen ist, wenn die gesuchten Elemente nicht schon in der aktuellen Sicht dargestellt werden. Die Navigation mittels Scroll Bars ist dann überlegen, wenn das gesuchte Element bereits in der Sicht dargestellt ist. Hier wirkt sich die projektionsbedingte Verzerrung negativ aus.

Zum hier vorgestellten Fischaugenkonzept ist anzumerken, dass nicht-lineare Projektionstechniken mit In-Situ-Vergrösserung auf den ersten Blick zwar sehr attraktiv erscheinen aber grundsätzlich nicht unproblematisch sind. Neben den bereits in Kapitel 6.2.2 angesprochenen Problemen beim Fall ‘Fokus-auf-Fokus’, können die erzeugten Sichten mit zunehmender Darstellungstiefe in Richtung eines Explosive Zoom degenerieren.

Der Kontext wird zwar noch dargestellt, jedoch in einer Form, welche nicht mehr unbedingt zweckdienlich ist, d.h. er (der Kontext) muss sukzessiv soweit verkleinert oder verzerrt werden, dass er irgendwann nicht mehr vernünftig zu erkennen ist. In Abb. 54 auf Seite 89 wird dieser Fall illustriert. Hierzu wird die Sicht aus Abb. 49 auf Seite 83 weiter verfeinert, indem nun die Komponenten des Objekts Keypad dargestellt werden sollen.

Unabhängig davon, ob die verfeinerte Sicht durch eine grafische Verzerrung (siehe Abb. 53) oder durch Filterung erzeugt wird, gelangt man in eine Situation, in welcher früher oder später nicht mehr genügend Platz vorhanden ist, um den Kontext sinnvoll anzuzeigen (siehe Abb. 54a). Verzichtet man auf die Okklusionsfreiheit, um zusätzlich Platz zu gewinnen, ändert dies grundsätzlich nichts an dieser Problematik (siehe Abb. 54b).

Ein weiteres Problem betrifft die Darstellung der Beziehungen, bzw. die Darstellung zweier kooperierender Objekte, wenn diese Objekte nicht Komponenten des gleichen Objekts sind und demnach auf der gleichen Darstellungsebene liegen. Benutzt man Projektionstechniken

---

7 Hierbei wird davon ausgegangen, dass keine Anforderungen bezüglich der minimalen Grösse angezeigter Elemente gestellt werden. Existieren derartige Anforderungen, so kann entweder nicht beliebig tief verfeinert werden, oder es muss grundsätzlich auf In-Situ-Vergrösserung verzichtet werden (vgl. hierzu Kapitel 6.2.3, Hybride Techniken).

8 Konkret ging es in diesem Experiment darum, die optimale Fahrtroute zwischen zwei Stationen in einem U-Bahnplan herauszufinden.

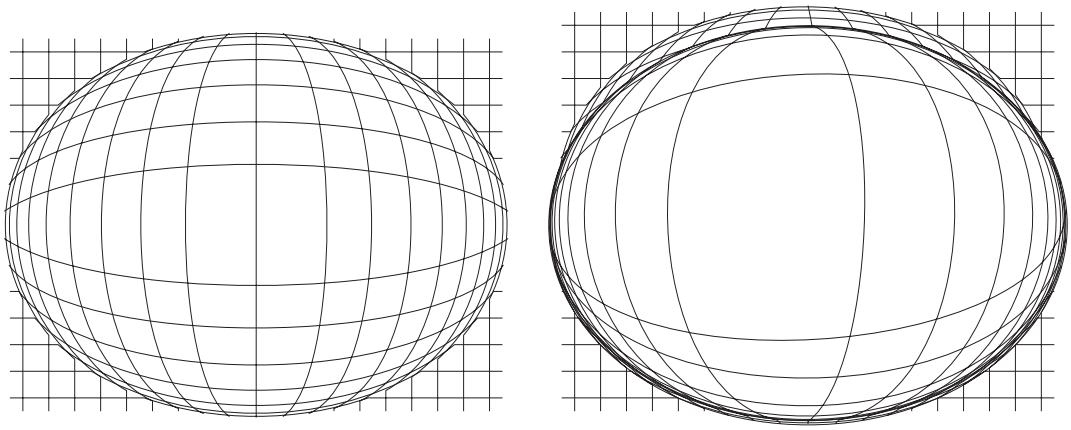


Abbildung 53: *Änderung des Projektionskörpers bei Zoom In.*

mit Einfachfokus, so können u.U. die kooperierenden Objekte und die Beziehungen zu diesen nicht dargestellt werden. Dies ist ein ähnliches Problem wie beim Explosive Zoom der perspektivlosen Konzepten (1-Fensterkonzept, 2-Fensterkonzept; vgl. Kapitel 7.3).

Benutzt man In-Situ-vergrössernde Projektionstechniken, welche Mehrfachfoki erlauben – die hier vorgestellte von Furnas [Furn86] gehört übrigens nicht dazu – so handelt man sich die bereits erwähnten Anomalien ein (siehe Kapitel 6.2.2). Zusätzlich verschärft sich das angesprochene Problem, dass Sichten mit zunehmender Darstellungstiefe in Richtung eines Explosive Zoom degenerieren.

7.4.1 Existierende Fischaugenkonzepte

Es existieren mittlerweile zahlreiche, spezifische Fischaugenkonzepte. Das Prinzip dieser Konzepte ist zwar immer gleich – perspektivbasierte Betrachtung für Sichten mit lokalem Detail und globalem Kontext – jedoch unterscheiden sich die einzelnen Ansätze darin, welche Art von Datenstruktur für welche Art von Anwendung visualisiert werden soll und welche Eigenschaften die unterliegende Projektionstechnik hat (klassisch nicht-linear oder unter Verwendung einer hybriden Technik). Hier eine kurze Auflistung einiger Ansätze.

Kadmon und Shlomi [Kad+78] beschreiben bereits 1978 eine derartige Technik, zur Betrachtung von Landkarten, die sie “Polyfocal Projection” nannten. Mit den ‘Generalized Fisheye Views’ von Furnas [Furn81] [Furn86] wurde die Idee populär. Hier wurden Fischaugenkonzepte zur Filterung von Programm-Code, für biologische Taxonomien sowie für Kalender gezeigt. Kurz darauf folgte das ‘Bifocal Display’ von Robert Spence und Mark Apperley [Spe+82]. Hier wird eine einfache, zweigeteilte Sicht beschrieben mit einem zentralen Bereich mit hoher Vergrösserung und daran angrenzenden Bereichen, wo entsprechend verkleinert wird. Remde, Gomez und Landau [Rem+87] übernahmen die Idee, um ein elektronisches Buch (SUPERBOOK) zu realisieren. Hier waren die Kapitel manuell aufklappbar. In Abhängigkeit davon, wieviele Kapitel ausgewählt sind, werden entsprechend Unterkapitel angezeigt

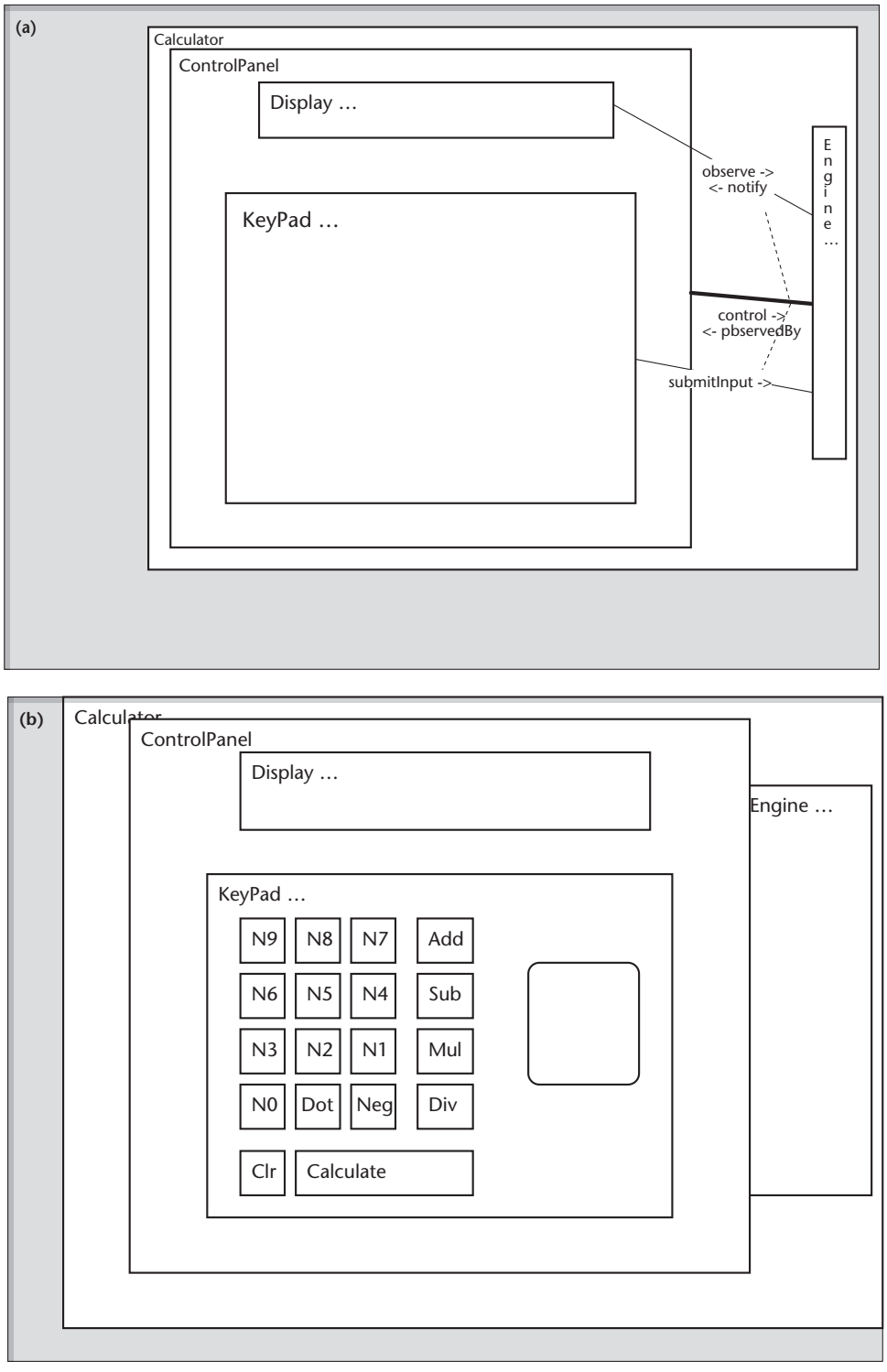


Abbildung 54: *Verfeinerung der Sicht aus Abb. 49 als Beispiel für Anomalien, die bei Fischaugensichten mit In-Situ-Vergrösserung bei hoher Darstellungstiefe auftreten.*
In Abb. 52 ist der Fokus auf dem Objekt ControlPanel. Nun wird KeyPad fokussiert. (a) Der erzeugte Platz reicht aber nicht aus, um die Komponente von KeyPad anzuzeigen. Es müsste nochmals fokussiert werden, was eine noch stärkere Verzerrung der umliegenden Objekte Display, Engine, etc, zur Folge hätte. (b) Verzichtet man auf die Okklusionsfreiheit um zusätzlich Platz zu schaffen, so geht der Kontext verloren.

oder eben verborgen. Eine Evaluierung dieses Konzepts zeigte, dass Studenten über dieses System Suchanfragen besser beantworten konnten, als über herkömmlichen Text.

Sarkar und Brown [Sar+92] verwendeten das Fischaugenkonzept zur Visualisierung von Graphen. Knoten werden entsprechend ihrer Position und Grösse verzerrt, um so deren Relevanz wiederzugeben. Diese Technik wurde später über eine sog. 'Rubber Sheet' Metapher erweitert [Sar+93], so dass Mehrfachfoki möglich waren. Die 'Rubber Sheet' Metapher wurde wiederum von Kaltenbach, Robiland und Frasson aufgegriffen [Kal+91], um in Hypertext-Systemen die Aufteilung des Bildschirmplatzes zu regeln.

Noik [Noik93] kombinierte Fischaugensichten und hierarchische Graphen zu einem Hypertext System. Knoten repräsentieren Dokumente. Diese sind in einer entsprechenden Hierarchie organisiert. Wird ein Knoten dargestellt, geben Grösse und Detaillierung Auskunft über dessen aktuelle Relevanz. Das IDG-System [Fein88] visualisiert gerichtete, azyklische Graphen (DAGs) und erlaubt es, mehrere Bereiche mit unterschiedlichem Detaillierungsgrad in verschiedenen Fenstern darzustellen.

Die 'Tree Maps' von Johnson und Shneiderman [Joh+91] sind ein In-Situ-Ansatz zur Visualisierung von Baumstrukturen. Ebenfalls zur Visualisierung von Baumstrukturen dient der 'FlexView' Ansatz von Shaffer und Greenberg [Sc+93b]. Jedem Knoten können Attribute zugewiesen werden. Auf Basis dieser Attribute können Anfragen gestellt werden und die Visualisierung passt die Darstellung (Farbe, Grösse, etc.) der Teil-/Unterbäume entsprechend den gefundenen Übereinstimmungen an. Zur Visualisierung grosser Tabellen dient die von Rao und Card [Rao+94] entwickelte 'TableLens'. Der von Macinlay, Robertson und Card [Mac+91] entwickelte 'Perspective Wall' hat seine Wurzeln im o.g. Ansatz von Spence und Apperley [Spe+82] und erzeugt eine perspektivische Sicht mit zwei Fluchtpunkten auf eine flache Struktur. Darauf folgte die sog. DocLens [Rob+93], welche einige der Mängel des 'Perspective Wall' beseitigte. Ebenfalls von Macinlay, Robertson und Card [Rob+91] stammen die 'Cone Trees' und 'Cam Trees' zur dreidimensionalen und animierten Betrachtung von Baumstrukturen. Die sog. 'Fractal Views' [Koik95] versuchen durch Filterung die Gesamtmenge an dargestellter Information konstant zu halten.

8 ZUSAMMENFASSUNG & DISKUSSION

8.1 Diskussion der vorgestellten Konzepte

Von den drei vorgestellten Visualisierungskonzepten (siehe Kapitel 7.2 - 7.4) leiden 1-Fensterkonzepte mit Explosive Zoom stark unter dem auftretenden Kontextverlust. Die Orientierungsmöglichkeiten sind bei diesem Konzept für den Betrachter entsprechend erschwert. Wird zur logischen Navigation anstatt des Explosive Zoom ein Full Zoom verwendet, so wird zwar der Kontext beibehalten, aber um den Preis eines immensen Platzbedarfs der Sicht und mit den entsprechenden Konsequenzen bezüglich des Aufwands für die physikalische Navigation (vgl. hierzu Kapitel 6.2.1). Ferner besteht durch den Full Zoom zusätzlich die Gefahr der Detailüberflutung. Angesichts der hohen Integrationsdichte von Modellelementen bei ADORA-Modellen lässt dies die Anwendung einer Ausprägung des 1-Fensterkonzepts eher unpraktikabel erscheinen. Insbesondere zusammenhängende Sichten für grosse Modelle mit hoher Integrationsdichte lassen sich so nicht mehr vernünftig überschauen.

Unabhängig von der verwendeten Zoom Strategie bieten 1-Fensterkonzepte (im eigentlichen Sinn, also ohne zusätzliche Sichten) im direkten Vergleich mit 2-Fensterkonzepten mehr Platz für die Bereitstellung der eigentlichen Arbeits- bzw. Hauptsicht. Das Übersichtsfenster im 2-Fensterkonzept bietet eine gewisse, wenn auch limitierte Kontextinformation und ermöglicht so eine verbesserte logische Navigation. In [Bea+90] wird die Hypothese bestätigt, dass die logische (Roam-)Navigation durch ein Übersichtsfenster schneller ist als die Navigation über Scroll Bars. Da beim 2-Fensterkonzept dem Betrachter die Integrationsarbeit für die beiden Sichten (Übersicht und Arbeitssicht) überlassen bleibt (vgl. [Leun89]), stellt dieses Konzept im Vergleich zum 1-Fensterkonzept erhöhte kognitive Anforderungen. Eine Sicht, die als Übersicht fungiert, hat den Anspruch, möglichst das ganze Modell zu zeigen. Sie ist i.d.R. nicht weiter verkleinerbar und sie sollte keine oder nur eine geringe horizontale Verkürzung aufweisen (vgl. Kapitel 7.3). Bei grossen Modellen verschärft dies die Integrationsproblematik zunehmend, da nun der Zusammenhang zwischen einer stark vergröberten Übersicht und der detaillierten Arbeitssicht nicht mehr ohne weiteres ersichtlich ist. Sinnvollerweise kommt bei 2-Fensterkonzepten vornehmlich Explosive Zoom zum Einsatz. Dies führt dazu, dass im Vergleich zu den anderen beiden Konzepten – 1-Fenster und Fischaugenkonzept – das Problem der Darstellung kooperierender Objekte (insbesondere die Beziehungen zu diesen, vgl. Kapitel 7.3) beim 2-Fensterkonzept am Ausgeprägtesten ist.

Fischaugenkonzepte bieten die Möglichkeit, Detail und Kontext gleichzeitig und in einer einzigen (Arbeits-)Sicht darzustellen. In dieser Hinsicht sind sie den hier vorgestellten 1- und 2-Fensterkonzepten klar überlegen. Lokales Detail und globaler Kontext ist speziell im Hinblick auf die Vermeidung von kognitivem Ballast eine Kernforderung für ein Visualisierungskonzept für ADORA-Modelle. Insbesondere die guten Orientierungsmöglichkeiten im Fischaugenkonzept tragen zu geringem kognitiven Ballast bei und machen dieses Konzept zu einem Favoriten für die Visualisierung von ADORA-Modellen.

Von den in Kapitel 7.4.1 vorgestellten Ansätzen wären auf den ersten Blick [Furn86] [Fein88] [Sar+92] [Sar+93] [Noik93] und im besonderen [Sc+93a] grundsätzlich für die Visualisierung von ADORA-Modellen anwendbar. Alle diese Ansätze sind jedoch entweder Techniken mit In-Situ-Vergrösserung und/oder mit einem Einfachfokus.

[Sar+92] [Sar+93] und (leider auch) [Sc+93a] sind Ansätze mit In-Situ-Vergrösserung. Bei Verwendung einer nicht-linearen Projektionstechnik mit In-Situ-Vergrösserung könnte rein theoretisch auf physikalische Navigation vollständig verzichtet werden (vgl. Kapitel 7.4 und auch [Kaen96]). Die durch die Verwendung von Techniken mit In-Situ-Vergrösserung auftretenden Probleme und Anomalien (vgl. Kapitel 6.2.2, Kapitel 7.4, [Leun89] [Sch+96]) lassen dies jedoch wenig praktikabel erscheinen und wiegen den Vorteil der wegfallenden physikalischen Navigation nicht auf. Um die erwähnten Anomalien zu vermeiden (vgl. Kapitel 6.2.3), ist es zweckmässig, auf eine In-Situ-Vergrösserung zu verzichten und eine Projektionstechnik zu verwenden, welche bei der Verfeinerung einer Sicht nur soviel zusätzliches Detail erzeugt, dass sich der physikalische Navigationsaufwand in Grenzen hält.

[Furn86] [Fein88] [Noik93] sind ebenfalls Ansätze mit In-Situ-Vergrösserung, verfügen aber lediglich über einen Einfachfokus. Das vom Explosive Zoom her bekannte Problem der nicht dargestellten kooperierenden Objekte (vgl. S. 80), lässt sich jedoch am besten durch Verwendung von nicht-linearen Projektionstechniken entschärfen, die Mehrfachfoki erlauben. Bereits aus diesem Grund ist die Verwendung einer dieser Ansätze unvorteilhaft. Zusätzlich sollte von der In-Situ-Vergrösserung abgesehen werden, um die bekannten Anomalien (s.o.) bei der Darstellung vermeiden zu können.

8.2 Konzepte in existierenden Werkzeugen

Um den aktuellen Stand umgesetzter Visualisierungskonzepte in Analyse und Entwurfswerkzeugen zu untersuchen, wurden in [Be+98a] die folgenden Werkzeuge betrachtet: KOGGE Tool BONSAI [Koe+96], SOM - Tool [Fer+94], Macrotec [Kel+95], EGS1 [Gas+95], SystemSpecs [SysS96], Statemate [Stat96], MicroSaint [Micr96], Arena [Aren96], ObjecTime Toolset [ObjT96], Rational Rose [Rose96], Software through Pictures - StP [StP96], Objectory [Objy96], Innovator CASE Workbench [Inno96].

Die meisten Werkzeuge, welche auf flachen oder quasi flachen Modellen operieren, unterstützen ausschliesslich Scrolling und bisweilen Vergrösserungs-/Verkleinerungsmechanismen (Scaling, siehe Kapitel 6.3.2). Einige Werkzeuge setzen ein 2-Fensterkonzept um und verfügen über ein Übersichtsfenster zur besseren Orientierung. Roam Navigation im Übersichtsfenster wurde bislang nicht umgesetzt. Ebensowenig kommen nicht-lineare bzw. hybride Projetionstechniken wie beispielsweise [Mac+91] oder [Sar+92] zum Einsatz, die sich speziell für flache Strukturen eignen würden.

Die Werkzeuge, die auf hierarchischen Modellen operieren, stellen normalerweise einen einzelnen Knoten mit seinen direkten Nachfolgern in einer Sicht dar. Nichtlineare Techniken, welche sich für Netzwerke bzw. Hierarchien eignen würden, wie beispielsweise [Furn86]

[Sar+92] [Sc+93a] [Koik95], werden nicht implementiert – auch nicht ansatzweise. Ein paar wenige Werkzeuge wie z.B. Statemate sind in der Lage, alle Knoten in einer Sicht darzustellen.

Fast alle Werkzeuge, die auf hierarchischen Modellen arbeiten, stellen lediglich Explosive Zoom sowie manchmal ein Übersichtsfenster mit Roam-Bar (siehe Kapitel 6.3.2 sowie 7.3) als logische Navigationsart bereit und arbeiten somit nicht kontexterhaltend.

Globaler Kontext und lokales Detail in einer Sicht ist bei sehr wenigen der untersuchten Werkzeuge realisiert, wie z.B. [Kel+95]. Da keine 'richtigen' Projektionstechniken zum Einsatz kommen, gehen Verfeinerungen oder Vergröberung mit einem Vollverlust der Positionierung der Elemente einher, so dass das gewünschte Layout stets manuell wieder hergestellt werden muss. Volle Flexibilität bei Scrolling und Zooming wird generell nicht angeboten.

Zusammengefasst lässt sich hier sagen, dass die in den existierenden Analyse- und Entwurfswerkzeugen verwendeten Visualisierungskonzepte dem Stand der Technik um gut zehn Jahre hinterherhinken[9]. 2-Fensterkonzepte werden mittlerweile vermehrt in Analyse- und Entwurfswerkzeugen implementiert. Fischaugenkonzepte bzw. Konzepte, deren Sichten lokales Detail und globalen Kontext integrieren, wurden in den betrachteten Analyse- und Entwurfswerkzeugen bislang nicht umgesetzt.

Auf die Frage, warum Fischaugensichten ausser in Forschungsprototypen quasi nicht umgesetzt sind, können folgende Gründe angeführt bzw. vermutet werden:

Fischaugensichten sind kein allzu bekanntes Konzept, d.h. sie sind bei weitem nicht so bekannt wie die konventionellen 1- oder 2-Fensterkonzepte. Ausserhalb der 'HCI-Community' sind kaum Veröffentlichungen über Fischaugensichten zu finden. Auch gibt es keine Transferarbeiten, die versuchen, dieses Konzept speziell für die Visualisierung von Spezifikationsmodellen anzuwenden. Dieses Wissen müsste vor einer eventuellen Realisierung erst aufgebaut werden.

Fischaugensichten sind aufwendig zu implementieren. Zum einen sind derartige Visualisierungen schon an sich aufwendig. Der Kontext muss bei der Erzeugung von Sichten mitberücksichtigt werden und inbesondere hierfür werden höhere algorithmische Anforderungen gestellt als bei konventionellen Visualisierungen. Die Schwierigkeiten liegen oft im Detail und müssen vor der eigentlichen Realisierung geklärt werden (vgl. Kapitel 10, Algorithmische Realisierung). Dies erfordert lange Planungsphasen bzw. Vorstudien. Diese wiederum erschweren ein inkrementelles Vorgehen. Zum anderen kann bei der Realisierung nicht oder nur sehr eingeschränkt auf vorhandene Bibliotheken zurückgegriffen werden.

Fischaugensichten sind kein 'Silver Bullet' und nicht jedes Modell ist für eine perspektivbasierte Betrachtungsweise geeignet. Fischaugensichten bringen dann den meisten Nutzen, wenn sie zur Visualisierung von Modellen benutzt werden, die explizit Information darüber enthalten, was unter welchem Blickwinkel als Vergröberung bzw. Abstraktion von was angesehen werden kann. Der Kontext kann dann abstrahiert dargestellt werden und es können echte abs-

[9] Wahrscheinlich sind es eher 20 Jahre. Dem interessierten Leser seien hierzu die ersten beiden Seiten von [Furn81] empfohlen.

trakte Sichten modellgetrieben erstellt werden (vgl. Kapitel 9.2.2, Visualisierung). Nicht jedes Modell verfügt über diese Informationen – insbesondere heutige objektorientierte Spezifikationssprachen weisen diesbezüglich deutliche Mängel auf. Werden Fischaugensichten zur Visualisierung solcher Modelle benutzt, bringen sie nicht mehr den selben Nutzen, da bei der Generierung einer Sicht genau die Informationen darüber fehlen, ‘was unter welchem Blickwinkel als Vergröberung bzw. Abstraktion von was angesehen werden kann‘. Es können weiterhin Sichten erzeugt werden, die lokales Detail und globalen Kontext zusammen darstellen. Diese Sichten entlasten den Betrachter zwar immer noch von (möglicherweise) überflüssigem Detail, stellen aber – im Sinne einer echte Abstraktion – keine abstrakte Sicht auf ein Modell mehr dar. Behilft man sich mit entsprechender Heuristik, wird die Realisierung (noch) aufwendiger und im schlimmsten Fall – dann wenn die Heuristik versagt – werden Abstraktionen vorgetäuscht, die keine sind.

Visualisierung von ADORA-Modellen

In diesem Teil wird das eigentliche Konzept zur Visualisierung von ADORA-Modellen beschrieben. ADORA-Modelle weisen zwei Besonderheiten auf: Zum einen sind sie durch die entsprechenden Abstraktionsmechanismen hierarchisch gegliedert und zum anderen sind es integrierte Aspektmodelle. Wie diesen Eigenschaften bei der Visualisierung Rechnung getragen werden kann, ist Gegenstand dieses Teils der Arbeit.

Kapitel 9 beschäftigt sich mit den Anforderungen an ein Visualisierungskonzept für ADORA-Modelle. Dies erfolgt mitunter rückblickend auf die in Kapitel 7.1 vorgestellten Konzepte.

Wie das zuvor eingeführte Visualisierungskonzept realisiert werden kann, wird in Kapitel 10 erläutert. Hier wird schwerpunktmässig auf die Algorithmik der im Rahmen dieser Arbeit entwickelten Projektionstechnik eingegangen, die für die Generierung von Sichten zur Anwendung kommt.

Kapitel 11 zeigt die Anwendung des Konzepts für ADORA. In Kapitel 12 findet sich dann eine Zusammenfassung sowie eine abschliessende Diskussion des zuvor vorgestellten Visualisierungskonzepts für ADORA-Modelle.

9 Herleitung des Visualisierungskonzepts für ADORA-Modelle

Ein gutes Visualisierungskonzept ist wesentlich, sowohl für die Verständlichkeit, wie auch für die Einfachheit der Anwendung von grafischen Modellen. Die Untersuchung bestehender Konzepte (siehe Kapitel 7.2 - 7.4) und der in Analyse- und Entwurfswerkzeugen (siehe Kapitel 8.2) tatsächlich umgesetzten Konzepte, hat gezeigt, dass zwar einige Konzepte prinzipiell anwendbar wären, keines jedoch uneingeschränkt geeignet ist (siehe Kapitel 8.1). In diesem Kapitel soll nun näher darauf eingegangen werden, wie ein Visualisierungskonzept für ADORA-Modelle beschaffen sein sollte und wie es tatsächlich umgesetzt wurde. Neben den mittlerweile eher offensichtlichen Forderungen nach kontexterhaltenden also orientierungsfreundlichen Sichten und geringem kognitiven Ballast werden hier neu noch zwei Aspekte angeschnitten. Diese Aspekte sind *Primär-* und *Sekundärnotation*.

9.1 Anforderungen

Evaluierungen von existierenden perspektivbasierten Strategien unter Verwendung von nichtlinearen Projektionstechniken untermauern die Hypothese, dass diese Art der Betrachtung perspektivlosen Strategien mit linearen Projektionstechniken insbesondere dann überlegen ist, wenn es sich um die Visualisierung grosser, integrierter und/oder hierarchischer Strukturen handelt (vgl. [Hol+89] [Chi+91] [Sc+93a] [Sch+96] [Kaen96] und auch Kapitel 6 und 7.1). Die Überlegenheit perspektivbasierter Strategien in diesem Bereich ist darauf zurückzuführen, dass hiermit Sichten bereitgestellt werden können, die Detail und Kontext integriert und ausgewogen darstellen. Derartige Sichten vereinfachen wiederum die Orientierung für den Benutzer. Vereinfachte Orientierung erleichtert das Modellverständnis und trägt damit zur Verringerung des kognitiven Ballasts bei.

Das 1-Fensterkonzept hat hier eine eher magere Bilanz. In perspektivlosen Mehrsicht- oder Mehrfensterkonzepten (vgl. Kapitel 7.2, 7.3) sind zwar ebenfalls erweiterte Orientierungsmöglichkeiten vorhanden, jedoch erhöht jede zusätzliche Sicht die kognitiven Anforderungen an den Betrachter, da dieser für die Navigation zusätzliche Integrationsarbeit leisten muss (vgl. [Furn81] oder [Leun89]). Dieser zusätzliche Integrationsaufwand, wie auch die Ressource, die eine Sicht benötigt – sprich der Platz auf dem Ausgabegerät – muss durch den Nutzen, den die zusätzliche Sicht bringt, gerechtfertigt sein. Die Überlegenheit des 2-Fensterkonzepts gegenüber dem 1-Fensterkonzept (vgl. [Bea+90]) ist hauptsächlich durch einen ökonomischeren Umgang mit der Ressource ‘Anzeigegerät’ zu erklären, so dass trotz zusätzlich zu leistender Integrationsarbeit beim 2-Fensterkonzept durch die permanent vorhandene Übersicht weniger physikalische Navigation in der Arbeitssicht anfällt.

Durch geeignete Wahl der Basisstrategie – perspektivlos vs. perspektivbehaftet – können zusätzliche (Über-)Sichten überflüssig werden (vgl. 2-Fensterkonzept und Fischaugenkonzept), so dass der Integrationsaufwand weitgehend wegfällt. Neben der Forderung nach loka-

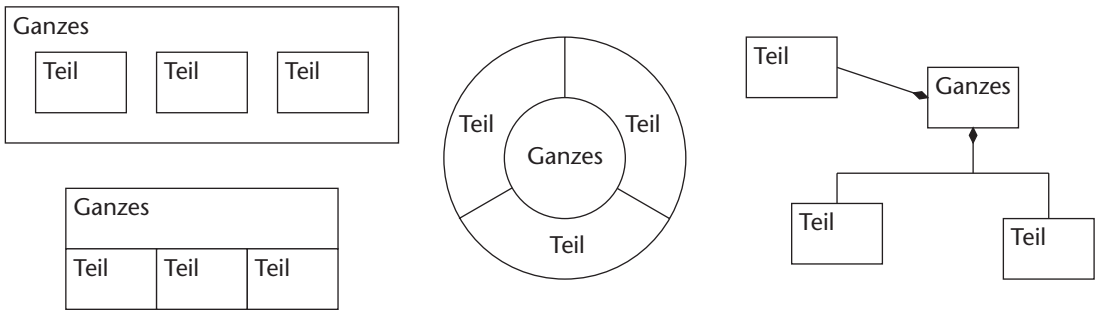


Abbildung 55: *Beispiel für verschiedene Möglichkeiten, wie eine Teil-Ganzes-Beziehung visualisiert werden könnte. Eine Deutung oder Wertung sei dem Betrachter überlassen.*

lem Detail und globalem Kontext sollte generell die Anzahl an gleichzeitig bereitgestellten oder bereitzustellenden Sichten möglichst gering sein. Die Anzahl an grundsätzlich bereitstellbaren Variationen von Darstellungsebene und Darstellungstiefe in einer Sicht sollte möglichst hoch sein.

Je weniger Sichten zur Verfügung stehen oder je mehr Aufwand betrieben werden muss, um eine Modellsicht zu erzeugen, welche eine bestimmte Orientierung optimal ermöglicht, desto flexibler und vielfältiger müssen allerdings die Möglichkeiten – sprich Navigationsinstrumente – sein, um diejenigen Elemente eines Modells zu bestimmen, welche dann schliesslich in dieser Sicht dargestellt werden. Mit jedem zusätzlichen Navigationsinstrument erhöht sich jedoch wiederum der kognitive Ballast.

9.1.1 Primär- und Sekundärnotation

Ein dritter wesentlicher Punkt ist das Modell, welches visualisiert wird. Die Visualisierung muss der Semantik des Modells angemessen sein. In erster Linie heisst dies, es müssen passende Zeichen zur Repräsentation der Modellelemente gefunden werden; d.h. Zeichen, die der Bedeutung der verschiedenen Modellelemente angemessen sind. In [Petr95] wird dies als sog. *Primärnotation* bezeichnet. Je nach Kontext, Betrachter, kulturellem Hintergrund etc. kann die Deutung der Zeichen unterschiedlich sein und so das Verständnis eines Modells entweder erleichtern oder erschweren (vgl. Abb. 55). Die Hauptschwierigkeit besteht darin, Zeichen zu finden, die von möglichst vielen potentiellen Betrachtern gleich gedeutet werden. Eine brauchbare Primärnotation erlaubt eine möglichst direkte Deutung der Zeichen.

Anmerkung: Für ADORA ist die Primärnotation in Form der konkreten Syntax durch die Sprachdefinition bereits nahezu vollständig vorgegeben (siehe [Joos99] oder Kapitel 5) und wird daher hier nicht weiter diskutiert.

Neben diesen eher formalen Aspekten der Beschaffenheit der Primärnotation kommt noch ein weiterer wichtiger Punkt hinzu: Viel von dem, was zu einer guten Verständlichkeit grafischer Repräsentationen beiträgt, ist nicht Teil der Primärnotation, sondern Teil einer Art ‘zweiter Notation’ der sog. *Sekundärnotation*. Diese betrifft Aspekte wie Positionierung und Proportio-

nierung der Elemente, etc.; also kurz das Layout. In [Petr95] wird deutlich auf die tragende Rolle der Sekundärnotation hingewiesen.

> *… that much of what contributed to the comprehensibility of a graphical representation isn't part of the formal programming notation but a 'secondary notation' of layout, typographic cues, and graphical enhancements that is subject to individual skill … graphics (alone) do not guarantee clarity: 'good graphics relies on secondary notation' [Petr95]*

Nicht alleine eine 'gute' grafische Primärnotation garantiert die Verständlichkeit, sondern auch eine gute Nutzung der Sekundärnotation. Dies rührt daher, dass das Lesen grafischer Modelle eine erlernte Fähigkeit ist (siehe [Pet+93]).

Die Struktur einer bestimmten Darstellung, die Beziehungen, die Relevanz bestimmter Elemente, etc. sind nicht per se offensichtlich, sondern werden vom Betrachter in einem hohen Masse durch die Positionierung und Proportionierung der Elemente gedeutet (siehe [Petr95]). Das Erwerben dieser Fähigkeit ist ein Lernprozess. Der Grad des Fortschritts dieses Lernprozesses oder kurz die Erfahrung des Betrachters hat Einfluss darauf, wieviel Nutzen aus der Sekundärnotation gezogen wird. Je mehr Erfahrung ein Betrachter hat, desto mehr wird er nach bestimmten – über die Sekundärnotation vermittelten – Hinweisen suchen. Anfänger scheitern nicht allein deswegen, weil sie die Primärnotation nicht gut genug kennen, sondern weil sie noch nicht genügend Erfahrung in der Deutung der Sekundärnotation gesammelt haben. Experten haben dann erhebliche Schwierigkeiten, wenn sie über die Sekundärnotation irregeführt werden, d.h. sie glauben dann ein bestimmtes Muster zu erkennen, doch dieses entpuppt sich nicht als das, was sich normalerweise hinter einem solchem Muster verbirgt (vgl. [Chi+88] [Petr95]).

Lässt die Art und Weise wie Sichten generiert werden (vgl. Abb. 56 und 57) keine oder nur sehr eingeschränkt Sekundärnotationen zu (z.B. [Joh+91] oder [Rob+91]), so erschwert dies empfindlich das Verständnis der unterliegenden Modelle, wobei ein Experte, also jemand, der mit der Zeit gelernt hat, die Sekundärnotation zu deuten, diesen Mangel schwerwiegender empfindet als ein unerfahrener Betrachter.

Die zur Sichtengenerierung verwendete Projektionstechnik sollte daher sowohl Sekundärnotationen zulassen, wie auch eine vorhandene Sekundärnotation so weit wie möglich erhalten. Während dies bei linearen Projektionstechniken meist a priori gegeben ist, stellt dies an nichtlineare Projektionstechniken besondere Anforderungen. Insbesondere Techniken mit In-Situ-Vergrösserung sind hier problematisch (siehe Abb. 54).

Anmerkung: Die fehlende Möglichkeit zur Anlage und zum Erhalt von Sekundärnotationen ist einer der Hauptgründe, weshalb hier Ansätze generell nicht betrachtet werden – wie beispielsweise automatische Graphlayouter – in welchen der Benutzer keine oder nur sehr wenig Möglichkeiten hat, auf die Anordnung der Elemente Einfluss zu nehmen (vgl. Kapitel 7.1).

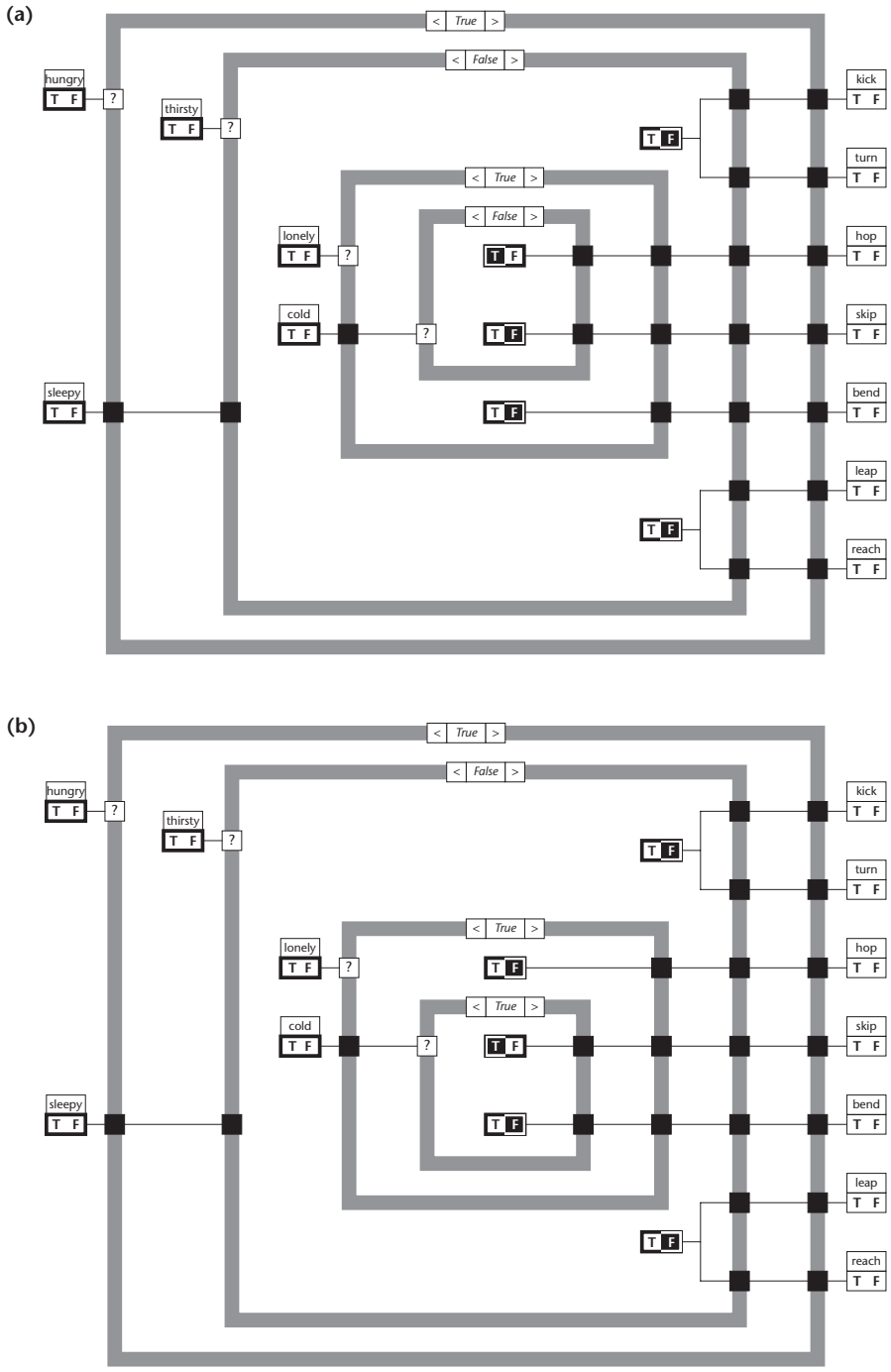


Abbildung 56: *Beispiel aus [Petr95] für eine interaktive Notation.*
Die Notation ist ausgelegt für eine entsprechende Werkzeugunterstützung und erlaubt keine Sekundärnotation. Steht kein Werkzeug zur Verfügung, muss für jede mögliche Belegung manuell neu gezeichnet werden. Zur Erklärung: Mausklicks in die True/False-Buttons bewirken einen Wechsel zwischen den entsprechenden Zweigen einer Bedingung. (a) zeigt eine Sicht mit hungry = true, thirsty = false, lonely = true, cold = false. Der Wert von sleepy ist dort irrelevant; bei anderen Belegungen kann er relevant sein. Die Ausgabe wird beschrieben, indem jeder mögliche Ausgang mit einem entsprechenden Wahrheitswert belegt wird. (b) zeigt eine Sicht mit hungry = true, thirsty = false, lonely = true, cold = true. Der innerste Rahmen wird beim Umschalten des Wahrheitswertes verschoben. Solange cold = true ist, ist hop nicht möglich, sondern nur skip oder bend.

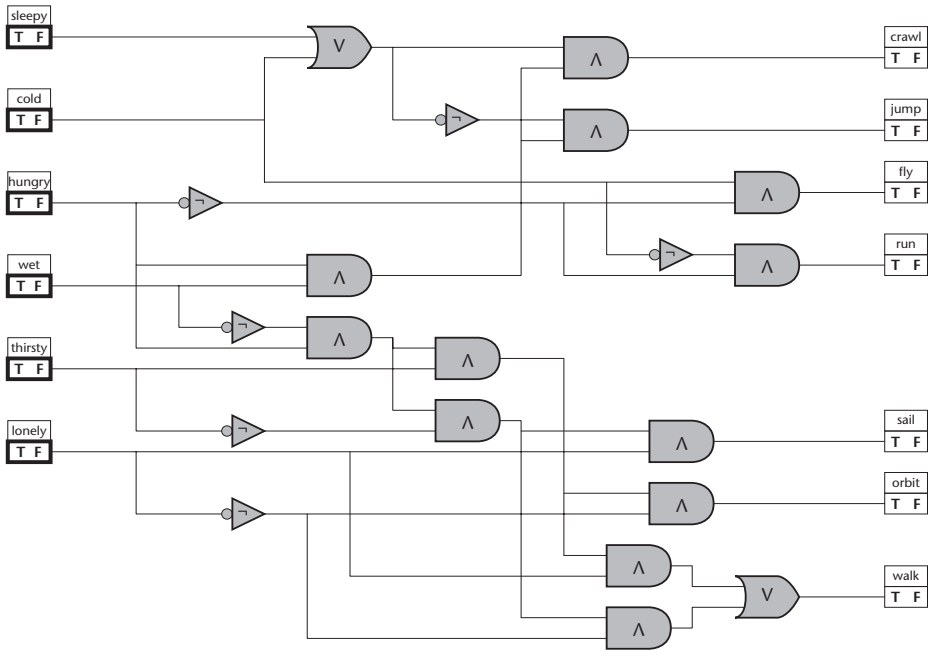


Abbildung 57: *Beispiel aus [Petr95] für ein nicht-interaktives Pendant zu Abb. 56. Der Benutzer bestimmt die Anordnung der Elemente und dieses Layout kann unverändert bleiben, unabhängig von der Belegung der Werte.*

Zusammenfassung

Zusammengefasst sollte in einem guten Konzept (vgl. auch [Be+98a] [Be+98b]):

G1 Die Orientierung im unterliegenden Modell dadurch unterstützt werden, dass in einer Sicht soviel Detail wie jeweils notwendig angezeigt werden kann, ohne dass der globale Kontext der angezeigten Details verloren geht.

G2 Der kognitiven Ballast für die Navigation möglichst minimal sein. Neben den durch eine Sicht gegebenen Orientierungsmöglichkeiten sollten die angebotenen Navigationsinstrumente durch den Benutzer einfach und direkt interpretierbar sein.

G3 Der Semantik der unterliegenden Notation Rechnung getragen werden, so dass die Ausdrucksstärke und die Verständlichkeit des Modells durch die Primärnotation adäquat zur Geltung kommt.

G4 Eine Sekundärnotation möglich sein und im Zuge von Navigationsaktivitäten soweit wie möglich erhalten werden.

Offensichtlich sind in dieser Aufzählung G1 und G4 nicht unabhängig, sondern beeinflussen sich gegenseitig. Wird eine kontexterhaltende Sicht durch Verfeinerung oder Vergröberung aus oder auf Basis einer bestehenden Sicht generiert, so wird die Proportionierung und Positionierung der Elemente verändert. Dies hat wiederum Einfluss auf die mittels der Sekundärnotation vermittelten Informationen.

Navigations-Szenarien

Unabhängig von der Rolle, die ein Benutzer konkret innehat, d.h. unabhängig davon, ob er Modellierer oder Modellnutzer ist, gibt es einige typische Interaktionen, die immer wieder durchgeführt werden. Kenntnis derselben ist notwendig, wenn es darum geht – unter Berücksichtigung der o.g. allgemeinen Qualitäten – Kernanforderungen an ein Visualisierungskonzept abzuleiten. Diese 'typischen Interaktionen' werden hier in Form sog. Navigations-Szenarien festgehalten.

Als typische Interaktionen des Benutzers mit dem System wurden die Szenarien S1 - S3 unterstellt[1], wobei die Szenarien S1, S2 die logische Navigation und S3 die physikalische Navigation betreffen.

S1 Der Benutzer möchte eines (S1.1), mehrere (S1.2) oder alle (S1.3) Elemente/ Objekte detaillierter darstellen. Hierzu wird logisch navigiert und die aktuelle Sicht verfeinert. Durch die Verfeinerung werden die entsprechenden Kompositionen, Beziehungen, etc. sichtbar.

S2 Der Benutzer möchte eines (S2.1), mehrere (S2.2) oder alle (S2.3) Elemente/ Objekte vergröbert darstellen, also eine Abstraktion der aktuellen Sicht erhalten. Hierzu wird logisch navigiert und die aktuelle Sicht vergröbert. Durch die Vergröberung wird von der internen Struktur der entsprechenden Kompositionen inkl. Beziehungen, etc. abstrahiert.

S3 Der Benutzer möchte den in der aktuellen Sicht dargestellten Ausschnitt verändern, ohne dass die logische Struktur der Sicht verändert wird. Hierzu wird physikalisch navigiert und die aktuelle Sicht entsprechend vergrössert (nicht vergröbert) oder der sichtbare Ausschnitt wird verschoben.

Kernanforderungen

Nachfolgend sind Kernanforderungen R1 - R7 an ein Visualisierungskonzept für ADORA-Modelle angeführt. Aus der Kombination der allgemeinen Qualitäten G1 - G4 und der Navigations-Szenarien S1 - S3 lassen sich die Anforderungen R1 - R6 an ein Visualisierungskonzept für ADORA-Modelle ableiten.

Zusätzlich kommt noch die Anforderung R7 hinzu, welche sich speziell dadurch ergibt, dass nicht grundsätzlich davon ausgegangen werden kann, dass sämtliche Arbeiten mit ADORA bzw. mit einer Spezifikationssprache werkzeuggestützt ablaufen. Ein gewisser Teil der Modellierungsarbeit wird (immer) mit Papier und Bleistift stattfinden. Hier ist es hilfreich, wenn der Unterschied zwischen Bildschirm- und Papierdarstellung nicht allzu gross ist.

---

[1] Die hier angeführten Szenarien wurden aus den wichtigsten Elementarszenarien zusammengefasst. Sie könnten genauso in elementarer Form präsentiert werden, so dass aus den drei Szenarien S1 - S3 dann acht bis zehn Szenarien werden. Da dies nach Ansicht des Autors hier aber keinen zusätzliche Nutzen erbringt, wurde darauf verzichtet.

R1 Der Detaillierungsgrad der in einer Sicht dargestellten Elemente ist frei wählbar (G1, S1, S2).

R2 Sichten stellen in einer ausgewogenen Menge gleichzeitig lokales Detail und globalen Kontext bereit (G2, S1, S2).

R3 Dargestellte Objekte überlappen sich nicht (G1, G2, S1, S2).

R4 Es ist möglich, in einer Sicht mehrere Bereiche oder Objekte detailliert darzustellen (G1, G2, S1.2, S1.3, S2.2, S2.2).

R5 Es gibt keine Seiteneffekte oder Abhängigkeiten zwischen logischer und physikalischer Navigation (G2, G4, S1, S2 vs. S3). Speziell physikalische Navigation verändert die logische Struktur einer Sicht nicht (G4, S3).

R6 Proportionierung und Positionierung der Objekte eines Modells werden bei der Sichtengenerierung beachtet (G4).

R7 Sichten werden so generiert, dass sie grundsätzlich auch von Hand auf Papier gezeichnet werden könnten (vgl. Abb. 56a und 56b).

9.2 Grundlegende Entwurfsentscheidungen

Anhand der Anforderungen R1 - R7 skizzieren sich die wichtigsten Eigenschaften der Realisierung des Visualisierungskonzepts für ADORA-Modelle wie folgt:

- Variable Darstellungstiefe der Sicht (R1).
- Perspektivbasierte Betrachtung mit kontexterhaltender Sicht (DE = 0) (R2)
- Verfeinert/Vergröbert wird knoten- bzw. komponentenweise bezüglich der Darstellungstiefe (R1, R2).
- Okklusionsfreie Projektionstechnik (R3)
- Mehrfachfoki (R4)
- Projektionstechnik ohne In-Situ-Vergrösserung, Fokus nicht in der Sicht, sondern auf dem Element verankert (R5)
- Inkrementelle Anpassung anstatt vollautomatischem Layout (R6)
- 2D Projektion (R7)

Die meisten der o.g. Eigenschaften beschreiben Eigenschaften der für die Sichtengenerierung zur Anwendung kommenden Projektionstechnik. Da existierende, nicht-lineare und hybride Ansätze (siehe Kapitel 7.4.1) nicht alle diese Eigenschaften aufweisen (siehe auch Kapitel 8.1), wurde eine eigene, hybride Projektionstechnik entwickelt. Diese wird im Anschluss in Kapitel 10 ausführlich vorgestellt. Zuvor wird aber noch kurz das Konzept beschrieben, ohne jedoch auf die Algorithmik der Sichterzeugung einzugehen.

9.2.1 Präsentationsform

Präsentiert wird ein Fenster mit einer kontexterhaltenden Sicht (siehe Abb. 58). Wie auch beim Fischaugenkonzept wird eine vertikale Vergröberung des Modells angezeigt. Allerdings nur solange ausreichend Platz auf dem Anzeigegerät besteht. Ist nicht mehr genügend Platz auf dem Anzeigegerät vorhanden, so wird sukzessiv horizontal vergröbert. Die Balance zwischen lokalem Detail und globalem Kontext – und damit auch indirekt die Grösse der Sicht – bestimmt der Benutzer.

Anmerkung: Die Möglichkeit, analog zum 1-Fensterkonzept ein Mehrfensterkonzept abzuleiten, besteht grundsätzlich, wurde aber nicht weiter verfolgt.

9.2.2 Visualisierung

Die grundsätzliche Idee dieses Ansatzes ist es, eine hybride Projektionstechnik ohne In-Situ-Vergrösserung zur Visualisierung von ADORA-Modellen zu verwenden.

Die Erzeugung von Sichten ist modellgetrieben; sie folgt der Struktur und den Abstraktionen des ADORA-Modells. Die Generierung von verfeinerten und/oder vergröberten Sichten erfolgt unter Ausnützung der Dekompositionsstruktur eines ADORA-Modells. Die auf dieser Basis erzeugten Vergröberungen sind echte Abstraktionen. Sie wurden nicht ‘einfach’ durch mehr oder weniger differenziertes Weglassen von Sprachelementen erzeugt, sondern durch die Verwendung einer durch die Notation gegebenen und explizit vorhandenen Modellstrukturierung als Teil-Ganzes-Hierarchie.

Eine Sicht integriert lokales Detail und globalen Kontext. Dies vereinfacht die Orientierung und hält den kognitiven Ballast gering. Da dieses Konzept es erlaubt, für den Betrachter momentan weniger interessante Elemente in einer vergröberten Form, zusammen mit Elementen darzustellen, welche detailliert gezeigt werden, ist es sehr gut geeignet, die Abstraktionsmechanismen des ADORA-Objektmodells zu unterstützen.

Es werden hierbei keine Annahmen darüber getroffen, welche weiteren Elemente ausser den zu verfeinernden/vergröbernden wie detailliert dargestellt werden (vgl. Kapitel 9.2.3). Der Benutzer bestimmt, welche Elemente (Objekte, Zustände, etc.) wie detailliert dargestellt werden. Davon abhängige Elemente (Beziehungen, Zustandsübergänge, etc.) werden in Abhängigkeit von der Sichtbarkeit ihrer Konstituenten dargestellt (siehe Abb. 58a vs. 58b). Sichten werden inkrementell generiert, d.h. auf Basis einer existierenden Sicht. Das Layout der bestehenden Sicht und damit auch deren Sekundärnotation wird bei Vergröberungen/Verfeinerungen soweit wie möglich erhalten. So werden beispielweise bei einer Verfeinerung ausschliesslich diejenigen Anpassungen vorgenommen, die notwendig sind, um den Platz für exakt die Details der zu verfeinernden Komponente zu schaffen. Die Proportionierung der übrigen Elemente wird nicht verändert. Sie werden aber entsprechend dem zusätzlichen Platzbedarf verschoben, wobei jedoch die relative Positionierung der Elemente zueinander soweit wie möglich erhalten bleibt.

Ebenfalls um den kognitiven Ballast möglichst gering zu halten, unterstützt das Konzept Mehrfachfoki. Ohne dass eine Sicht degeneriert (siehe Abb. 54 sowie Kapitel 9.2.3), kann jederzeit

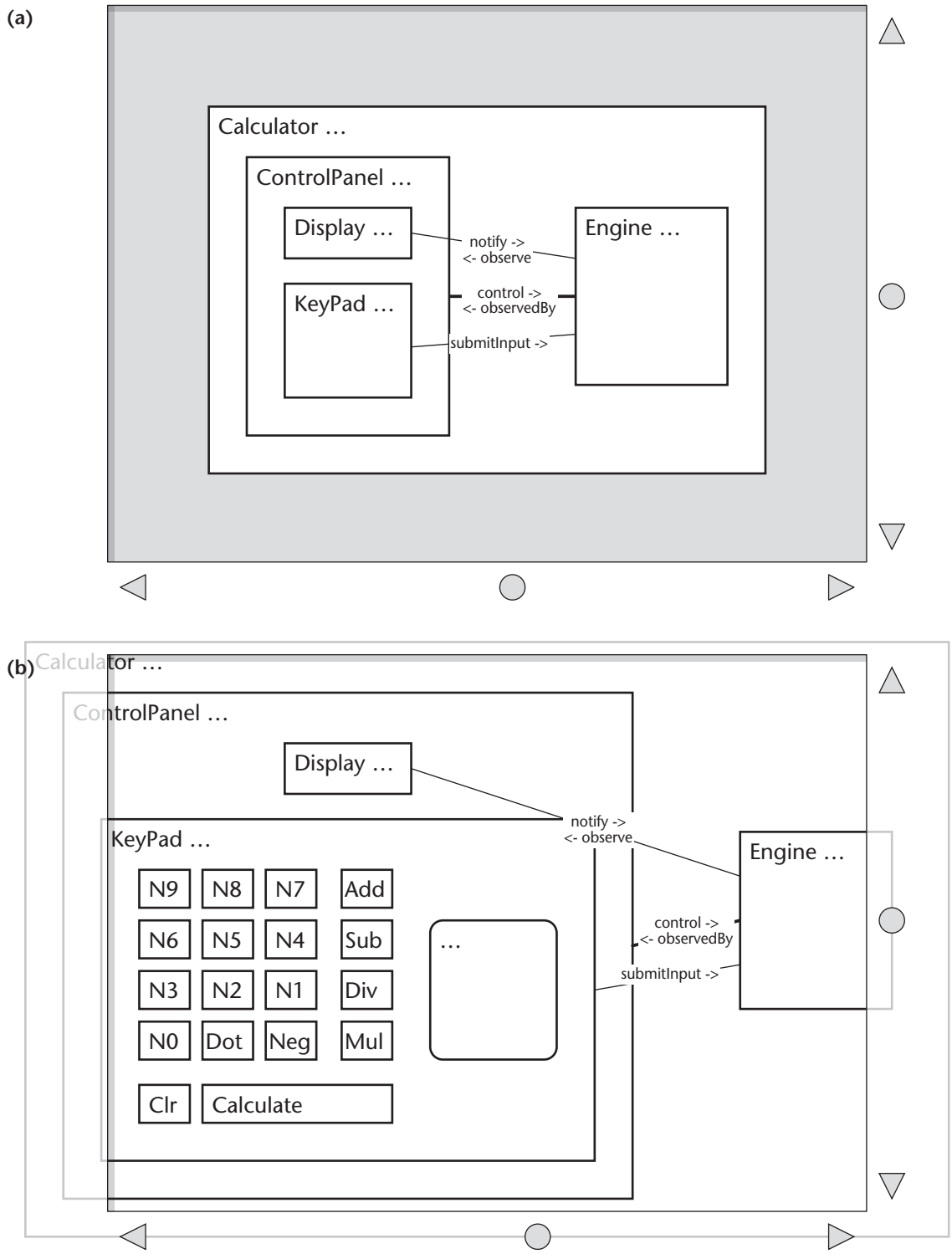


Abbildung 58: *Zwei Sichten des Taschenrechnermodells, die unter Anwendung der zur Visualisierung von ADORA-Modellen entwickelten hybriden Projektionstechnik entstanden sind.*
Die Sicht (b) wurde auf Basis der Sicht (a) erzeugt, indem die Darstellung der Tastatur (KeyPad) verfeinert wurde. Die Projektionstechnik arbeitet nicht in-situ-vergrössernd, so dass die entstandene Sicht mehr Platz benötigt als Sicht (a). Die Technik arbeitet okklusionsfrei, was die Objekte betrifft, aber die Beziehung $rel_{notify/observe}$ zeigt in Sicht (b) eine Überlappung mit dem Objekt KeyPad. Wie diese Probleme angegangen werden können, wird in Kapitel 10.2 angesprochen.

ein zusätzliches Element verfeinert werden, oder es kann ein bereits detailliert dargestelltes Element weiter verfeinert werden (vgl. auch Anmerkungen zu ‘Fokus auf Fokus’ in Kapitel 6.2.2 sowie 6.2.3), allerdings um den Preis des Verlusts der In-Situ-Vergrösserung.

9.2.3 Navigation

Da die Projektionstechnik nicht in-situ-vergrössernd arbeitet, muss physikalisch navigiert werden können. Physikalisch navigiert wird in bekannter Art und Weise mittels Scrolling über Scroll Bars/Rollbalken.

Logisch navigiert wird über einen Selective Zoom/Fisheye Zoom. Es wird knotenweise bezüglich der Darstellungstiefe verfeinert oder vergröbert. Der zusätzliche Platzbedarf bei einer Verfeinerung hängt nur vom zu verfeinernden Element ab und ist somit wesentlich geringer als beim Full Zoom, wo er von sämtlichen verfeinerten Elementen abhängig ist (natürlich unter der Voraussetzung, dass kein Trivialfall vorliegt und nicht nur das aktuell verfeinerte Element weiter verfeinert werden kann). Anmerkung: Grundsätzlich würde es die Projektionstechnik erlauben, die Darstellungstiefe ganzer Bereiche oder mehrerer benachbarter Objekte auf gleicher Darstellungsebene gleichzeitig zu ändern und so einen Full Zoom zu emulieren.

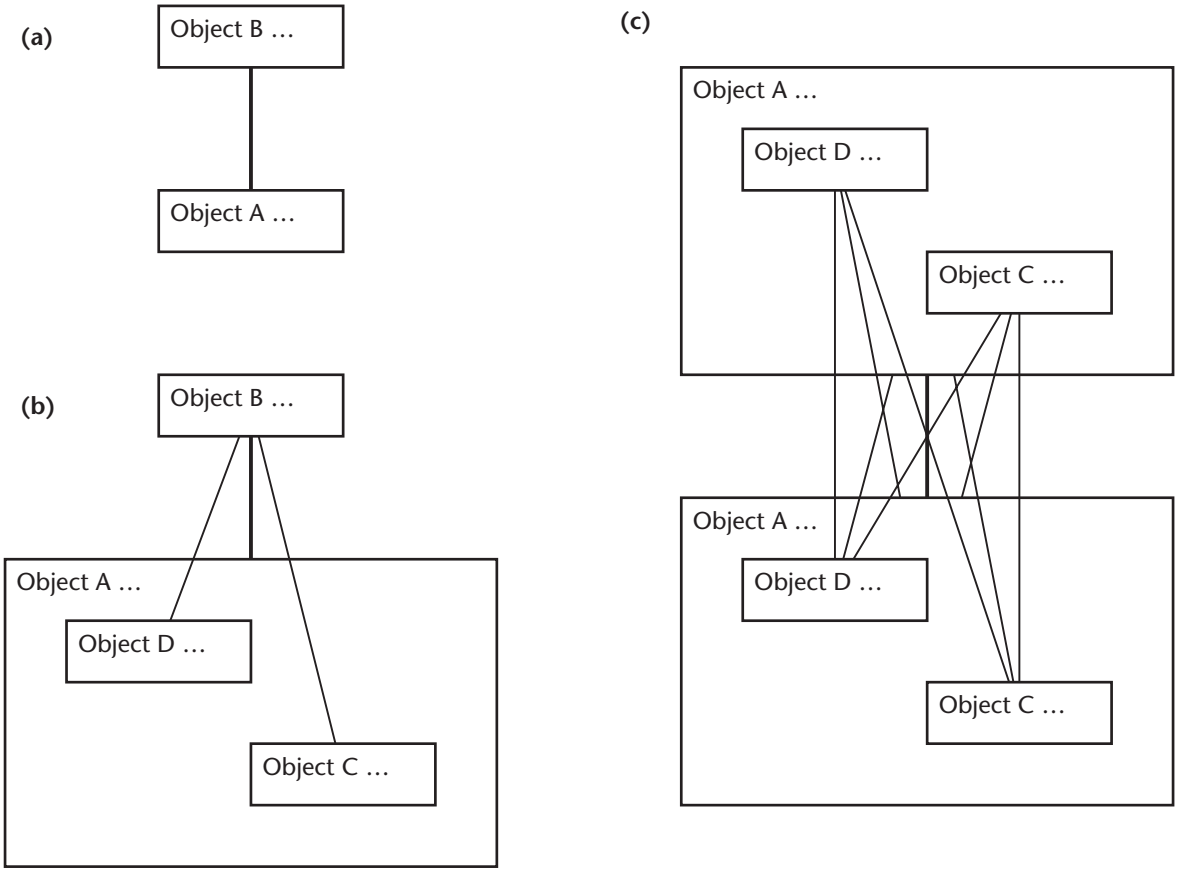


Abbildung 59: *Abhängige Objekte wie Beziehungen, Zustandsübergänge, etc. werden entsprechend der Sichtbarkeit ihrer konstituierenden Objekte angezeigt.*

Da Mehrfachfoki möglich sind und abhängige Elemente[1] nur dann dargestellt werden, wenn ihre konstituierenden Elemente sichtbar sind, kann benutzerseitig das Problem der nicht dargestellten kooperierenden Objekte (siehe Kapitel 7.3) einfach und elegant gelöst werden. Zuerst sichtbar ist die Abstraktion eines Modellelements, wie z.B. einer Beziehung. An der Darstellung des Elements ist zu erkennen, ob sich bei einer Verfeinerung weitere Details auftun oder nicht, so dass hierfür keine unnötigen Navigationsschritte anfallen (Stichwort: brauchbare Primärnotation). Werden im Zuge einer Verfeinerung Elemente sichtbar, so werden die abhängigen Elemente – im Fall einer Beziehung beispielsweise deren Unterbeziehungen – genau dann angezeigt, wenn sämtliche konstituierenden Objekte sichtbar sind (siehe Abb. 59).

Damit physikalische Navigation die logische Struktur einer Sicht nicht verändert, werden Foki nicht in der Sicht, sondern auf den Elementen verankert (vgl. Abb. 53 auf Seite 88) – der Fokus wird beim Scrollen mitbewegt. Ein Objekt bleibt solange verfeinert dargestellt, bis es der Benutzer wieder vergröbert. Ein Objekt wird nicht automatisch vergröbert oder verfeinert, weil es sich in einen bestimmten Bereich des Anzeigegeräts befindet. Veränderungen bei der Darstellung von Kontextinformation und Detaillierungsgrad werden also ausschliesslich explizit und niemals implizit durchgeführt.

[1] Mit abhängigen Elementen sind Modell-Elemente gemeint, die nur im Zusammenhang mit ihren konstituierenden Elementen existieren, wie beispielsweise Beziehungen, Zustandsübergänge, etc.

10 ALGORITHMISCHE REALISIERUNG

In diesem Kapitel wird die algorithmische Realisierung der Projektionstechnik erläutert, die für die Visualisierung von ADORA-Modellen zur Anwendung kommt. Bei dieser Projektionstechnik handelt es sich um eine hybride Technik. Sie generiert kontexterhaltende Sichten bei garantierter Okklusionsfreiheit, ist aber nicht in-situ-vergrössernd, wenn verfeinert wird.

10.1 Grundkonzept

Das Prinzip der hier vorgestellten Projektionstechnik besteht auf der Verschiebung von Elementen ‘on demand’. Bei einem Zoom-Schritt wird die Grösse des zu verfeinernden bzw. vergröbernden Elements verändert, um entweder Platz für die zusätzlich anzuzeigenden Details zu schaffen (einzoomen) oder um Platz freizugeben (auszoomen). Sämtliche Elemente, die von dieser Grössenänderung betroffen sind, werden radial verschoben und zwar um den Betrag, um den das Element in dieser Richtung vergrössert oder verkleinert wurde.

Der Algorithmus hierfür arbeitet auf jedem gegebenen Layout. Er justiert schrittweise, versucht aber, das ursprüngliche Layout so weit wie möglich beizubehalten. Dies ermöglicht es dem Benutzer, das Layout eines Modells zu verändern, ohne dass diese Anpassungen verloren gehen, wenn verfeinert oder vergröbert wird (vgl. Sekundärnotation, Kapitel 9).

Anmerkung: Diese Charakteristik ist mit Ansätzen, die versuchen den gesamten Layoutprozess zu automatisieren, (nahezu) nicht zu erreichen.

Basisalgorithmus

Soll beispielsweise ein Objekt explodiert werden – so dass seine innere Struktur (Komponenten, Zustände, etc.) sichtbar wird – werden alle Objekte auf der gleichen Darstellungsebene wie dieses Objekt mit ihren Komponenten entsprechend verschoben. Ist das zu explodierende Objekt Komponente eines anderen Objekts, so wird nun die Grösse dieses Objekts angepasst, und die Objekte auf dieser Darstellungsebene werden entsprechend verschoben usw.

Nun wird der Algorithmus im Detail vorgestellt (siehe Abb. 60).

```
void reshape( anObject : Object )
{
    calculate new size of anObject;                                  // Schritt (1)
    for each otherObject on the same level as anObject do            // Schritt (2)
    {
        calculate shift_vector between anObject and otherObject;
        shift otherObject by the shift_vector
    }
    if surroundingCompositionOf( anObject ) must also be reshaped then   // Schritt (3)
    {
        reshape( surroundingComposition( anObject ) )
    }
}
```

Abbildung 60: *Pseudocode für den Zoom-Algorithmus*

Der Basisalgorithmus besteht aus drei Schritten (siehe auch Kommentare in Abb. 60), die ggf. rekursiv wiederholt werden. Diese drei Schritte sind:

(1) Berechne die neue Grösse des Objekts, welches verfeinert dargestellt werden soll. Die neue Grösse des zu verfeinernden Objekts ist von den neu zu zeigenden Details des Objekts, also seiner internen Struktur (Komponenten, Zustände, etc.) abhängig. Die neue Grösse des Objekts muss mindestens so dimensioniert sein, dass alle neu anzuzeigenden Objekte dargestellt werden können.

(2) Berechne für jedes umgebende Objekt auf der gleichen Darstellungsebene bzw. Stufe den Verschiebevektor *V,* und verschiebe das Objekt um diesen Vektor. Der Verschiebevektor beschreibt hierbei für jedes zu verschiebende Objekt, welche Verschiebung in x- und y-Richtung für dieses Objekt, als Konsequenz der Grössenänderung des vergröbert oder verfeinert darzustellenden Objekts, vorzunehmen ist (siehe Abb. 61).

(3) Wenn umgebende Kompositionen existieren, dann müssen auch diese entsprechend vergrössert oder verkleinert werden. Vergrössere oder verkleinere diese rekursiv von innen nach aussen, und verfahre dabei entsprechend den Schritten (1) und (2).

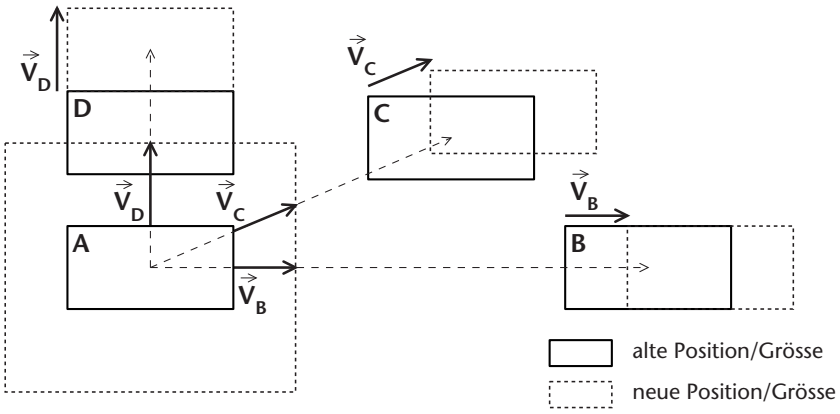


Abbildung 61: *Illustration des Zoom-Algorithmus.*

10.1.1 Berechnung der neuen Objektgrösse

Ein Objekt A mit den Komponenten A^1, A^2 .. A^n sei dargestellt durch das Rechteck R^A mit den Punkten P_1 (x_{p1}, y_{p1}), P_2 (x_{p2}, y_{p2}), P_3 (x_{p3}, y_{p3}) und P_4 (x_{p4}, y_{p4}) (siehe Abb. 62). Die neue Grösse des Rechtecks R'^A für das Objekt A im verfeinerten Zustand wird dann folgendermassen berechnet:

$R'^A = [\ P'_1, P'_2, P'_3, P'_4\ ]$

$P'_1 = (x_{p'1}, y_{p'1}) = (\ Max^{i=1..n}(\ x_{p'1}^{Ai}) + c\ ,\ Max^{i=1..n}(\ y_{p'1}^{Ai}) + c\ )$

$P'_2 = (x_{p'2}, y_{p'2}) = (\ Min^{i=1..n}(\ x_{p'2}^{Ai}) - c,\ Max^{i=1..n}(\ y_{p'2}^{Ai}) + c\ )$

$P'_3 = (x_{p'3}, y_{p'3}) = (\ Min^{i=1..n}(\ x_{p'3}^{Ai}) - c,\ Min^{i=1..n}(\ y_{p'3}^{Ai}) - c\ )$

$P'_4 = (x_{p'4}, y_{p'4}) = (\ Max^{i=1..n}(\ x_{p'4}^{Ai}) + c,\ Min^{i=1..n}(\ y_{p'4}^{Ai}) - c\ )$

wobei jeweils $c \geq 0$

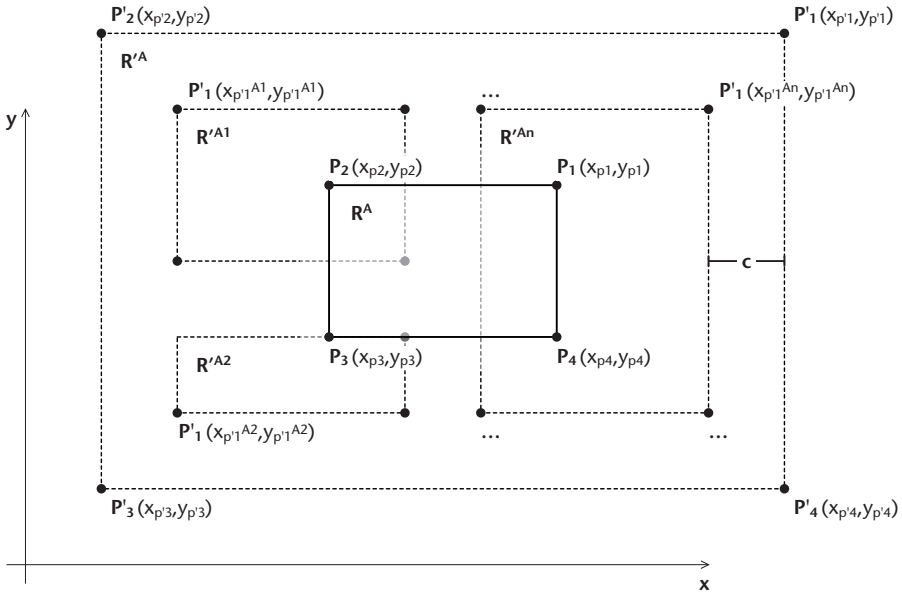


Abbildung 62: *Berechnung der Grösse für ein zu verfeinerndes Objekt.*

Die neue Objektgrösse berechnet sich also als das kleinste Rechteck, welches die direkten Komponenten von A beinhalten kann, plus einem Platzhalter c, welcher einen minimalen Abstand der Komponenten zur Komposition vorgibt. Wird eine Komponente von A verfeinert, wiederholt sich der ganze Vorgang rekursiv. Sind die Koordinaten der Komponenten nicht absolut, sondern relativ zur Komposition angegeben – was i.d.R. der Fall sein dürfte – so muss nach der Berechnung von R'^A noch eine Verschiebung erfolgen, so dass die Mittelpunkte von R^A und R'^A zusammenfallen.

Alternativ zur Berechnung eines umschliessenden Rechtecks kann die neue Grösse auch benutzerdefiniert sein, zumindest insofern die benutzerdefinierte Grösse die minimal notwendige Grösse nicht unterschreitet. Für ein Objekt werden dann zwei Grössenangaben vermerkt; die Grösse im vergröberten Zustand und die Grösse im verfeinerten Zustand, wenn alle Komponenten, Zustände, etc. vergröbert sind.

10.1.2 Verschiebevektor I

Nachdem die neue Grösse bekannt ist, kann der Verschiebevektor berechnet werden. Basis für die Berechnung des Verschiebevektors bildet der in [Be+98a] aufgeführte Algorithmus. Dieser Algorithmus arbeitet allerdings unter bestimmten Umständen nicht okklusionsfrei. Bevor die okklusionsfreie Variante vorgestellt wird, soll dieser Algorithmus erklärt werden.

Der Verschiebevektor (siehe Vektor *V* in Abb. 61 sowie 64) gibt für jedes Objekt an, um welchen Betrag und in welche Richtung es bei einer Verfeinerung oder Vergröberung zu verschieben ist. Wie dieser Vektor zustande kommt und was er bei der Projektion bewirkt, wird nachfolgend dargelegt.

Ansatzpunkt für die Verschiebung ist stets der Mittelpunkt eines Objekts. Der besseren Beschreibbarkeit wegen wird im folgenden angenommen, dass Objekte hier durch Rechtecke

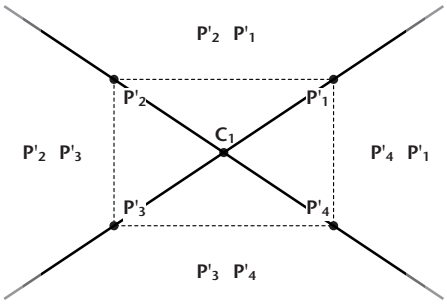


Abbildung 63: *Quadranten*

repräsentiert sind. Gegeben sei also der Mittelpunkt eines zu verschiebenden Rechtecks. Das Objekt C_2 in Abb. 64 ist ein solches Objekt, welches hier bedingt durch die Grössenänderung von C_1 verschoben werden muss.

Um den Verschiebevektor *V* berechnen zu können, wird als erstes herausgefunden, in welchen der vier Quadranten $(P'_1P'_2)$, $(P'_2P'_3)$, $(P'_3P'_4)$, $(P'_4P'_1)$ der Mittelpunkt des zu verschiebenden Rechtecks fällt. Die Quadranten bilden sich über Linien durch die gegenüberliegenden Eckpunkte des zu expandierenden Rechtecks (siehe Abb. 63).

Auf Basis der Strahlensätze lässt sich entscheiden, ob der Mittelpunkt des zu verschiebenden Rechtecks in einen bestimmten Quadranten fällt. Im Beispiel (siehe Abb. 64) fällt der Mittelpunkt C_2 in den Quadranten $P'_1P'_2$ da folgendes gilt:

$$C_2 \in P'_4P'_1 \leftrightarrow \left[(x_{c_1} < x_{c_2}) \wedge y_{p'_4} < (x_{p'_1} - x_{c_1}) \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}} + y_{c_1} \leq y_{p'_1} = true \right]$$

Wie bereits erwähnt, werden Elemente radial verschoben und zwar um den Betrag, um den in die entsprechende Richtung vergrössert oder verkleinert wurde.

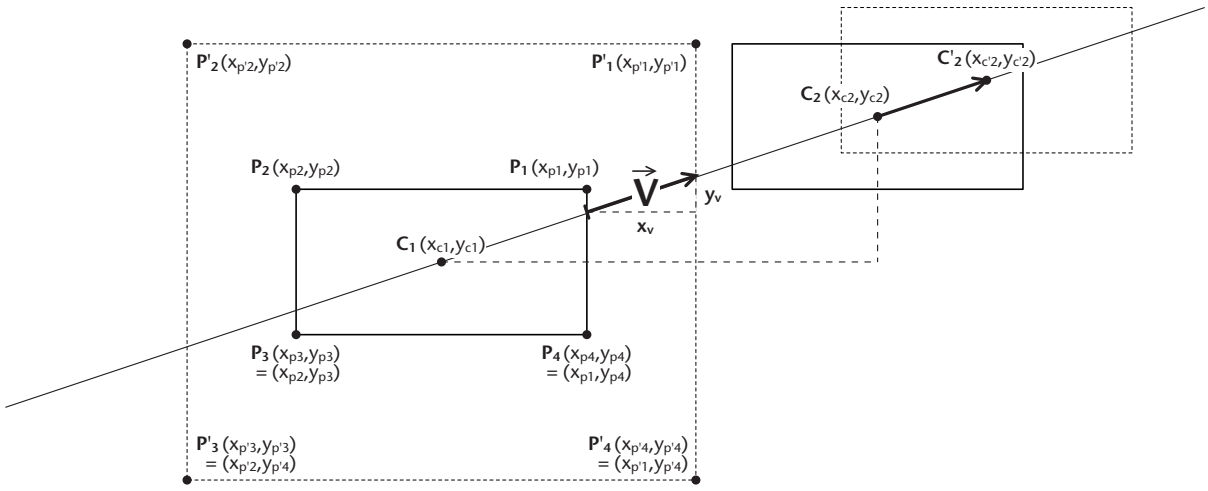


Abbildung 64: *Illustration der Daten, welche notwendig sind, um für ein Element/Rechteck den Verschiebevektor V zu berechnen.*

Wenn der Mittelpunkt des zu verschiebenden Rechtecks (siehe C_2 in Abb. 64) wie hier im Beispiel nordöstlich innerhalb des Quadranten $P'_4P'_1$ liegt, dann wird dieses nach rechts oben verschoben: Nach rechts um den Betrag, um welches sich das expandierte Rechteck (C_1) nach rechts vergrössert hat und nach oben, um den Betrag, um welchen sich das expandierte Rechteck C_1 im Verhältnis zum zu verschiebenden Rechteck C_2 nach oben vergrössert hat. Liegt der Mittelpunkt in einem anderen Quadranten, wird sinngemäss verfahren (siehe allgemeine Formel zur Berechnung des Verschiebvektors).

Der Verschiebevektor $\vec{V} = (x_v, y_v)$ für C_2 wird dann wie folgt berechnet:

$$x_v = x_{p'_1} - x_{p_1}$$

Nach den Strahlensätzen gilt:

$$\frac{y_v}{x_v} = \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}}$$
$$\rightarrow y_v = x_v \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}}$$

Verallgemeinert man dies für alle vier Quadranten, so berechnet sich der Verschiebevektor $\vec{V} = (x_v, y_v)$ wie folgt:

$$\vec{V} = \begin{cases} \text{wenn } (x_{c_1} < x_{c_2}) \wedge y_{p'_4} < (x_{p'_1} - x_{c_1}) \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}} + y_{c_1} \leq y_{p'_1} \text{ dann } \begin{pmatrix} x_v = x_{p'_1} - x_{p_1} \\ y_v = x_v \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}} \end{pmatrix} \\ \text{wenn } (y_{c_1} < y_{c_2}) \wedge x_{p'_2} \leq (y_{p'_2} - y_{c_2}) \frac{x_{c_2} - x_{c_1}}{y_{c_2} - y_{c_1}} + x_{c_1} < x_{p'_1} \text{ dann } \begin{pmatrix} x_v = y_v \frac{x_{c_2} - x_{c_1}}{y_{c_2} - y_{c_1}} \\ y_v = y_{p'_2} - y_{p_2} \end{pmatrix} \\ \text{wenn } (x_{c_1} > x_{c_2}) \wedge y_{p'_3} < (x_{p'_3} - x_{c_1}) \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}} + y_{c_1} \leq y_{p'_2} \text{ dann } \begin{pmatrix} x_v = x_{p'_3} - x_{p_3} \\ y_v = x_v \frac{y_{c_2} - y_{c_1}}{x_{c_2} - x_{c_1}} \end{pmatrix} \\ \text{wenn } (y_{c_1} > y_{c_2}) \wedge x_{p'_3} \leq (y_{p'_4} - y_{c_1}) \frac{x_{c_2} - x_{c_1}}{y_{c_2} - y_{c_1}} + x_{c_1} < x_{p'_4} \text{ dann } \begin{pmatrix} x_v = y_v \frac{x_{c_2} - x_{c_1}}{y_{c_2} - y_{c_1}} \\ y_v = y_{p'_4} - y_{p_4} \end{pmatrix} \end{cases}$$

Die hier angeführte Berechnung des Verschiebevektors[1] geht – wie gesagt – davon aus, dass es sich bei den zu verschiebenden Objekten um Rechtecke handelt oder genauer formuliert, dass die Gestalt der Objekte von aussen betrachtet rechteckig ist. Ist dies nicht der Fall und es handelt sich um Vielecke, muss die Fallunterscheidung zur Berechnung des Verschiebevektors entsprechend der Anzahl Ecken geändert werden. Handelt es sich um Kreise, so ist keine Fallunterscheidung mehr nötig, und der Verschiebevektor kann über trigonometrische Funktionen berechnet werden.

Dieses Verfahren arbeitet nicht mehr einwandfrei, wenn:

(1) die Hülle der zu verschiebenden Elemente nicht konvex ist, d.h. nicht mehr durch ein Vieleck ohne Einbuchtungen oder einen Kreis repräsentiert wird oder wenn

(2) nicht sämtliche Rechtecke, Vielecke, etc. im geometrischen Sinn ähnlich zueinander sind (siehe Abb. 65).

Fall (1) ist unproblematisch, da keines der ADORA-Modellelemente durch Vielecke mit Einbuchtungen repräsentiert wird und selbst wenn, könnte eine konvexe Hülle in Form eines entsprechenden umschreibenden Vielecks berechnet und als Substitut für Verschiebungszwecke herangezogen werden.

Fall (2) ist problematischer. Die Okklusionen sind zwar konstant, d.h. eine einzelne Okklusion nimmt nicht beständig zu, wenn verfeinert wird. Mit dem gegebenen Algorithmus zur Bestimmung des Schiebevektors lassen sich Okklusionen jedoch nur dann vollständig verhindern, wenn (zu Verschiebungszwecken) generell mit ähnlichen Rechtecken gearbeitet würde (siehe Abb. 65b). Offensichtlich ist die Unterscheidung in die vier Quadranten nicht ausreichend.

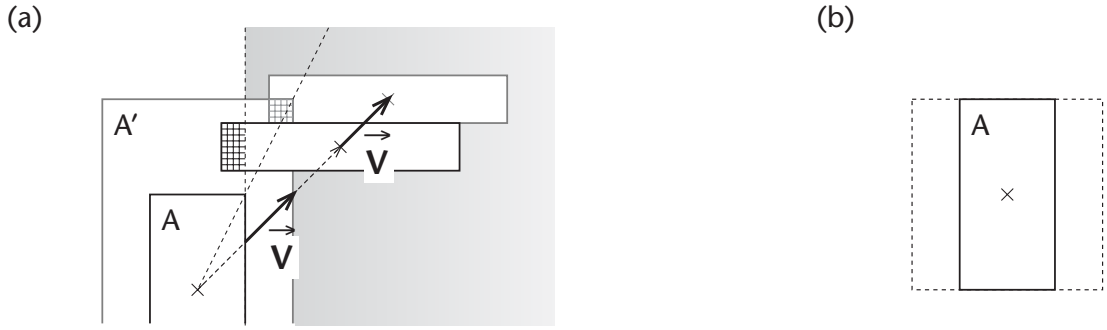


Abbildung 65: *Okklusionen können auftreten, wenn die Rechtecke nicht ähnlich zueinander sind. (a) In diesem Fall werden nur die Rechtecke, die sich vollständig in der grau markierten Halbebene befinden, korrekt verschoben. Der Teil des zu verschiebenden Rechtecks, der sich nicht in der grau markierten Halbebene befindet, bildet eine Okklusion. (b) Sind alle Rechtecke ähnlich zueinander oder werden sie (für Verschiebungszwecke) ähnlich gemacht, z.B. indem zu jedem Rechteck ein umschreibendes Quadrat bestimmt wird, treten keine Okklusionen auf. Dies geschieht allerdings auf Kosten des Platzverbrauchs, inbesondere wenn Elemente sehr flach oder sehr hoch sind.*

[1] In [Be+98a] wird der Verschiebevektor äquivalent berechnet – jedoch nicht direkt, sondern in zwei Schritten und unter Verwendung einer Verschiebefunktion *f* sowie ihrer Inversen.

10.1.3 Verschiebevektor II

Für die okklusionsfreie Variante wird nicht mehr in die vier Quadranten eingeteilt, sondern es werden vier Halbebenen (siehe *N, S, W, O* in Abb. 66) unterschieden. In Abhängigkeit davon, in welcher dieser Halbebenen sich der Mittelpunkt eines Elements befindet, können bestimmte Mindestdistanzen für die Verschiebung vorausgesetzt werden. Fällt der Mittelpunkt eines zu verschiebenden Elements in *genau* eine dieser Halbebenen, so bleibt im Prinzip alles beim alten und der Verschiebevektor *V* bildet sich wie gehabt. Fällt der Mittelpunkt eines zu verschiebenden Elements in zwei der Halbebenen, so muss dieses Element um den vollen Betrag in x und y-Richtung verschoben werden, um eine Okklusion sicher zu vermeiden.

Befindet sich beispielsweise der Mittelpunkt eines Elements in Halbebene *N*, so muss das Element den vollen Expansionsbetrag nach oben (siehe *n* in Abb. 66) verschoben werden, um eine Okklusion zu vermeiden. Befindet sich der Mittelpunkt in der Halbebene *E*, muss um den vollen Betrag nach rechts verschoben werden, usw. Befindet sich der Mittelpunkt eines Elements in Halbebene *N* und in Halbebene *E*, so muss um den vollen Betrag nach oben *und* den vollen Betrag nach rechts verschoben werden. Für Elemente, deren Mittelpunkt in zwei anderen Halbebenen liegt, gilt Entsprechendes.

Der Verschiebevektor *V* wird hierfür über acht Fallunterscheidungen berechnet (vgl. S. 113). Der besseren Lesbarkeit wegen wurden zusätzlich die Variablen *n, s, w, e* eingeführt (siehe

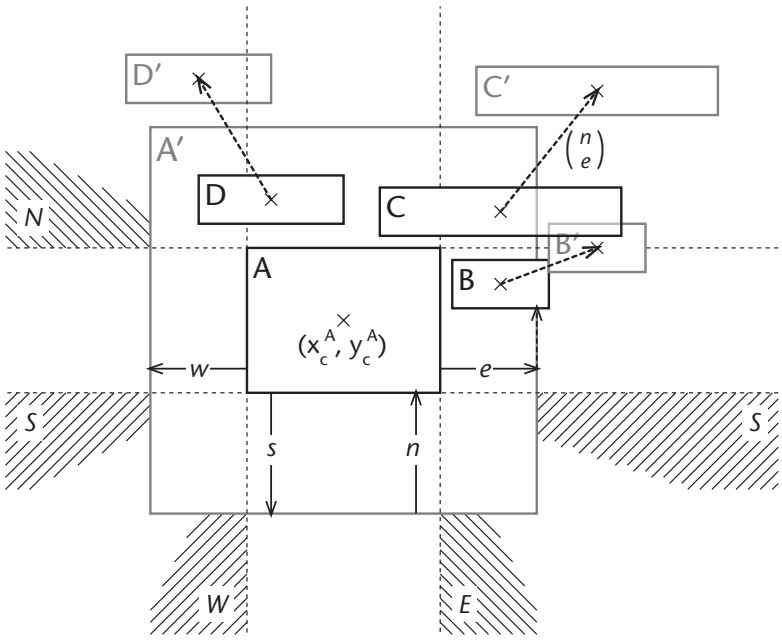


Abbildung 66: *Okklusionsfreie Variante des Algorithmus.*
Es werden vier sog. Halbebenen N, S, W, O unterschieden (siehe Schraffur). Elemente, deren Mittelpunkt in zwei Halbebenen liegt, wie z.B. das Element C, werden im Zuge der Expansion von A um den vollen Betrag in x- und y-Richtung verschoben.

Abb. 66), welche den absoluten Expansionsbetrag in die jeweilige Richtung wiedergeben und sich wie folgt berechnen: $n = y_{p'_2} - y_{p_2}$, $s = y_{p'_4} - y_{p_4}$, $w = x_{p'_3} - x_{p_3}$, $e = x_{p'_1} - x_{p_1}$.
Der eigentliche Verschiebevektor $\vec{V} = (x_v, y_v)$ berechnet sich dann folgendermassen:

$$
\vec{V} = \begin{cases}
\text{wenn } (c_2 \in E) \wedge \neg((c_2 \in N) \vee (c_2 \in S)) \text{ dann } \begin{pmatrix} x_v = e = x_{p'_1} - x_{p_1} \\ y_v = x_v \dfrac{(y_{c_2} - y_{c_1})}{(x_{c_2} - x_{c_1})} \end{pmatrix} \\
\text{wenn } (c_2 \in E) \wedge (c_2 \in N) \text{ dann } \begin{pmatrix} x_v = e \\ y_v = n \end{pmatrix} \\
\text{wenn } (c_2 \in N) \wedge \neg((c_2 \in E) \vee (c_2 \in W)) \text{ dann } \begin{pmatrix} x_v = y_v \dfrac{(x_{c_2} - x_{c_1})}{(y_{c_2} - y_{c_1})} \\ y_v = n \end{pmatrix} \\
\text{wenn } (c_2 \in N) \wedge (c_2 \in W) \text{ dann } \begin{pmatrix} x_v = w \\ y_v = n \end{pmatrix} \\
\text{wenn } (c_2 \in W) \wedge \neg((c_2 \in N) \vee (c_2 \in S)) \text{ dann } \begin{pmatrix} x_v = w \\ y_v = x_v \dfrac{(y_{c_2} - y_{c_1})}{(x_{c_2} - x_{c_1})} \end{pmatrix} \\
\text{wenn } (c_2 \in W) \wedge (c_2 \in S) \text{ dann } \begin{pmatrix} x_v = w \\ y_v = s \end{pmatrix} \\
\text{wenn } (c_2 \in S) \wedge \neg((c_2 \in W) \vee (c_2 \in E)) \text{ dann } \begin{pmatrix} x_v = y_v \dfrac{(x_{c_2} - x_{c_1})}{(y_{c_2} - y_{c_1})} \\ y_v = s \end{pmatrix} \\
\text{wenn } (c_2 \in S) \wedge (c_2 \in E) \text{ dann } \begin{pmatrix} x_v = e \\ y_v = s \end{pmatrix}
\end{cases}
$$

Hierbei wird vorausgesetzt, dass keiner der Nenner in den Brüchen gleich 0 ist. Setzt man voraus, dass die Elemente vor dem Verschieben okklusionsfrei waren und trifft (o.B.d.A.) die Einschränkung, dass es keine Elemente der Grösse $(0, 0)$ geben darf, so ist dies generell der Fall.

10.1.4 Rekursive Anwendung, Koordinatensystem

Verfeinert man eine eingeschachtelte Komponente, so hat dies zur Konsequenz, dass sich die Grösse aller Kompositionen ändert, die diese Komponenten enthalten. Der Verschiebealgorithmus wird nun entsprechend der Beschreibung auf S. 109 rekursiv auf allen Stufen der Hierarchie angewendet.

Generell werden die Koordinaten eingeschachtelter Objekte (siehe B, E in Abb. 67) relativ zur Lage des Objekts (siehe A in Abb. 67) angegeben, in welchem diese Objekte direkt enthalten sind. Möchte man daran festhalten, dass der Ursprung des Koordinatensystems in der linken unteren Ecke einer Komponente liegt, so sind bei rekursiver Anwendung des Verschiebealgorithmuses nach der eigentlichen Verschiebung Anpassungen im Koordinatensystem des Objekts notwendig, welches das 'gezoomte' Objekt enthält. Bei dieser Anpassung werden die

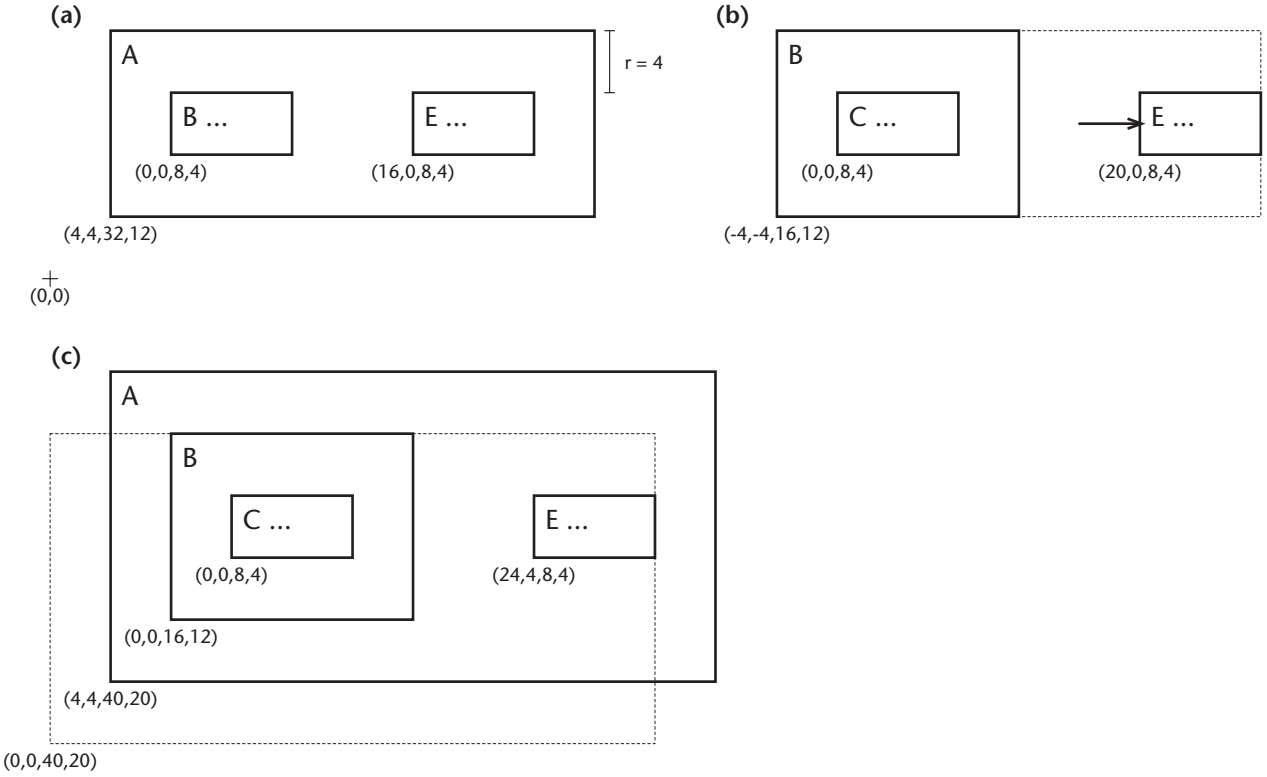


Abbildung 67: *Beispiel für die Anpassung der Koordinatensysteme bei rekursiver Anwendung. (a) Ausgangslage: Es soll ein Zoom-In auf B erfolgen. Die Koordinaten sind der besseren Übersichtlichkeit wegen jeweils in der Form (x-Position, y-Position, Breite, Höhe) angegeben. Ausserdem wird ein Rand von 4 angenommen (vgl. Kapitel 10.1.1, Berechnung der neuen Objektgrösse). (b) Zwischenschritt: Zoom-In auf B bewirkt eine Grössenänderung von B sowie eine Verschiebung von E. (c) Endzustand: Da sich die Grösse von B verändert hat, muss das Verfahren (rekursiv) auch auf A angewandt werden (vgl. Abb. 60). Das Koordinatensystem von A wurde angepasst. Hierfür wurden die Koordinaten von C und E umgerechnet, entsprechend der Grössenänderung von B nach rechts und nach oben. Der gepunktete Rahmen zeigt A nach der Expansion von B und vor der Anpassung des Koordinatensystems. Da A in keinem Objekt enthalten ist (Top-Level-Komponente) und keine Objekte auf gleicher Ebene existieren, muss lediglich das übergeordnete Koordinatensystem angepasst werden.*

Koordinaten sämtlicher Objekte umgerechnet, die ebenfalls direkte Komponente der Komposition sind, welche das ‘gezoomte’ Objekt enthält. Diese Umrechnung korrigiert (virtuell) den Nullpunkt des Koordinatensystems dieses Objekts. Und zwar genau um den Betrag, um welchen sich das ‘gezoomte’ Objekt in die entsprechende Richtung vergrössert (oder verkleinert) hat. Im Beispiel in Abb. 67 hat der Zoom-In auf B zur Folge, dass die Koordinaten der Komponenten von A, also von B und E angepasst werden müssen.

10.2 Spezialfälle und Verfeinerung

Drei Themen gilt es hier anzuschneiden: Erstens, wie arbeitet die Projektionstechnik, wenn ganze Bereiche oder auch mehrere Objekte in einem Schritt verfeinert werden (siehe Anmerkungen zur Präsentationsform im Kapitel 9.2). Zweitens, wie können Linien, die vor einer Verfeinerung überschneidungsfrei waren, so gezeichnet werden, dass sie nach einem Verfeinerungsschritt keine Objekte überschneiden (siehe Abb. 58 sowie Kapitel 9.2).

10.2.1 Emulation eines Full Zoom

Da die Projektionstechnik Mehrfachfoki erlaubt, kann unter Ausnützung dieser Eigenschaft ein Full Zoom direkt emuliert werden. Im Prinzip wird auf mehrere Objekte gleichzeitig ein Fokus gelegt. Hierfür wird ein Bereich selektiert, alle Elemente im selektierten Bereich werden bestimmt und dann in einem Schritt verfeinert (siehe Abb. 68).

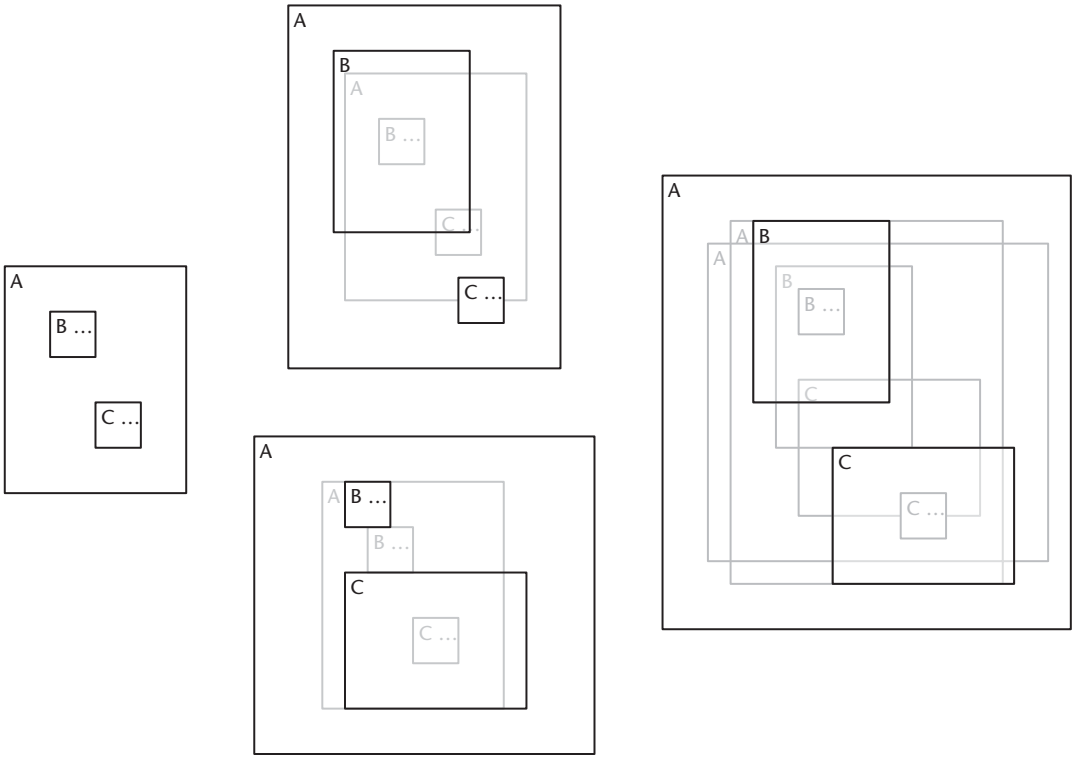


Abbildung 68: *Emulation eines Full Zooms durch gleichzeitige Verfeinerung von B und C. Hierbei ist es unerheblich, ob zuerst B und dann C verfeinert wird oder umgekehrt.*

Intern realisiert wird dies über eine sukzessive Verfeinerung der einzelnen Elemente unter Verwendung der in Kapitel 10.1 beschriebenen Projektionstechnik. Hierbei ist es unerheblich, in welcher Reihenfolge Verfeinerungs- oder Vergröberungsschritte geschehen – die letztlich entstehenden Sichten sind identisch. Das Verfahren ist also stabil (im Sinne einer Stabilität bei Sortieralgorithmen). Anmerkung: Alternativ kann auch über die Addition der einzelnen Verschiebevektoren für jedes Objekt der resultierende Vektor berechnet werden und anstatt vieler elementarer Verschiebungsschritte ein einziger, zusammengesetzter Verschiebungsschritt durchgeführt werden.

10.2.2 Das Linienproblem

… oder auch Poor-Man's-Routing

Die vorgestellte Projektionstechnik ist okklusionsfrei, d.h. vergrösserte Rechtecke (Objekte, Zustände, etc.) überlappen sich nicht. Wie bereits angedeutet (siehe Abb. 58), kann es jedoch vorkommen, dass nach einer Verfeinerung manche Linien (Beziehungen, Transitionen, etc.) über Rechtecke hinweg gezeichnet werden, obwohl dies vor der Verfeinerung nicht der Fall war. Dieser Fall wird hier als Linienproblem bezeichnet. Genau gesagt handelt es sich ja um Strecken und nicht um Linien, aber hier wird trotzdem der Begriff Linie weiterhin verwendet. Abb. 69 illustriert das Linienproblem. Auch wenn dies kein schwerwiegendes Problem ist, da nur sehr wenig verdeckt wird und der Kontext trotzdem deutbar bleibt, so ist es doch unschön und der guten Lesbarkeit einer Sicht abträglich.

Das Linienproblem kann auf unterschiedliche Weise angegangen werden. Doch bevor diesbezüglich Strategien vorgestellt werden, zuerst ein paar Worte darüber, was das Linienproblem nicht ist oder anders ausgedrückt, welche Probleme, die allgemein bei Überschneidungen von Linien auftreten, hier nicht behandelt werden.

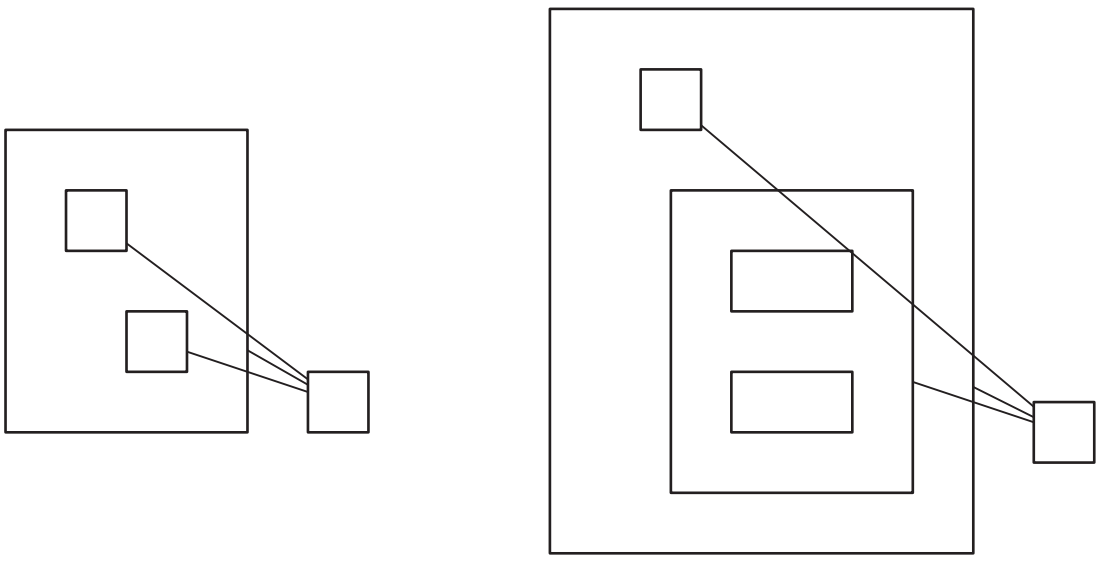


Abbildung 69: *Beim Linienproblem werden nach einer Verfeinerung Linien über ein Objekt hinweg gezeichnet, die vor der Verfeinerung nicht über dieses Objekt führten.*

Das hier behandelte Linienproblem betrifft nicht die Überschneidung von Linien untereinander, sondern die Überschneidung von Linien mit Objekten. Wenn sich Linien vor einer Verfeinerung überschnitten haben, so werden sie dies auch nachher tun. Generell die Überschneidungsfreiheit sämtlicher Linien zu garantieren, ist nicht möglich. Eine möglichst hohe Überschneidungsfreiheit von Linien zu garantieren, ist nicht Gegenstand dieser Arbeit. Im Bereich des Platinenlayouts gibt es Algorithmen, die die Anzahl von Linienüberschneidungen minimieren. Die hierfür verwendeten Routing-Algorithmen, wie z.B. der von Lee, haben allerdings eine zu hohe Zeitkomplexität, um für eine interaktive Nutzung wirklich brauchbar zu sein. Ausserdem *kann* es bei einem solchen Routing von Linien vorkommen, dass das Zeichnen einer einzigen neuen Linie eine komplett geänderte Wegführung aller bislang bestehenden Linien zur Folge hat. Wenn man nun zugrunde legt, dass Linien zur Repräsentation von Beziehungen, Zustandsübergängen, etc. verwendet werden, hätte ein radikales Rerouting zur Folge, dass durch das Ändern oder Hinzufügen einer Beziehung oder eines Zustandsübergangs u.U. nichts mehr so ausschaut wie vorher. Damit ist man wieder beim Diskussionspunkt “Sinn und Zweck der Sekundärnotation” angelangt.

Doch nun wieder zurück zum eigentlichen Linienproblem, der Frage also, wie man mit Linien umgeht, die vor einer Verfeinerung überschneidungsfrei waren, nach einer Verfeinerung jedoch Überschneidungen mit einem Objekt bzw. Rechteck aufweisen.

Grundsätzlich existieren drei Ansatzpunkte, das Linienproblem anzugehen:

(1) Die überdeckten Rechtecke repositionieren. Werden die überdeckten Rechtecke repositioniert, verliert der zur Projektion verwendete Algorithmus seine Stabilität. Es wäre dann für das Layout der entstehenden Sicht nicht mehr egal, ob zuerst ein Rechteck/Objekt und anschliessend ein anderes Rechteck/Objekt verfeinert wird (vgl. Abb. 68). Daher wird dieser Ansatzpunkt als hier nicht praktikabel angesehen und nicht weiterverfolgt.

(2) Die Linienführung so akzeptieren wie sie ist, aber (2.1) die Darstellung des durch die Linie repräsentierten Elements – beispielweise durch Transparenztechniken – so ändern, dass trotz Überdeckung noch erkennbar bleibt, was eigentlich verdeckt wurde oder (2.2) die Linie hinter dem Rechteck durchführen, so dass die Linie und nicht das Rechteck, über welches sie führt, verdeckt wird. Beide Möglichkeiten (2.1 und 2.2) sind praktikabel, werden aber nicht weiter diskutiert.

(3) Die Linienführung mehr oder weniger radikal ändern. Eine radikale Änderung der Linienführung minimiert zwar die Überschneidungen, erfordert aber genau wie bei den Linienüberschneidungen ‘echte’ Routing-Algorithmen mit den bereits angesprochenen Konsequenzen für Antwortzeiten und Sekundärnotation. Somit wird die Variante ‘radikale Änderung der Linienführung’ hier nicht weiterverfolgt. Übrig bleibt eine moderate Änderung der Linienführung. Hier werden kleine Korrekturen am Verlauf der Linie vorgenommen, die grobe Richtung der Linie wird aber beibehalten.

Moderate Änderung der Linienführung … oder auch Poor-Man's-Routing

Hier sollen nun Möglichkeiten vorgestellt werden, wie moderat Einfluss auf die Linienführung genommen werden kann.

Verteilung der Start- und Endpunkte. Wenn die Start- und Endpunkte der Linien nicht von der Elementmitte ausgehen (siehe Abb. 70a), sondern über die Ränder verteilt werden (siehe Abb. 70b), gewinnt man zusätzlichen Platz. Dieser kann genutzt werden, um die Überdeckung zu vermeiden.

Umgesetzt wird dies recht einfach, indem man bestimmt, welche Seite geschnitten wird und die dort anhängigen Start- bzw. Endpunkte gleichmässig verteilt. Nur hat diese Strategie, ausser wenn Rechtecke sehr hoch oder sehr breit sind, fast keinen Effekt (vgl. Abb. 70a und 70b).

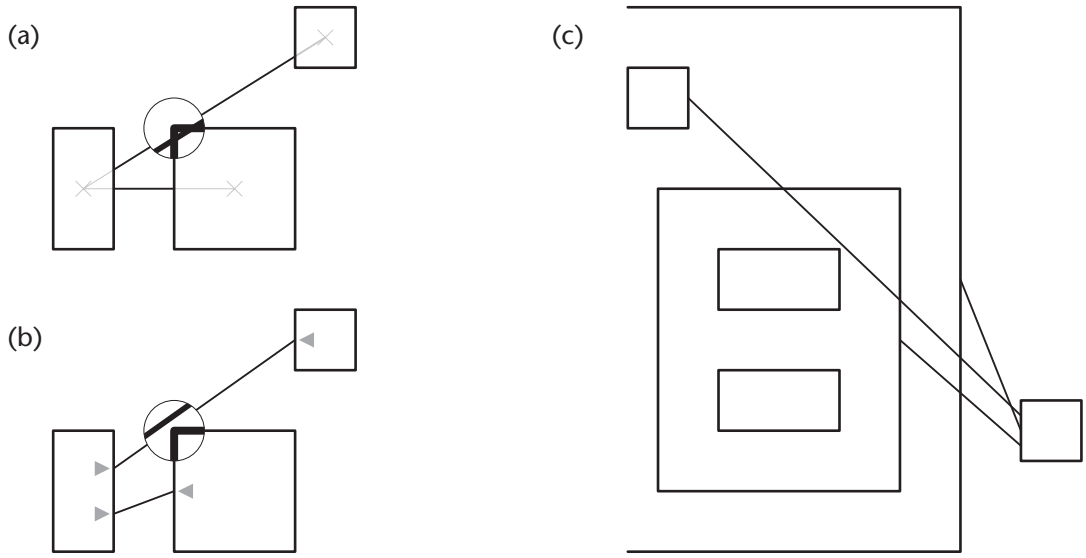


Abbildung 70: *Verteilen der Start- und Endpunkte der Linien über die Ränder als Strategie zur Lösung des Linienproblems.*

Dieses Prinzip wurde auch für Abb. 58 bereits angewandt und hat bereits dort – d.h. in keinem extra zur Demonstration des Problems konstruierten Beispiel – wenig Nutzen gebracht. Beim Beispiel aus Abb. 69 würde das Verteilen von Linien nicht nur keinen Nutzen, sondern Schaden bringen; die Überdeckung würde nicht aufgelöst. Zusätzlich würde eine Linienüberschneidung hinzukommen (vgl. Abb. 69 und 70c). Die Auflösung dieser neuen Überschneidung hätte wiederum Einfluss auf die Sekundärnotation. Verteilung der Start- und Endpunkte ist somit keine geeignete Strategie.

Einfügen von Hilfspunkten. Um die Überschneidung zu vermeiden, können Hilfspunkte einfügt werden, die die Linie um das Rechteck herumleiten. Die Umsetzung dieser Strategie erfordert zwei Schritte. Erstens, das Erkennen der Überdeckung (einfach) und zweitens, das Bestimmen der Hilfspunkte.

Bereits recht gute Ergebnisse liefert eine Strategie, die versucht, einen einzigen Hilfspunkt einzufügen (hier *Einpunktstrategie* genannt). Hierbei wird unter Beachtung der umliegenden Rechtecke sowie ggf. existierender weiterer Beziehungen versucht, einen Bereich zu bestim-

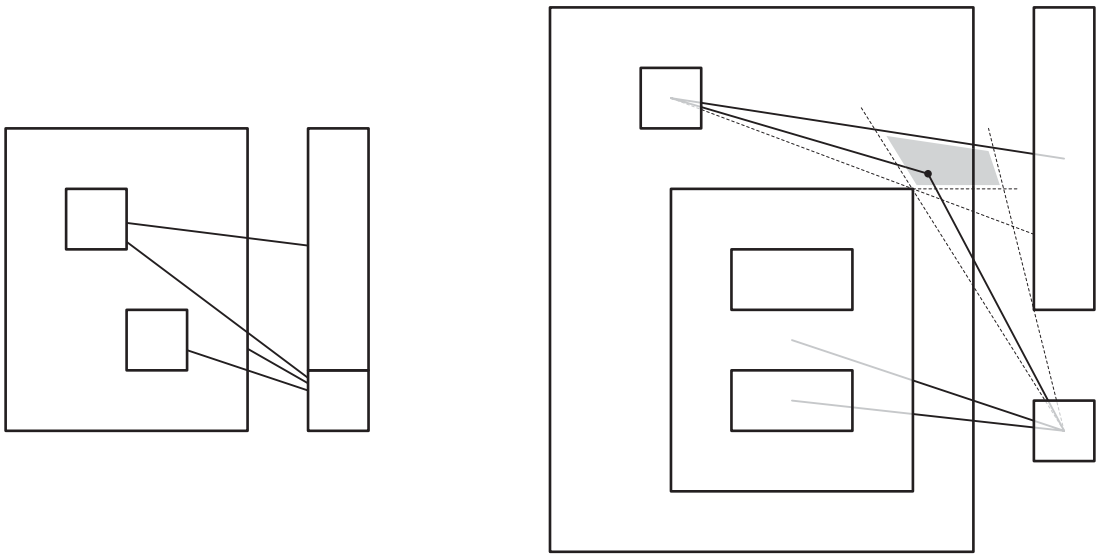


Abbildung 71: *Einpunktstrategie zur Überdeckungsvermeidung (vgl. Abb. 69).*

men, in welchem der Hilfspunkt liegen muss, damit die resultierende Linie überdeckungsfrei gezeichnet werden kann. Kann ein solcher Bereich bestimmt werden, so wird dort ein Punkt ausgewählt, welcher möglichst nahe am ursprünglichen, direkten Linienverlauf liegt (siehe Abb. 71.). Über eine entsprechende Parametrisierung kann ein bestimmter Mindestabstand des Hilfpunktes zu angrenzenden Objekten vorgegeben werden. Auf diesen Hilfspunkt wird auch die Linienbeschriftung platziert. Über die Grösse der Linienbeschriftung lässt sich zusätzlich zum unbedingt einzuhaltenden Mindestabstand ein gewünschter Abstand bestimmen, so dass auch die Linienbeschriftung keine Überdeckung erzeugt. Der neue Verlauf der Linie führt dann über diesen Punkt. Wird verfeinert oder vergröbert, so wird ausgehend vom ursprünglichen, direkten Linienverlauf versucht, die gleiche Strategie erneut anzuwenden. Kann kein Bereich

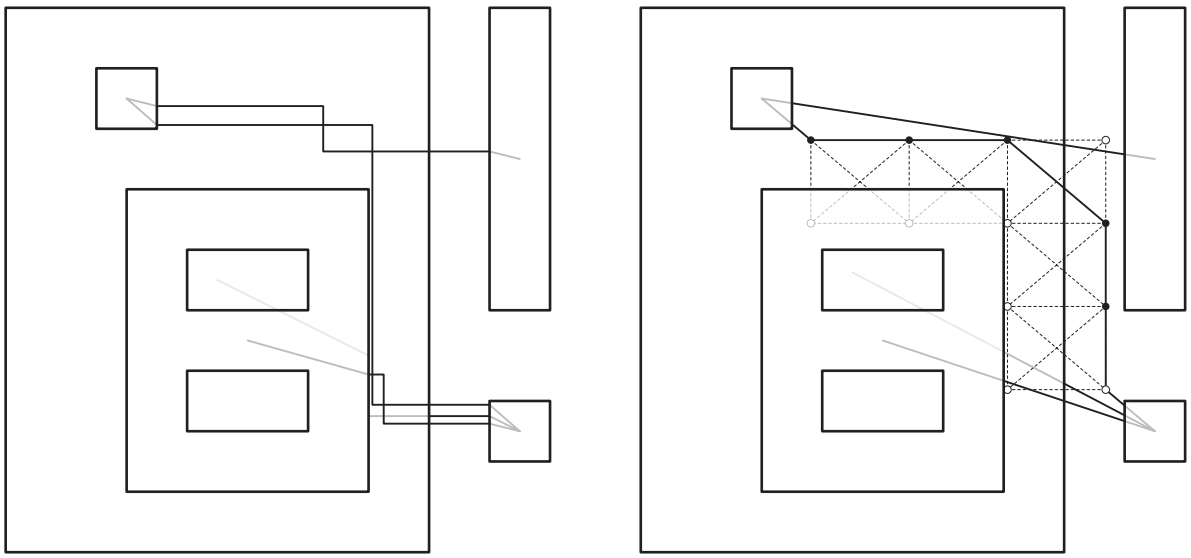


Abbildung 72: *Mehrpunktstrategien*

für den Hilfspunkt bestimmt werden, so wird die Linie wie ursprünglich gezeichnet, also direkt mit entsprechender Überdeckung

Die oben vorgestellte Einpunktstrategie ist zu einer sog. *Mehrpunktstrategie* verallgemeinerbar. Unter Angabe der Tiefe wird hierbei rekursiv zerteilt; beispielweise ergibt eine Tiefe von 2 dementsprechend $2^2 = 4$ Liniensegmente. Anschliessend wird mit jedem Linien-Segment wie oben beschrieben verfahren. Die Anzahl an Hilfspunkten steigt exponentiell mit der Tiefenangabe und somit auch der Aufwand für die Berechnung.

Andere Mehrpunktstrategien bestehen darin, einen Hilfsgraphen auf Basis eines Rasters zwischen Start- und Endpunkt zu generieren und diesen dazu zu benutzen, einen überschneidungsfreien Weg zu finden (siehe Abb. 72). Der Hilfsgraph repräsentiert die legalen Wege und wird aus den Knoten und Kanten des Rasters gebildet, die keine Überschneidung aufweisen. Für die Suche des Weges im Hilfsgraphen kommen Algorithmen à la Dijkstra zum Einsatz. Über die Anzahl der Knoten und Kanten des Hilfsgraphen lässt sich die Auflösung steuern; nach Bedarf kann die Auflösung automatisch erhöht werden, wenn kein brauchbarer Weg gefunden werden kann (und genügend Rechenleistung zur Verfügung steht). Natürlich lassen sich auch mit diesem Verfahren Überdeckungen und Überschneidungen nicht generell vermeiden.

Die Orientierung des Hilfsgraphen bestimmt auch, in welcher Form (rechtwinklig, abgeschrägt, etc.) die Linien gezeichnet werden. Zu rechtwinkligen Linien ist anzumerken, dass sie die Orientierung erschweren, wenn hinreichend viele Linienüberschneidungen auftreten, da dann nicht ohne weiteres zu erkennen ist, wohin welche Linie führt (siehe Abb. 72 und 56b).

Spline-Interpolation

Eine weitere Möglichkeit, das Linienproblem anzugehen, besteht darin, die Linie durch eine Kurve zu ersetzen, welche ihrerseits durch einen Spline repräsentiert wird. Ein einfacher Test

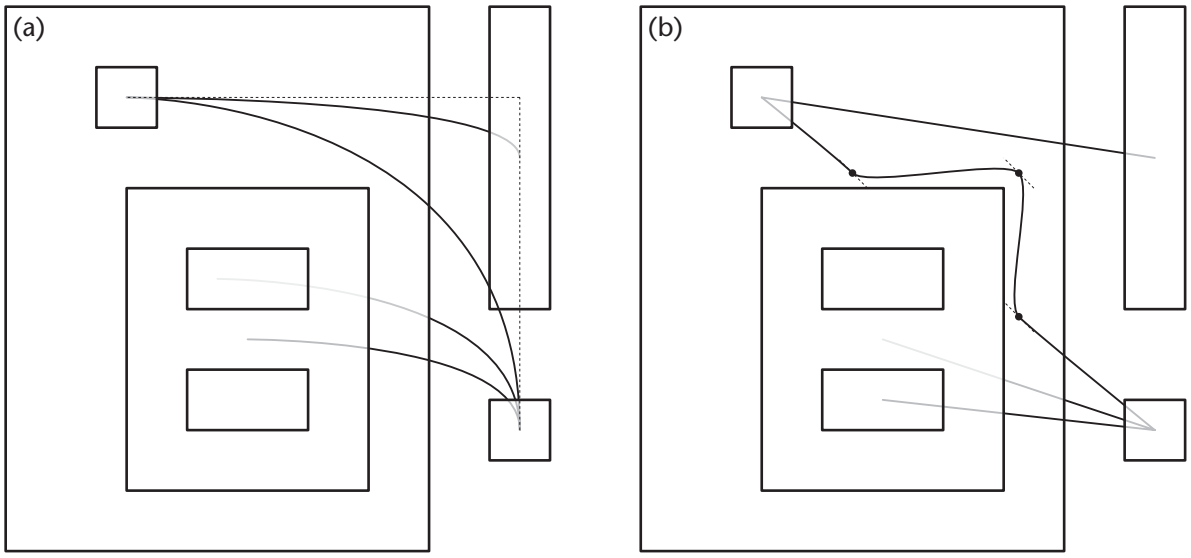


Abbildung 73: *Spline-Interpolation ohne Stützpunkte (a) und mit drei Stützpunkten (b). (a) würde für das Beispiel aus Abb. 69 gute Resultate liefern, erzeugt jedoch für das (leicht modifizierte) Beispiel aus Abb. 71 eine – wenn auch kleine – Überdeckung.*

entscheidet hierbei, in welche Richtung ‘der Bauch’ der Kurve geht. Diese Strategie liefert Verläufe, welche nahe am Original liegen und einfach zu interpretieren sind. Allerdings ist es aufwendig(er), zusätzliche Überschneidungen mit Linien sowie Überdeckungen zu erkennen und entsprechend zu reagieren.

Abschliessend sollte noch erwähnt werden, dass beliebige Kombinationen zwischen den einzelnen Strategien denkbar sind (siehe Abb. 71b), beispielsweise um die Ecken bei Einpunkt- oder Mehrpunktstrategien zu glätten. Je tiefgreifender jedoch der Eingriff in die ursprüngliche Linienführung ist, desto stärker wird der Betrachter kognitiv belastet und desto umfangreicher ist der Eingriff in die Sekundärnotation (vgl. Abb. 71, 72, 73). Mehrpunktstrategien haben hier das Nachsehen gegenüber Einpunktstrategien.

10.2.3 Positionierung von Linienbeschriftungen

Linienbeschriftungen müssen zum einen positioniert werden und benötigen zum anderen Platz. Auch hier sind verschiedene Varianten denkbar. Die Hauptentscheidung, die es hier zu treffen gilt, ist: Soll eine Linienbeschriftung als eigenständiges, vom Benutzer zu positionierendes Element angesehen werden oder nicht.

Vorteile von benutzerpositionierbaren Linienbeschriftungen sind, dass benutzerseitig mehr Möglichkeiten bestehen, auf die Sekundärnotation Einfluss zu nehmen und dass sie als Stützpunkte verwendet werden können, um Linienüberschneidungen zu vermeiden. Der Hauptnachteil ist, dass u.U. auch die Linienbeschriftung manuell verschoben werden muss, wenn eines der konstituierenden Elemente, welche über die Linie verbunden sind, manuell verschoben wird. Dies erzeugt zusätzlichen Navigationsaufwand und erhöht den kognitiven Ballast. Benutzerposionierbare Linienbeschriftungen werden daher hier nicht weiterverfolgt.

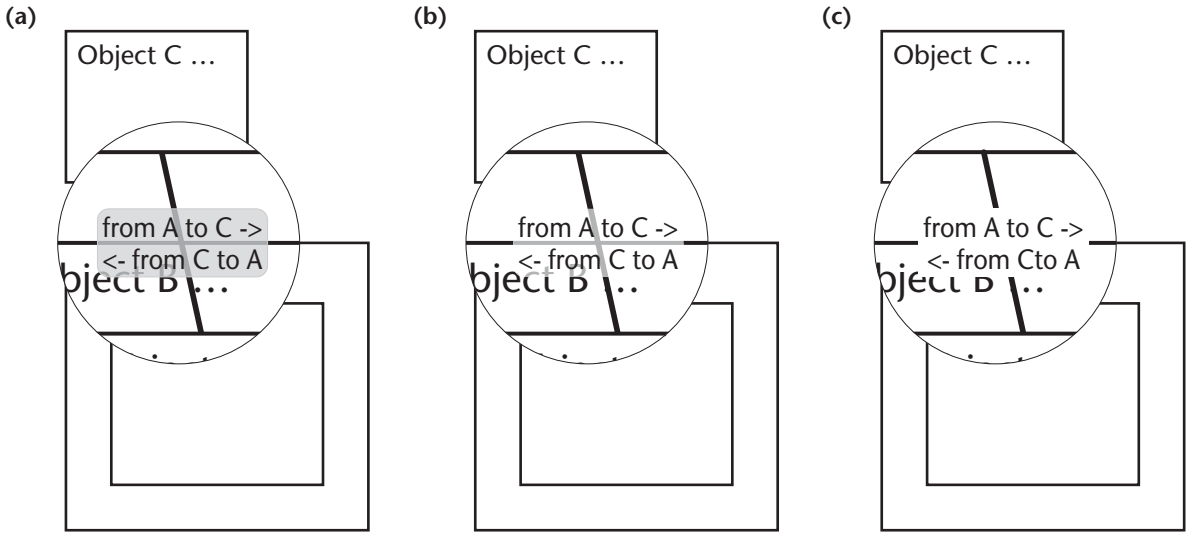


Abbildung 74: *Transparenztechniken wie in (a) und (b) zur Beschriftung von Linien ermöglichen, dass ansonsten verdeckte Bereiche (siehe (c)) erkennbar bleiben. Navigationsaktivitäten, die nur dazu dienen würden, diese Bereiche sichtbar zu machen, werden so weitgehend überflüssig.*

Wird eine Linienbeschriftung nicht als benutzerpositionierbares Objekt angesehen, so muss sie vollständig automatisch positioniert werden, ohne dass der Benutzer diesen Vorgang beeinflussen kann. Über entsprechende Parametrisierung kann bestimmt werden, in welchem Teil der Linie (in der Mitte, im vorderen Drittel, im hinteren Drittel, etc.) die Linienbeschriftung zu platzieren ist. Soll dieser Vorgang okklusionsfrei geschehen – d.h. Linienbeschriftungen verdecken keine anderen Elemente, sofern die Linie selbst keine Überschneidungen aufweist – so verliert der Zoom-Algorithmus seine Stabilität. Nachdem verfeinert wurde, müssten ggf. alle Elemente nochmals repositioniert weden, damit keine Überdeckungen mit der Linienbeschriftung auftreten. Wird wieder vergröbert, so ist nicht immer rekonstruierbar, welcher Platz durch die u.U. nun wegfallende Linienbeschriftung zusätzlich frei wird. Linienbeschriftung werden daher nicht okklusionsfrei positioniert. Da Linienbeschriftungen i.d.R. nicht so umfangreich sind, bzw. sein sollten, dass sie andere Elemente grossenteils verdecken, hält sich der durch die Überdeckung eventuell entstehende Kontextverlust in Grenzen. Finden entsprechende Transparenztechniken Verwendung, so treten keine Verdeckungen mehr im eigenlichen Sinn auf, und das darunterliegende Objekt bleibt durchwegs erkennbar (vgl. Abb. 74a und 74b/c; siehe auch Abb. 76-78).

10.2.4 Okklusionen bei Vergröberung

Der Zoom-Algorithmus in der hier vorgestellten Form sorgt dafür, dass ein okklusionsfreies Layout bei Verfeinerungsschritten erhalten bleibt. Er sorgt auch dafür, dass die Vergröberung eines ursprünglich okklusionsfreien Layouts ebenfalls okklusionsfrei bleibt.

Wird jedoch in einer bereits verfeinert dargestellten Sicht ein Element hinzugefügt, das sich nicht innerhalb desjenigen Elements befindet, welches am detailliertesten dargestellt ist, so kann es vorkommen, dass nachfolgende Vergröberungen nicht mehr zu einer okklusionsfreien Sicht führen (siehe Beispiel in Abb. 75).

Es gibt drei Möglichkeiten, mit diesem Problem – Okklusion bei Vergröberung – umzugehen:

(1) Vermeiden der Situation. Dies kann dadurch erfolgen, dass eingeschachtelte Elemente nur demjenigen Element hinzugefügt werden können, welches aktuell am Detailliertesten dargestellt ist.

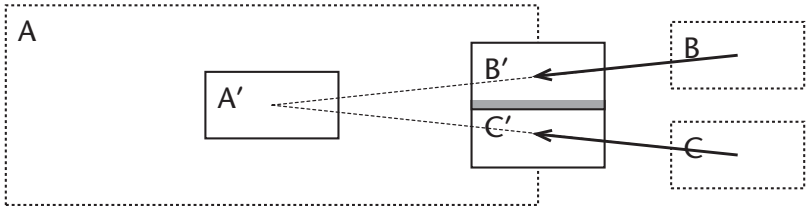


Abbildung 75: *Beispiel für eine Okklusion, die nach einer Vergröberungen auftreten kann, wenn im verfeinerten Zustand Elemente hinzugefügt wurden.*
Der grau markiert Bereich zeigt die entstehende Okklusion, die bei der Vergröberung von A auftritt, wenn die Objekte B und C hinzugefügt werden, während das Objekt verfeinert dargestellt ist.

(2) Erkennen und Auflösen der Okklusion; wobei Auflösen entweder durch (a) automatische Repositionierung erfolgen kann, oder durch (b) Markieren der Okklusion bei erzwungener manueller Repositionierung vor dem nächsten Vergröberungs- bzw. Verfeinerungsschritt.

(3) Duldung der Okklusion.

Möglichkeit 1 – Vermeidung – schränkt die Freiheit des Benutzer sehr stark ein und führt zu höherem kognitiven Ballast beim Edieren, da zusätzliche Navigationsschritte notwendig werden, wenn an besagten Stellen Elemente eingefügt werden sollen. Diese Möglichkeit wird daher als wenig praktikabel angesehen. Möglichkeit 3 – Duldung – schränkt die Freiheit des Benutzer nicht ein, führt aber beim genaueren Hinschauen ebenfalls zu erhöhtem kognitiven Ballast, da langfristig Sichten geduldet werden, die Okklusionen aufweisen. Über kurz oder lang hat dies Kontextverlust zur Folge, wenn besagte Okklusionen nicht konsequent beseitigt werden. Die (berechtigte) Bequemlichkeit des Benutzers kann dazu führen, dass derartige Okklusionen lange bestehen bleiben, bevor sie durch manuelles Repositionieren der betroffenen Elemente beseitigt werden.

Es bleibt die Möglichkeit 2 – Erkennen und Auflösen. Erkennen lässt sich eine solche Situation, indem geprüft wird, ob eine okklusionsfreie Darstellung möglich ist, wenn alle Elemente auf der Stufe des neu eingefügten Elements so vergröbert werden, dass keine weiteren Komponenten sichtbar sind.

Bleibt noch die Frage, ob automatisch oder manuell repositioniert werden soll. Eine automatische Repositionierung ist sicherlich bequemer als eine erzwungene manuelle Repositionierung, birgt aber die Gefahr, dass sukzessiv die Sekundärnotation zerstört wird, wenn der Benutzer die automatische Repositionierung nicht mitbekommt oder unreflektiert akzeptiert. Ferner ist es nicht immer auf den ersten Blick ersichtlich, warum ein Element partout nicht an bestimmten Stellen platziert werden kann. Eine erzwungene manuelle Repositionierung ist zwar deutlich unbequem(er), hält den Benutzer jedoch dazu an, die neue Position des Elements zu reflektieren.

11 ANWENDUNG DES KONZEPTS FÜR ADORA

Am Beispiel des Taschenrechners sollen nun Verfeinerungs- und Vergröberungsschritte illustriert werden, um zu zeigen, wie die Projektionstechnik im Zusammenhang einer Sequenz von Verfeinerungen und Vergröberungen arbeitet (siehe nachfolgende Abb. 76 - 78). Weitere Beispiele finden sich auch in [Be+98a] [Be+98b].

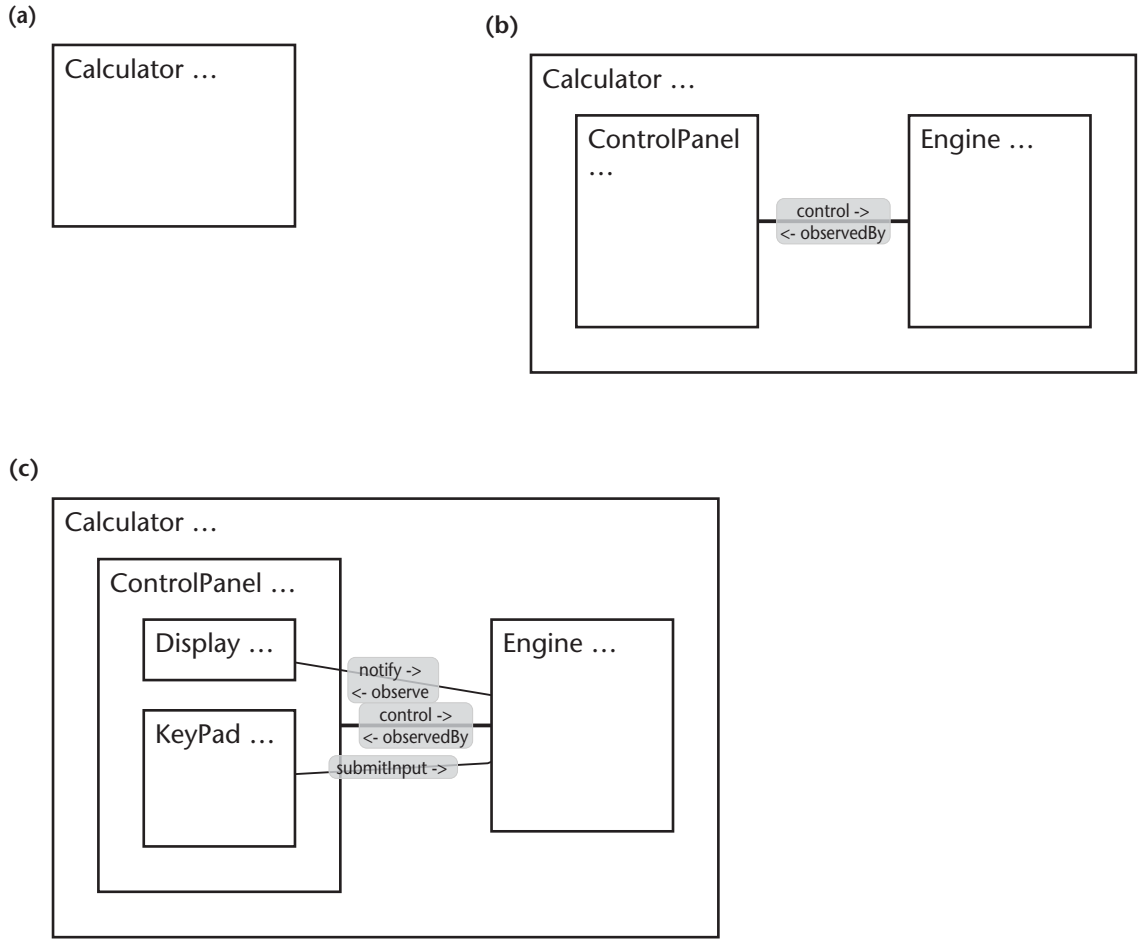
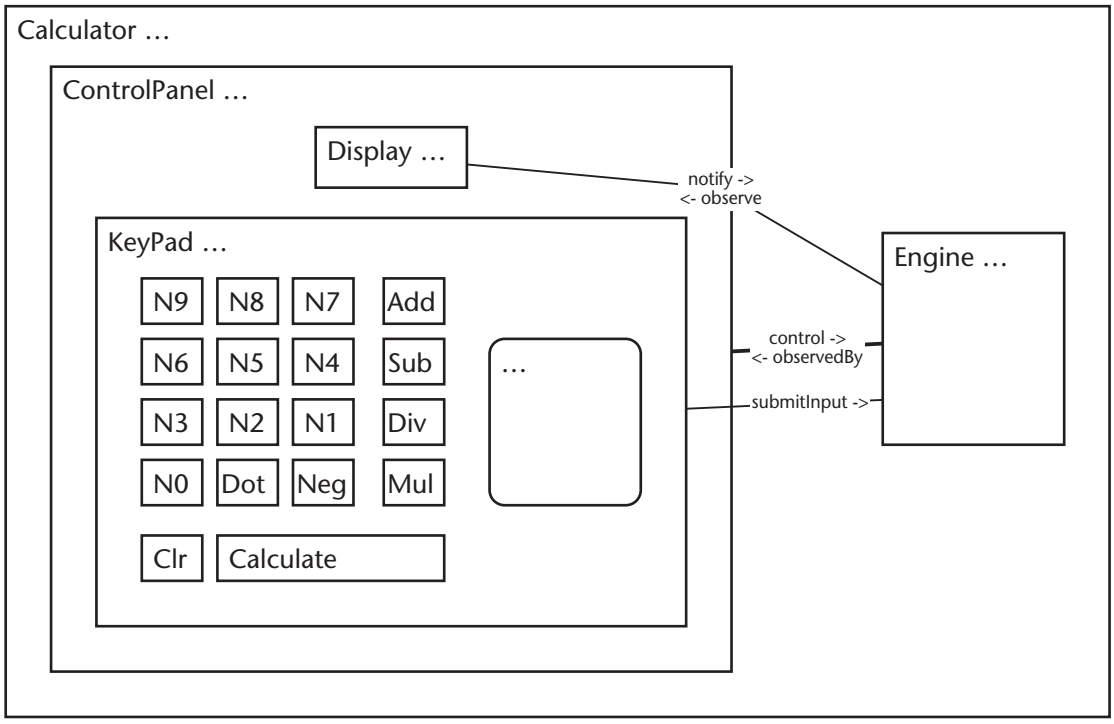


Abbildung 76: *(a) Ausgangslage, DE=0 und DT=1. (b) Verfeinerungsschritt, die Komponenten des Taschenrechners (ControlPanel und Engine) werden sichtbar. (c) Verfeinerung des Bedienfelds (ControlPanel); Anzeige (Display) und Tastatur (KeyPad) werden sichtbar. Zur Vermeidung von Überdeckungen (Beziehungsnamen in (c)) wird eine Einpunktstrategie angewandt.*

Bis zur in Abb. 76c gezeigten Verfeinerung ist für das Modell noch kein Verhalten sichtbar, d.h keine Elemente der verhaltensorientierten Einblendung wie Zustände, Zustandsübergänge, etc. sind sichtbar. Dies ändert sich, wenn eines der Objekte KeyPad, Display oder Engine verfeinert wird. Abb. 77d zeigt das Modell des Taschenrechners, wobei die Tastatur (KeyPad) nun detailliert dargestellt ist, so dass der Zustandsautomat, welcher das Verhalten beschreibt, sichtbar ist. Anstatt einer weiteren Verfeinerung wird in Abb. 77e die Sicht gezeigt, die entsteht, wenn die verhaltensorientierte Einblendung nicht mehr angezeigt wird. Dies entspricht einer

(d)



(e)

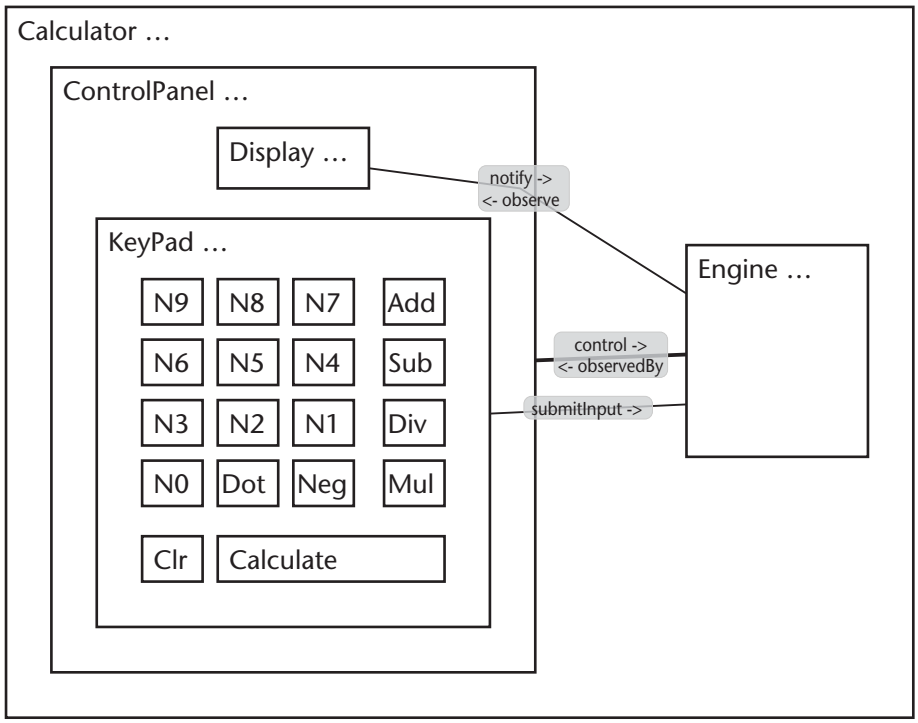


Abbildung 77: *(d) Verfeinerung der Tastatur (KeyPad), die Tasten der Tastatur (N0, N1, etc.) und ihre Verhaltensbeschreibung werden sichtbar. (e) Verkürzung auf Schemaebene, verhaltensorientierte Aspekte (Zustände, Zustandsübergänge etc.) werden ausgeblendet.*

vertikalen Verkürzung auf Schemaebene (siehe Kapitel 6.1, Grundbegriffe). Demzufolge wird in Abb. 77f auch die Verhaltensbeschreibung der Engine nicht angezeigt und nur die beiden Komponenten der Engine sind sichtbar.

Abb. 77g zeigt das Modell, nachdem Bedienfeld (ControlPanel) und Rechenwerk (Engine) wieder vergröbert wurden. Da sich auf dieser Ebene keine Verhaltenbeschreibung befindet, ist die Sicht identisch zur Sicht in Abb. 76b. Andernfalls, d.h. wenn sich dort eine Verhaltensbeschreibung befinden würde, wären die Sichten nicht identisch, da die unterliegende Projektionstechnik nach einer vertikalen Verkürzung auf Schemaebene natürlich nicht mehr stabil arbeitet.

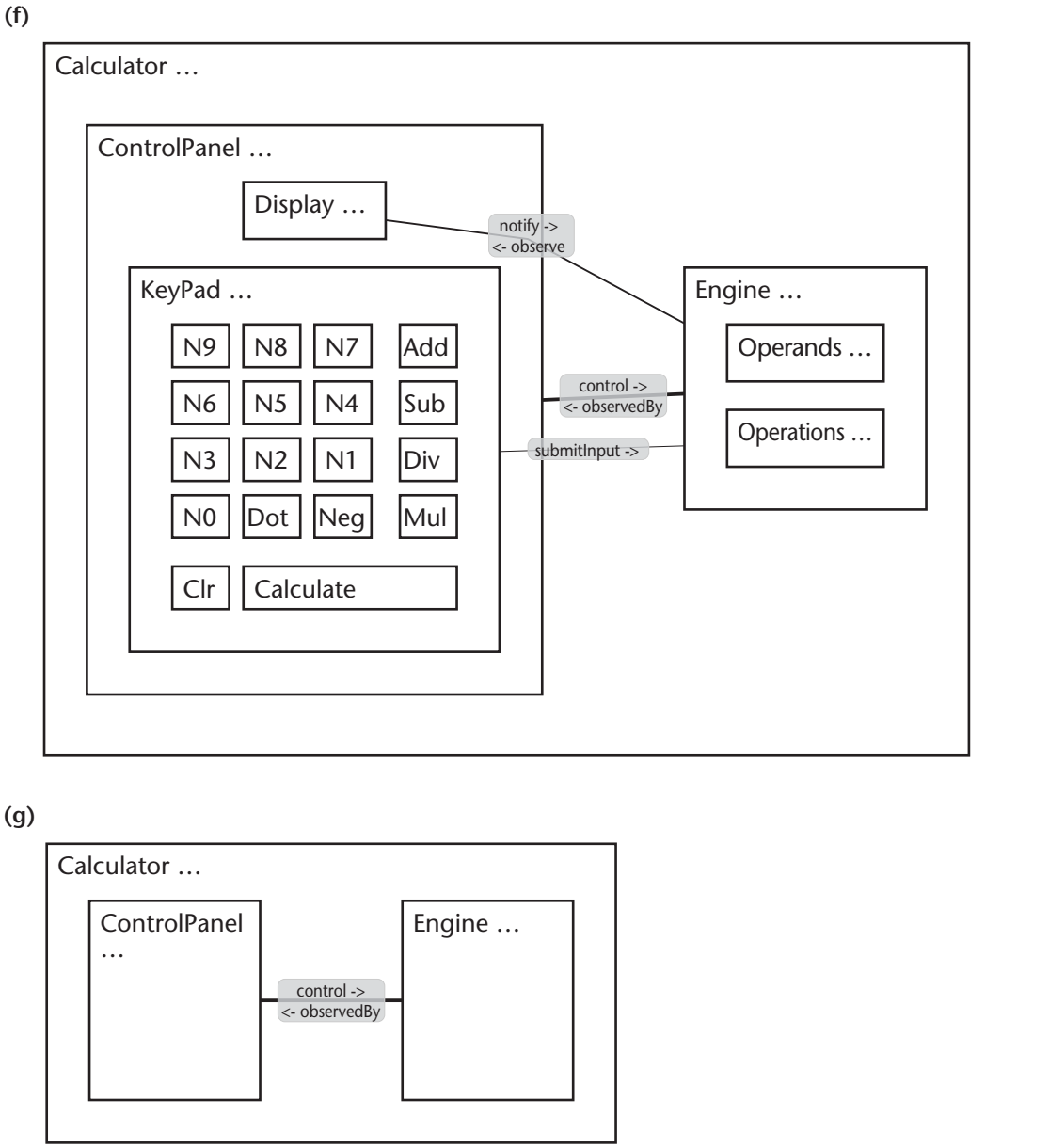


Abbildung 78: *(e) Verfeinerung der Engine. Da die Elemente der verhaltensorientierten Einblendung nicht angezeigt werden (siehe Text und Abb. 77e), wird auch für die Engine kein Automat angezeigt. (f) zeigt die Sicht, nachdem ControlPanel und Engine wieder vergröbert wurden. (f) ist somit identisch zur Sicht in Abb. 76b.*

12 ZUSAMMENFASSUNG & DISKUSSION

Die Diskussion der existierenden Visualisierungskonzepte – 1-Fensterkonzept, 2-Fensterkonzept und Fischaugenkonzept – hat die grundsätzlichen Vorteile des Fischaugenkonzepts bezogen auf die Orientierungsmöglichkeiten erläutert (siehe Kapitel 8, insbesondere Kapitel 8.1 auf S. 91). Neben den Vorteilen dieses Konzepts wurden auch die Problembereiche angesprochen, die insbesondere bei Projektionstechniken mit In-Situ-Vergrösserung und Mehrfachfoki zu den bekannten Darstellungsanomalien führen. Aufbauend darauf wurden die Anforderungen an ein Visualisierungskonzept für ADORA-Modelle dargelegt und hierbei speziell die Wichtigkeit der Sekundärnotation hervorgehoben. Abgeleitet von diesen Anforderungen wurde das im Rahmen dieser Arbeit entwickelte Visualisierungskonzept für ADORA-Modelle vorgestellt.

Dieses Konzept basiert im Kern auf einer hybriden Projektionstechnik, welche es erlaubt, kontexterhaltende Sichten zu erzeugen, die frei von den angeführten Darstellungsanomalien sind und gleichzeitig einer ggf. vorhandenen Sekundärnotation Rechnung tragen. Diese beiden Faktoren machen den Hauptunterschied zu existierenden Ansätzen aus. Die Projektionstechnik ist flexibel genug, um auch in Mehrfensterkonzepten anwendbar zu sein. Ferner ist die Projektionstechnik nicht nur dann anwendbar, wenn logisch über einen Selective Zoom oder Fisheye Zoom navigiert wird, sondern auch dann, wenn logisch über einen Full Zoom navigiert werden soll. Daraus resultiert eine Verringerung der Anzahl wünschenswerter Navigationsinstrumente wie auch eine vereinfachte und vereinheitlichte Bedienung eines Werkzeugs, welches diese Projektionstechnik umsetzt.

Neben den angeführten Vorteilen gibt es auch Problembereiche. Hierzu zählt neben dem Verzicht auf eine In-Situ-Vergrösserung auch das Linienproblem. Da sich die angeführten Darstellungsanomalien nicht ohne einen Verzicht auf in-situ-vergrössernde Projektion beseitigen lassen, ist dies eher als prinzipbedingt anzusehen und kein echter Problembereich. Zur Lösung des Linienproblems wurden einige Strategien vorgestellt. Hier liefert die Einpunktstrategie bereits gute Ergebnisse, so dass die Implementierung anderer, aufwendigerer Strategien – insbesondere auch im Hinblick auf die Erhaltung der Sekundärnotation – fragwürdig erscheint. Um diese Frage abschliessend zu klären, braucht es verlässliche Annahmen darüber, wie stark der eigentliche Verlauf einer Linie ‘gestört’ werden darf, ohne dass die Sekundärnotation zerstört wird. Generell ist die Frage ‘Wie oder wie stark darf die Sekundärnotation gestört werden, damit sie noch als eigentliche Sekundärnotation von Nutzen ist’ sehr interessant, wurde aber im Rahmen dieser Arbeit nicht angegangen.

An die zu visualisierenden Modelle stellt der hier vorgestellte Ansatz generell einige Anforderungen. Er geht davon aus und wurde speziell darauf abgestimmt, dass Abstraktionen in Form hierarchischer Zerlegungen bereits im Modell enthalten und modelliert sind. Das Modell enthält also bereits explizit diese Informationen und eben genau diese Informationen werden interpretiert, um die entsprechenden Sichten modellgetrieben zu erzeugen. Ist dies nicht der Fall, d.h. ist das Modell flach bzw. hierarchiefrei, so kann der hier vorgestellte Ansatz nicht sinnvoll angewandt werden. Die generierten Sichten ‘degenerieren’ dann in Richtung eines 1-

Fensterkonzepts mit allen damit verbundenen Konsequenzen (vgl. Kapitel 8.2), so dass – vorausgesetzt ein Fischaugenkonzept soll überhaupt noch verwendet werden – andere, auf geometrischen Projektionen beruhende Ansätze wie beispielsweise der von [Sar+92] u.U. bessere Resultate bringen. Bei ADORA-Modellen kann sicherlich davon ausgegangen werden, dass Abstraktionen explizit modelliert sind oder es zumindest sein sollten – ist dies nicht der Fall, wurde ein ganz wesentliches Ausdrucksmittels der Sprache nicht benutzt. Sollte der in dieser Arbeit vorgestellte Ansatz zur Visualisierung anderer Modelle verwendet werden, muss im Einzelfall geprüft werden, ob und inwieweit (hierarchische) Strukturen vorhanden sind, die sich sinnvoll für die Generierung von Sichten verwenden lassen.

Eine explizite Validierung des Ansatzes wurde bislang nicht durchgeführt. Dies ist in erster Linie darin begründet, dass zum Zeitpunkt des Abschlusses der Arbeiten am Visualisierungskonzept gerade erst die Validierung der Sprache ADORA abgeschlossen war (siehe [Be+99b]). In dieser Validierung wurden die ADORA-Modelle zwar in der hier vorgestellten Form präsentiert. Gegenstand der Validierung war jedoch – wie gesagt – die Sprache und nicht das Visualisierungskonzept. Analogieschlüsse zu Evaluierungen anderer perspektivbasierter Konzepte wie [Chi+91] und [Hol+89] legen jedoch die Vermutung nahe, dass das vorgestellte Visualisierungskonzept für ADORA-Modelle trägt, sofern die getroffene Grundannahme stimmt. Prinzipbedingt ist diese Annahme bei allen perspektivbasierten Ansätzen gleich. Hierfür sei abermals Furnas zitiert:

> *… First, one typical reason people examine a structure is that they are interested in some particular detail. At the same time, they need context, i.e., some sense of global structure, and where within that structure their current focus resides. The idea is therefore to present detailed local regions, but to present selected important parts of the global structure as well … The psychological feasibility remains to be explored, but there is suggestive evidence. For example, studies of editing patterns have shown that the likelihood that a person will look at or jump to some other region falls off gradually with how far away it is [Alle82]. Seen in this light the idea is to match the display of the information to this behavior pattern and, since people have limited viewing capacity, give them access to nearby parts most easily, by showing them in full detail, and gradually less detail (only more global structure) at points further away … [Furn81]*

ADORA Werkzeugumgebung

Nachdem nun das Visualisierungskonzept vorgestellt ist, wird in diesem Teil ein Werkzeug behandelt, welches dieses Konzept umsetzt.

Kapitel 13 gibt einen kurzen konzeptionellen Überblick. Neben der Grobarchitektur des Werkzeugs wird einerseits eine Übersicht darüber vermittelt, wie das Werkzeug konfiguriert wird, und andererseits angeschnitten, wie der Bereich Modellintegrität angegangen wird.

In Kapitel 14 wird detailliert darauf eingegangen, wie die Konfiguration des Werkzeugs tatsächlich abläuft, d.h. was mittels welcher Notation konfiguriert werden kann, wie die Arbeit mit dem Konfigurationsskript abläuft und wo bei Erweiterungen wie implementiert werden muss.

Kapitel 15 behandelt dann eingehend den Bereich Modellintegrität. Es wird beschrieben, wie Integritätsbedingungen für ADORA-Modelle formuliert werden, wie das Zusammenspiel dieser Bedingungen mit der Konfiguration des Werkzeugs abläuft und wie diese Bedingungen automatisch überprüft werden können.

Der Teil zur Werkzeugumgebung endet in Kapitel 16 mit einer abschliessenden Diskussion und Zusammenfassung.

13 KONZEPTIONELLER ÜBERBLICK

Die ADORA-Werkzeugumgebung besteht derzeit aus einem grafischen Editor zur Erstellung und Bearbeitung von ADORA-L-Modellen und einem Übersetzer für Integritätsbedingungen, welcher zur Konfiguration dieses Editors benutzt wird. Vollständigkeitshalber erwähnt sei, dass ein Animator/Simulator geplant ist, dessen Konzeption allerdings nicht Gegenstand dieser Arbeit ist.

Hauptthema des Werkzeugteils, d.h. der Kapitel 13-16, ist die Konfiguration des grafischen Editors. Bevor hierauf eingegangen wird (siehe Kapitel 13.2ff.), soll ein kurzer Überblick über die Umsetzung des Visualisierungskonzepts und die Grobarchitektur des Editors gegeben werden (siehe Kapitel 13.1). Hier wird u.a. angesprochen, wie Sichten repräsentiert werden und was konzeptionell passiert, wenn eine Sicht ediert wird.

13.1 Grobarchitektur

Die Modellelemente sind Objekte der Klasse BasicElement und werden von einem Objekt der Klasse AdoraModel verwaltet. Diese ermöglicht das Hinzufügen, Löschen, etc. von Modellelementen und gibt ausserdem Auskunft darüber, welche Elemente sich im Modell befinden und wie der Zustand des Modells ist, z.B. welche Elemente momentan verfeinert sind. Die Ablage eines Modells ist keine Kernaufgabe der Klasse AdoraModel[1]. Sie ist aber dafür zuständig, diesen Prozess anzustossen.

Beide Klassen – die Klasse BasicElement und die Klasse AdoraModel – sind Unterklassen der Klasse Observable und damit über einen Observer beobachtbar, wobei das AdoraModel bereits als Observer für die Modellelemente fungiert. Komponenten, welche für die Darstellung zuständig sind, arbeiten ebenfalls als Beobachter (siehe [Gam+95]) für Objekte der Klasse AdoraModel und/oder BasicElement.

Die Beobachterhierarchie ist zweistufig mit folgenden Verantwortlichkeiten ausgelegt: Ein Modellelement ist – dem Beobachtermuster folgend – bei Änderungen verpflichtet, alle registrierten Observer und somit auch das AdoraModel zu benachrichtigen. Das AdoraModel entscheidet letztendlich, inwieweit eine Änderung eines Modellelements weitere Auswirkungen hat. Wird ein Objekt z.B. verschoben, besteht für das AdoraModel kein Handlungsbedarf. Dies kann direkt von der View des entsprechenden Elements gehandhabt werden. Wird ein Objekt verfeinert, so müssen Elemente repositioniert werden. Das AdoraModel ist nun dafür zuständig, diese Aktualisierung bei der LayoutEngine anzustossen. Wird beispielsweise der Name eines Objekts geändert, so muss das AdoraModel u.U. eine Überprüfung beim AdoraModelChecker einleiten, der dann seinerseits sicherstellt, dass der neue Name zulässig ist.

---

[1] Derzeit wird auch diese Aufgabe von der Klasse AdoraModel übernommen und hierfür der Serialisierungsmechanismus von Java benutzt. Sollen hier andere Mechanismen – wie z.B. eine SQL-Datenbank oder ein XML-basierte Repository – zum tragen kommen, dient die Klasse als Wrapper.

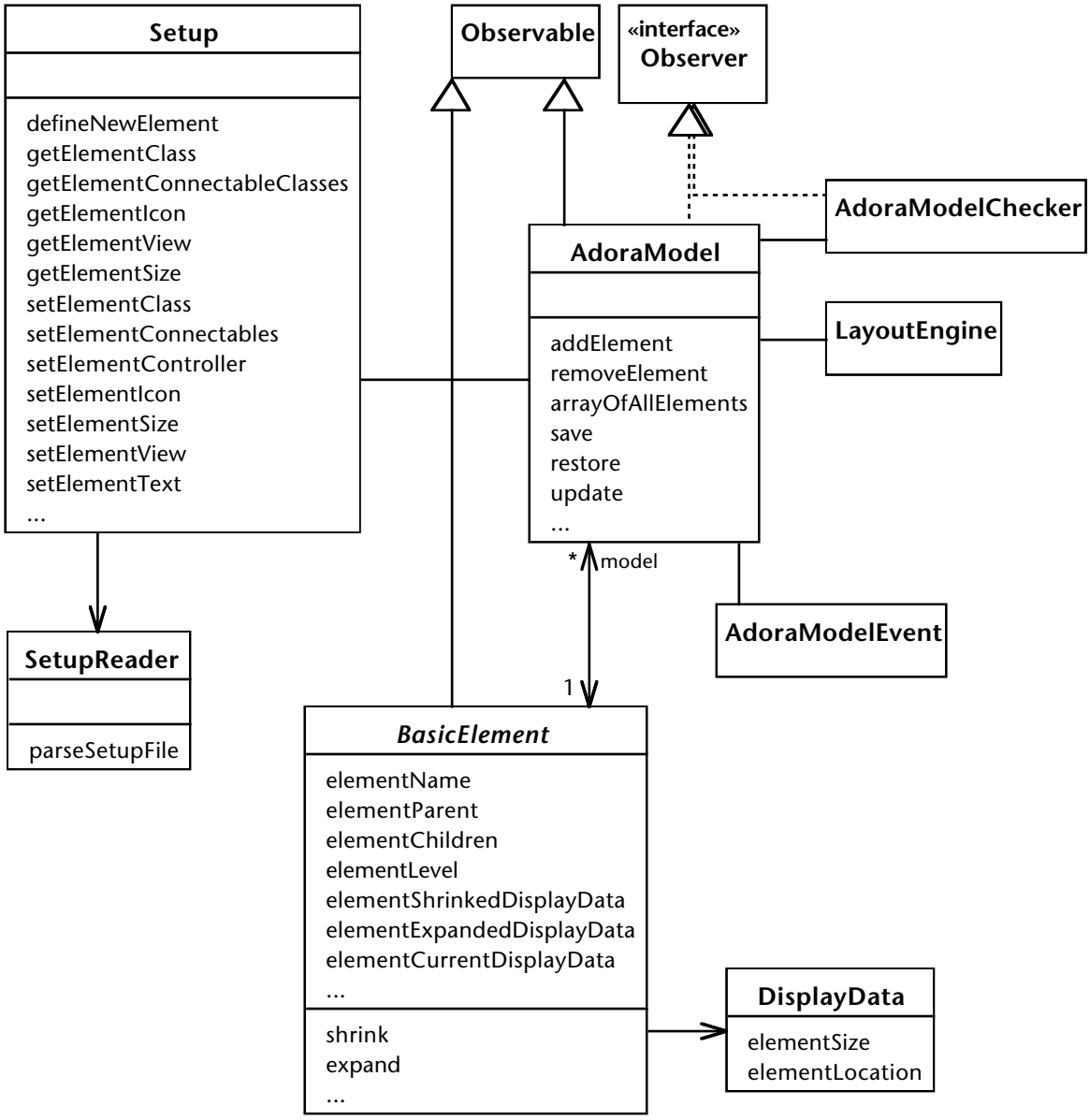


Abbildung 79: *Grobarchitektur der Modellkomponente des Werkzeugs*

Das AdoraModel kann hierbei über zwei Mechanismen delegieren: Zum einen *direkt* und zum anderen *mittels Benachrichtigung aller Beobachter.* Bei Aktivitäten, die synchronen Charakter haben und die Beobachter erst benachrichtigt werden sollen, wenn die Aktivität abgeschlossen ist, wird direkt delegiert. Eine solche Aktivität ist z.B. das Anpassen des Layouts nach einem Verfeinerungsschritt. Bei Aktivitäten, die asynchronen Charakter haben, werden über den Benachrichtigungsmechanismus des Observables die Observer benachrichtigt, die dann entsprechend reagieren können. Eine solche Aktivität ist z.B. das Überprüfen von Integritätsbedingungen. Zur Kommunikation dienen semantische Ereignisse, sogenannte ADORA-Model-Events (kurz AMEV), die über Objekte der Klasse AdoraModelEvent repräsentiert werden.

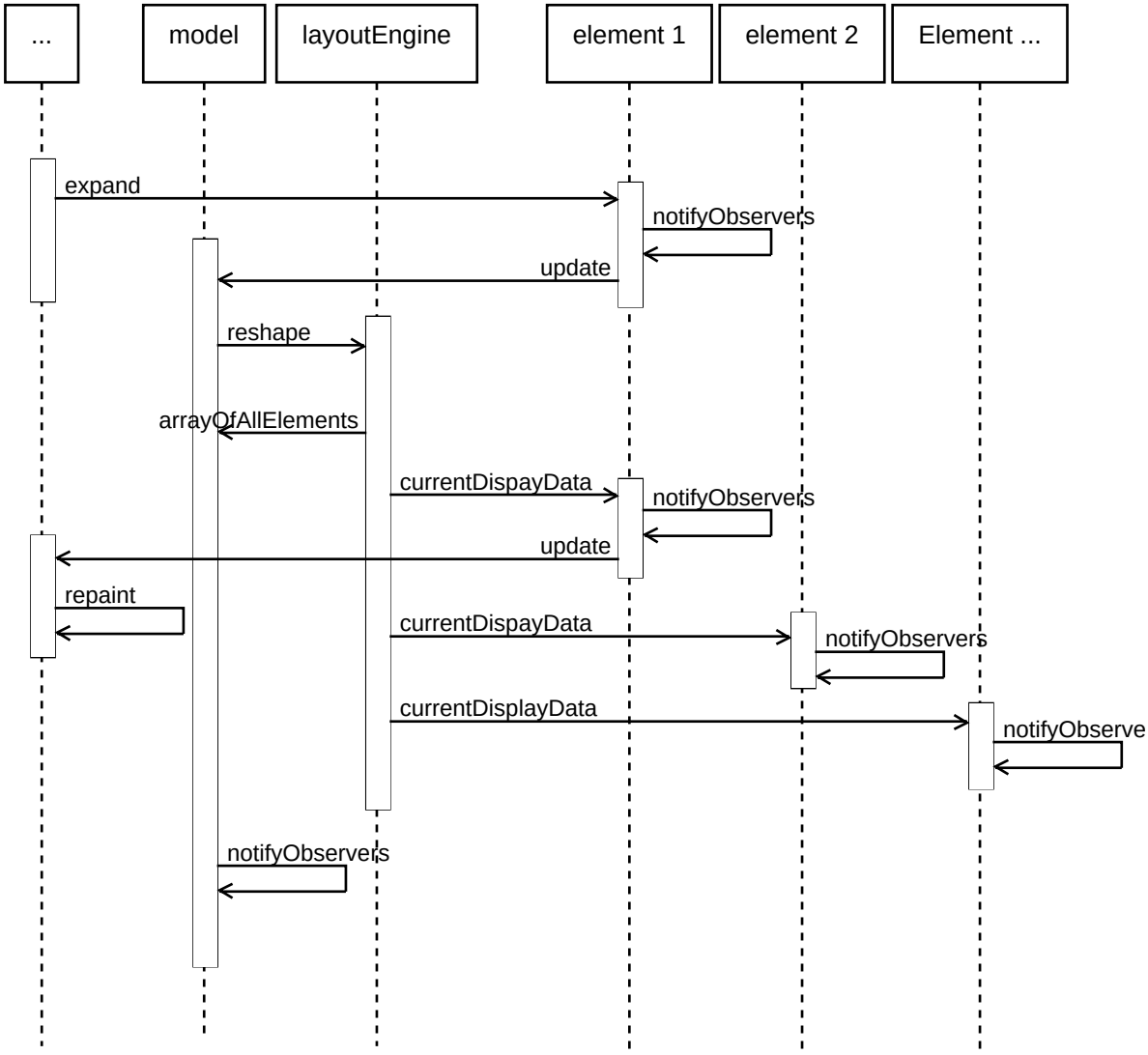


Abbildung 80: *Nachrichtenfluss bei einem Verfeinerungsschritt*

Während die physikalische Navigation mittels Scrollbars vollständig von den Darstellungskomponenten zu übernehmen ist[1], ist die Möglichkeit zur logischen Navigation in die Klasse AdoraModel integriert.

Das Werkzeug – so wie es hier konzipiert wurde – erlaubt entsprechend dem Visualisierungskonzept lediglich eine Sicht auf das Modell. Die aktuelle Position und Grösse des Objekts in dieser Sicht wird über ein Objekt der Klasse DisplayData beim Modellelement vermerkt und wird auch mit abgespeichert, so dass über Objekte der Klasse AdoraModel nicht nur das Modell, sondern auch die aktuelle logische Sicht repräsentiert wird.

[1] Entsprechende Widgets sind bereits in Java vorhanden, so dass diese Funktionalität über eine einfache Dekoration (siehe [Gam+95]) eines Anzeigepanels hergestellt werden kann.

Sollen mehrere Sichten möglich sein, so müssen diese Information entweder ausgelagert werden oder es muss zusätzlich vermerkt werden, für welche Sicht diese Informationen gültig sind. Die LayoutEngine muss dann ferner in der Lage sein, bei einer Verfeinerung oder Vergröberung ausschliesslich die Anzeigedaten für die entsprechende Sicht zu ändern, bzw. zwischen AdoraModel und LayoutEngine müss(te) noch ein entsprechender Wrapper gelegt werden, welcher eine Sicht repräsentiert.

Im Zusammenhang mit perspektivbasierter Betrachtungsweise sind Mehrsichtvarianten zwar grundsätzlich möglich, aber generell nicht ganz unproblematisch. Wird in einer Sicht ediert, so hat dies u.U. Auswirkungen auf andere Sichten. Wird beispielsweise in einer Sicht ein Komponente zu einer Komposition hinzugefügt, so müssen die Anzeigedaten in allen denjenigen Sichten aktualisiert werden, in welchen die entsprechende Komponente ebenfalls in verfeinerter Form sichtbar ist. Neben der Tatsache, dass zusätzliche Sichten nicht den gleichen Nutzen bringen wie beim 1- oder 2-Fensterkonzept, ist dieser zusätzliche Aufwand einer der Gründe dafür, weshalb hier lediglich eine 1-Sichtvariante verfolgt wurde.

13.2 Konfiguration des Werkzeugs

Der Sprachkern von ADORA-L ist zwar recht stabil (siehe Kapitel 5 sowie [Joos99]), dennoch ist ADORA-L eine Sprache, deren Definition derzeit nicht vollständig abgeschlossen ist. Bestehende Sprachkonstrukte können sich noch ändern und es können natürlich neue Konstrukte hinzukommen. Die zwischenzeitlich erfolgte Integration von Szenarien und Use Cases in die Sprache ist ein Beispiel hierfür. Bei der Konzeption des Werkzeugs konnte daher nicht von einer fixen Menge von Modellelementen ausgegangen werden, die sich nicht mehr oder nur noch geringfügig ändern.

Das Werkzeug muss immer wieder aktualisiert und angepasst werden (können), um mit der geplanten Evolution der Sprache Schritt zu halten. Eine solche Anpassung der Werkzeugumgebung muss möglichst einfach von statten gehen. Beispielsweise sollte der Kern der Steuerungslogik der einzelnen Werkzeugteile unangetastet bleiben können und auch nicht betrachtet werden müssen, wenn lediglich die Gestalt eines bestehenden Sprachelements verändert wird oder ein neues Element hinzukommt.

Natürlich können nicht sämtliche möglichen Änderungen oder Erweiterungen vorhergesehen werden, aber ganz bestimmte Änderungen werden immer vorkommen. Ziel ist es, die vorhersehbaren und mit aller Wahrscheinlichkeit auch häufig auftretenden Änderungen möglichst einfach durchführen zu können. Zu diesem Zweck ist das Werkzeug konfigurierbar. Konfigurierbarkeit in diesem Zusammenhang bedeutet zweierlei:

(1) *Änderung der vom Werkzeug verarbeiteten Sprachkonstrukte*; d.h. ohne den eigentlichen Quell-Code des Werkzeugs zu ändern, kann festgelegt werden, welche Sprachkonstrukte unterstützt werden, d.h. aus welchen Elementen ein Modell konkret besteht und wie diese Elemente zusammenarbeiten. Beispielsweise von wo nach wo eine Beziehung, ein Zustandsübergang, etc. angelegt werden kann oder ob

bestimmte Elemente andere Elemente enthalten können (siehe abstaktes Objekt, komplexer Zustand, etc.). Wird die konkrete und abstrakte Syntax der Sprache bzw. ihrer Konstrukte geändert oder ergänzt, so kann diese Änderung im Werkzeug nachvollzogen werden, ohne dass die Kernapplikationslogik des Werkzeugs angetastet werden muss.

(2) *Überprüfung der Korrektheit der Modelle*, d.h. es können Integritätsbedingungen formuliert oder anders ausgedrückt Invarianten angegeben werden, die für ein Modell immer gelten müssen, wenn dieses korrekt sein soll. Bestehende Invarianten können ergänzt und neue können hinzugefügt werden, ohne dass der eigentliche Code des Werkzeugs geändert werden muss. Ein Modell kann vom Werkzeug automatisch auf Einhaltung der Invarianten geprüft werden. Folgerichtig kann der Teil der Semantik der Sprachkonstrukte bzw. eines Modells beschrieben werden, welcher für das Werkzeug relevant ist, ohne dass die Kernapplikationslogik des Werkzeugs angetastet werden muss.

Das Werkzeug wird konfiguriert, indem die konkrete und abstrakte Syntax der Sprachelemente sowie deren (werkzeugrelevante) Semantik auf eine bestimmte Art und Weise festgelegt wird.

Im einzelnen geschieht dies folgendermassen:

Konkrete Syntax, Darstellung der Sprachkonstrukte

Erstellung von Darstellungsobjekten: Um die konkrete Syntax von Sprachkonstrukten festzulegen, muss programmiert werden. Genauer gesagt, es müssen bestehende Klassen entsprechend spezialisiert werden. In dieser Spezialisierung wird über die Redefinition einer bestimmten Methode (siehe Schablonenmethode [Gam+95]) das Aussehen der Elemente, also deren konkrete Syntax festgelegt. Je nachdem, welche Basisklasse für die Darstellung eines Sprachkonstrukts gewählt wird, sind bestimmte Eigenschaften und ein bestimmtes Aussehen bereits vorgegeben. Optional kann auch das Verhalten der Modellelemente bei Benutzerinteraktionen angepasst werden, z.B. wie der Benutzer den Namen oder die Grösse eines Objekts ändern kann, etc.

Abstrakte Syntax, Struktur der Sprachkonstrukte

Erstellung von Strukturobjekten: Um die abstrakte Syntax von Modellelementen festzulegen, muss – analog zur konkreten Syntax – ein wenig programmiert werden, d.h. es muss eine Klasse erstellt werden, welche die Struktur des Sprachkonstrukts repräsentiert. Analog zur Erstellung der Darstellungsobjekte wird auch hier von Basisklassen ausgegangen, die über das Entwurfsmuster ‘Schablonenmethode’ entsprechend angepasst werden.

Ausserdem muss mittels einer speziellen Konfigurationsnotation beschrieben werden, wie die Elemente zusammenarbeiten. Dieses Konfigurationsskript wird beim Start des Werkzeugs geparst, ausgewertet und entsprechend abgelegt. Genauer gesagt wird in diesem Skript festgelegt, aus welchen Sprachkonstrukten der Editor besteht, was dies für Sprachkonstrukte sind und wie die syntaktischen Bildungsregeln für korrekte ADORA-Modelle sind. Ferner wird in

diesem Skript beschrieben, welches Strukturobjekt mittels welchem Darstellungsobjekt visualisiert werden soll.

Semantik, Integrität des Modells

Die Semantik der Sprachkonstrukte wird auf prädikatenlogischer Basis festgelegt. Die Notation, welche hierfür zur Verfügung steht – ADORA-IB genannt – erlaubt es, für jedes Sprachkonstrukt prädikatenlogische Ausdrücke zu formulieren, die dann von jeder Ausprägung dieses Konstrukts eingehalten werden müssen.

In ADORA-IB formulierte Bedingungen können mittels eines entsprechenden Übersetzers in Java übersetzt und (nach erfolgter Übersetzung des Java-Quellcodes in ByteCode) automatisch überprüft werden. Es ist also nicht im Werkzeug unveränderlich festgelegt, was ein korrektes Modell ist und was nicht.

Wie der Editor konfiguriert wird, d.h. wie die konkrete und abstrakte Syntax sowie die Semantik beschrieben werden, wird in den nächsten Kapiteln ausführlicher angesprochen. Begleitend wird an einem kleinen Beispiel die praktische Umsetzung illustriert. Als Beispiel dient der denkbar einfachste grafische Editor für ADORA-L. Dieser Editor besteht lediglich aus Objekten und Beziehungen (siehe Abb. 81).

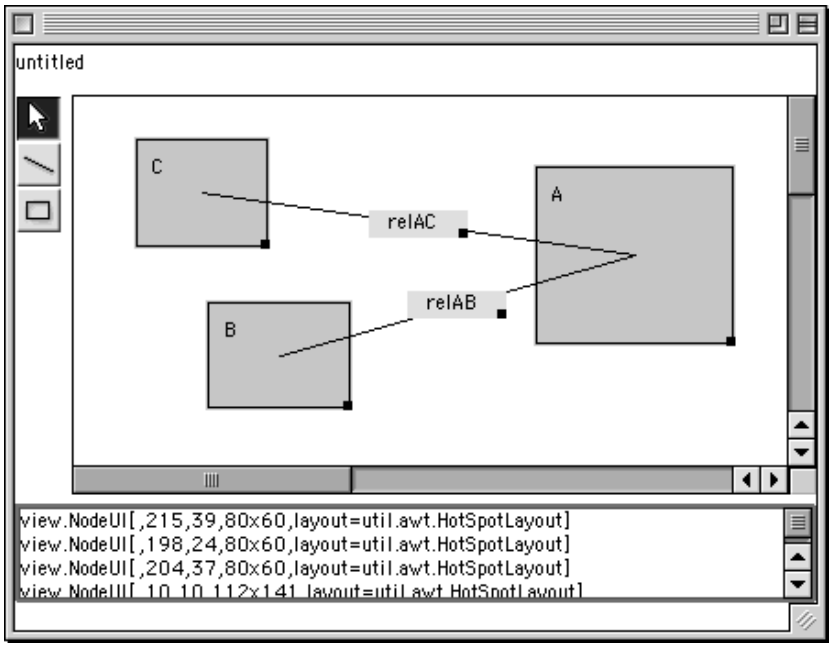


Abbildung 81: *Einfacher ADORA-Editor*

14 Konfiguration des Werkzeugs – Sprachkonstrukte

Das Werkzeug wird konfiguriert indem – dem MVC-Muster [Bus+96] folgend – für jedes Sprachkonstrukt eine Model-, View- sowie ggf. auch eine Controller Klasse erstellt und in einem Konfigurationsskript das Zusammenspiel dieser Teile angegeben wird.

Das Konfigurationsskript legt hierbei fest, welche Sprachelemente (repräsentiert durch entsprechende Model-Klassen) das Werkzeug kennt und über welche Darstellungsklassen diese dargestellt werden. Die Darstellungsobjekte folgen hierbei einem Kompositionsmuster (siehe [Gam+95], Kompositum). Über eine Schablonenmethode [Gam+95] wird die individuelle Darstellung ggf. angepasst.

Der hier verwendete Ansatz – eine Mischung aus Framework-Reuse und Skripting zu fahren – hat seine Wurzeln in Werkzeugen wie vis-A-vis [Lic+93] oder Kogge [Koe+96]. Wie die Konfiguration des Werkzeugs abläuft, wird nachfolgend exemplarisch erläutert.

14.1 Konfiguration der Struktur und Darstellung der Sprachkonstrukte

14.1.1 Abstrakte Syntax (erster Teil)

Für jedes Sprachkonstrukt, das vom Werkzeug verarbeitet werden soll, muss eine entsprechende Klasse existieren, welche die Struktur des Sprachkonstrukts repräsentiert. Diese Klasse erhält man im Regelfall über eine Spezialisierung gegebener Klassen und entsprechender Anpassung.

Basisklasse aller Elemente, die im Werkzeug verarbeitet werden können, ist die abstrakte Klasse BasicElement (siehe Abb. 82). In dieser Klasse ist bereits die Grundfunktionalität zur Einschachtelung weiterer Elemente vorhanden, aber noch nicht weiter ausgeführt. Diese wird ergänzt von zwei Schnittstellen. Grundsätzlich wird hier zwischen zwei Arten, sog. *Connectables* und *Connectives* unterschieden; wobei grundsätzlich Connectables über Connectives verbunden werden können und Connectives nur im Kontext ihrer Konstituenten, also der Connectables, die über sie verbunden sind, existent sein können.

Von der Klasse BasicElement abgeleitet sind die Klassen Node und Edge. Dies Klassen sind (normalerweise) der Ausgangspunkt, um neue Sprachkonstrukte zu schaffen. Die Klasse Node implementiert die Schnittstelle Connectable und die Klasse Edge die Schnittstelle Connective. Klassen die von Node abgeleitet werden, erben hierbei die Funktionalität, dass weitere Connectables eingeschachtelt werden können. Analog dazu erben Klassen die von Edge abgeleitet werden, die Funktionalität, dass sie Connectables verbinden können sowie die Funktionalität, dass weitere Connectives eingeschachtelt werden können.

Spezialisiert man eine gegebene Klasse, um ein neues Sprachkonstrukt zu erzeugen, so muss man sich darüber im Klaren sein, welche von diesen beiden Klassen (Node oder Edge) verwendet wird.

- Offensichtlich dienen Klassen wie Node, welche das Interface Connectable implementieren, als Basis für Objekte, Zustände, ö.ä.
- Klassen wie Edge mit Interface Connective dienen als Basis für Beziehungen, Zustandsübergänge, o.ä.

Sprachkonstrukte müssen nicht zwingend von Edge oder Node abgeleitet sein. In diesem Fall muss aber die entsprechende Funktionalität als Connectable oder Connective (oder beides) gegeben und ausprogrammiert sein.

Beispiel

Wie bereits angesprochen, soll der einfache ADORA-Editor (siehe Abb. 81) über zwei Sprachkonstrukte verfügen: abstrakte Objekte und Beziehungen. Es werden daher zwei Klassen erstellt – für jedes Sprachkonstrukt eine.

- Objekte der Klasse AbstractObject sollen abstrakte Objekte repräsentieren und demzufolge wird diese Klasse von Node abgeleitet.
- Objekte der Klasse Relation sollen Beziehungen zwischen abstrakten Objekten repräsentieren. Die Klasse Relation ist somit Unterklasse von Edge. (siehe Abb. 82).

Da es ein einfacher Editor ist, kommt in diesen Unterklassen nichts hinzu. Es gibt also keine neuen Instanz- oder Klassenvariablen und auch keine neuen Methoden. Auch wird nichts

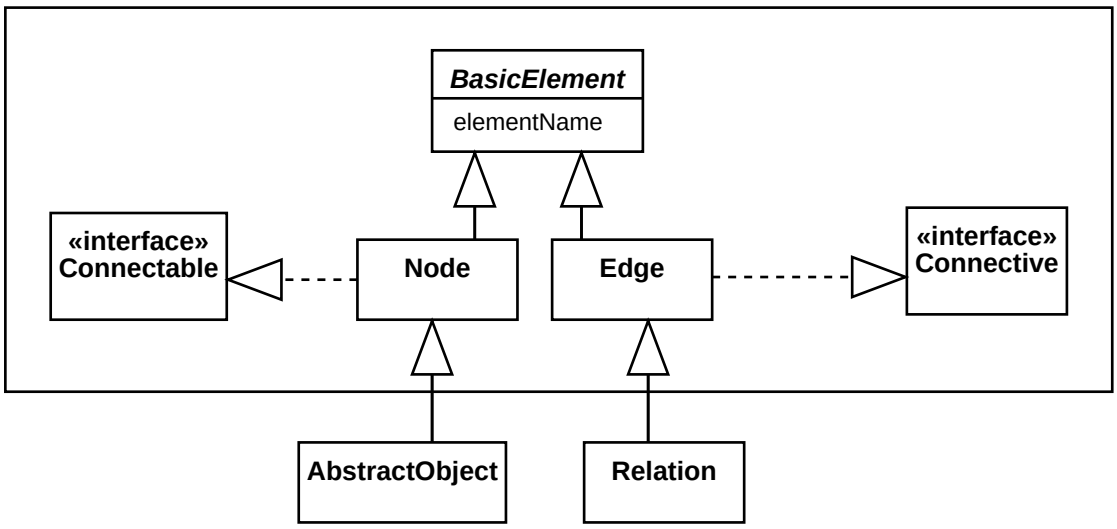


Abbildung 82: *Neue Sprachkonstrukte erhält man über die Spezialisierung vorgegebener Klassen (BasicElement, Node, Edge). Der Rahmen (oben) zeigt die Basisklassen für die Schaffung neuer Sprachkonstrukte. Die Klassen AbstractObject und Relation repräsentieren die entsprechenden Sprachelemente. Man bekommt sie über eine Spezialisierung von Node und Edge.*

Bestehendes redefiniert (was sowieso bei dieser Art von Klassen unüblich wäre, ganz im Gegensatz zu den Klassen, die für die Darstellung zuständig sind).

14.1.2 Konkrete Syntax (erster Teil)

In einem weiteren Schritt – also nachdem die Klassen existieren, welche die Struktur der Sprachkonstrukte repräsentieren (siehe vorheriges Kapitel 14.1.1) – werden die Klassen erstellt, die festlegen, wie ein Sprachkonstrukt eigentlich darzustellen ist.

Hierfür können ebenfalls Unterklassen gebildet werden und zwar von den Klassen BasicElementUI, EdgeUI oder NodeUI (siehe Abb. 84). Durch Redefinition einer Methode (draw) kann bestimmt werden, wie ein Sprachkonstrukt dargestellt wird (siehe Abb. 83).

```
// in BasicElement
abstract void draw( Graphics toDrawInto );
```

```
// in NodeUI
void draw( Graphics toDrawInto )
    {
    toDrawInto.setColor( Color.lightGray );
    toDrawInto.fillRect( 0, 0,
        getSize().width, getSize().height );
    toDrawInto.setColor( getForeground() );
    toDrawInto.drawRect( 1, 1,
        getSize().width-3, getSize().height-3 );
    toDrawInto.drawString(
        elementToObserve().toString(), 10, 20 );
    }
```

Abbildung 83: *Beispiel für die Redefinition der Schablonenmethode draw in NodeUI. Java-Quellcode für die Methode in der Klasse BasicElement (links) und in NodeUI (rechts).*

Wie gesagt, es können Unterklassen gebildet werden – müssen aber nicht. Im Gegensatz zu den Strukturklassen (siehe Kapitel 14.1.1) muss nicht für jedes Sprachkonstrukt eine eigene Darstellungsklasse vorhanden sein. Wenn die Standard-Darstellung von NodeUI und EdgeUI – Rechteck und Linie – zufriedenstellend ist, können NodeUI und/oder EdgeUI genausogut direkt und ohne weitere Spezialisierung verwendet werden.

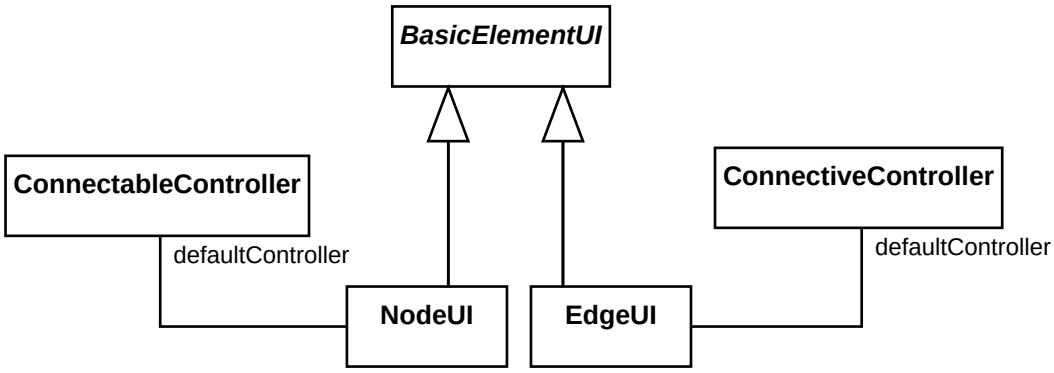


Abbildung 84: *UML-Klassendiagramm; Basisklassen für die Darstellung einzelner Modellelemente. NodeUI stellt ein Objekt der Klasse Node dar und EdgeUI ein Objekt der Klasse Edge.*

Wie zuvor bei den Strukturklassen gilt auch hier ganz analog, wenn die vorgegebenen Klassen (NodeUI, EdgeUI, etc.) nicht benutzt werden, so muss mindestens die ansonsten ererbte Funktionalität garantiert sein.

Jede *UI-Komponente benutzt als Voreinstellung einen Standard-Controller – ConnectableController für alle Klassen, die das Interface Connectable implementieren und ConnectiveController für alle Klassen, die das Interface Connective implementieren.

Soll auch die Steuerung entsprechend angepasst werden, so kann – mit den üblichen Konsequenzen – entweder ein neuer Controller erstellt oder wie gewohnt ein bereits existierender Controller spezialisiert werden. Standardmässig wird von den beiden Controllern das Erzeugen und Löschen von Elementen sowie die Grössenänderung unterstützt (siehe Abb. 85). Der ConnectableController unterstützt ausserdem die Verfeinerung bzw. Vergröberung von Elementen entsprechend dem in Teil II vorgestellten Visualisierungskonzept. Tiefer soll an dieser Stelle nicht darauf eingegangen werden. Wie angegeben werden kann, welche View-Komponente mit welchem-Controller zusammenarbeitet, wird später (siehe Kapitel 14.1.4 auf Seite 146) angesprochen.

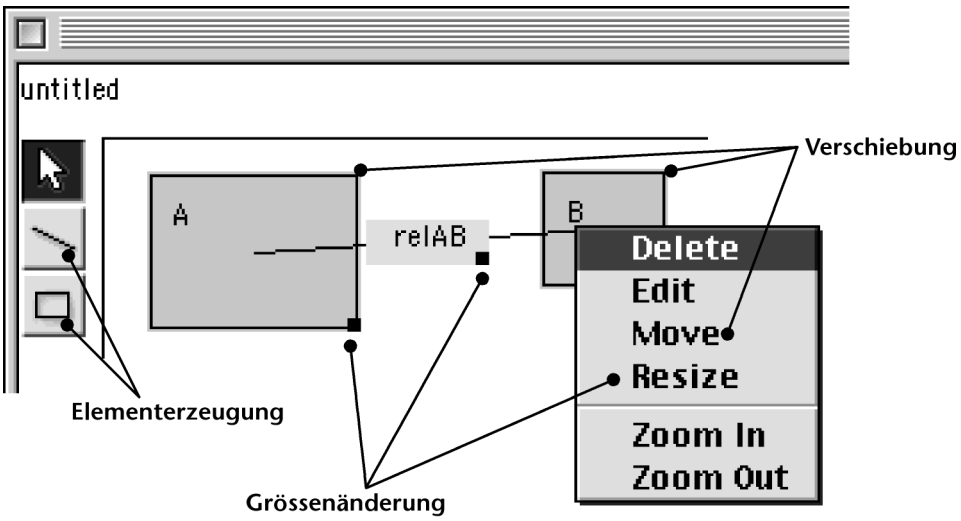


Abbildung 85: *Grundfunktionen der vorhandenen Controller*

Beispiel

Für unseren einfachen ADORA-Editor bilden wir keine Unterklassen. Die Klassen NodeUI und EdgeUI kommen unverändert zur Anwendung. Ebenso begnügen wir uns mit dem, was die beiden Standard-Controller ConnectableController und ConnectiveController an Steuerung anbieten.

Zusammengefasst werden also hier im Beispiel für das Sprachkonstrukt ‘abstraktes Objekt’ die Klassen AbstractObject, NodeUI sowie ConnectableController benutzt und für das Konstrukt ‘Beziehung’ die Klassen Relation, EdgeUI sowie ConnectiveController.

14.1.3 Abstrakte Syntax (zweiter Teil)

Nachdem für diejenigen Sprachkonstrukte, die im Werkzeug zur Verfügung stehen sollen, Model-, View- und Controller existieren, muss noch ein entsprechendes Konfigurationsskript erstellt werden.

In diesem Skript (vollständige EBNF-Grammatik siehe Anhang B) werden die Klassen, welche die Sprachkonstrukte repräsentieren, dem Werkzeug bekanntgemacht. Ferner wird festgelegt, wie diese Klassen bzw. deren Objekte (auf syntaktischer Ebene) im Werkzeug zusammenspielen. Beim Start des Werkzeugs wird das Skript gelesen, geparst, ausgewertet und entsprechend abgelegt. Das Werkzeug konfiguriert sich dann dementsprechend.

Im Konfigurationsskript wird unterschieden zwischen der Deklaration von Connectables und der Deklaration von Connectives. Für Connectables geschieht dies nach untenstehendem Schema. Zuerst wird nach dem Schlüsselwort connectable der Name angegeben, dann leitet das Schlüsselwort contains die Liste der Sprachkonstrukte ein, deren Elemente als Teil enthalten sein können. Eine Möglichkeit hierbei Kardinalitäten vorzugeben, gibt es nicht. Wird die contains-Anweisung weggelassen, so kann das Element keine weiteren Elemente enthalten.

```
connectable   <Name der Klasse, die das Sprachkonstrukt repräsentiert>          // Kopf
  contains    <Name(n) der Klasse(n), die die Sprachkonstrukte repräsentieren, deren
                Elemente enthalten sein können>
  {                                                                            // Rumpf
  … ausgespart, siehe Kapitel 14.1.4 …
  }
```

Abbildung 86: *Deklaration von Connectables*

Für Connectives geschieht die Deklaration ganz analog, jedoch unter zusätzlicher Angabe der Konstituenten. Das Schlüsselwort connects[1] leitet hierbei die Liste der Sprachkonstrukte ein, deren Elemente über den Connective verbunden werden können.

```
connective    <Name der Klasse, die das Sprachkonstrukt repräsentiert>
  contains    <Name(n) der Klasse(n), die die Sprachkonstrukte repräsentieren,
                deren Elemente enthalten sein können>
  connects    <Name(n) der Klasse(n), die die Sprachkonstrukte repräsentieren, die über den
                Connective verbunden werden können>
  {
  … ausgespart, siehe Kapitel 14.1.4 …
  }
```

Abbildung 87: *Deklaration von Connectives*

Anhand der Unterscheidung Connectable/Connective erkennt das Werkzeug, wie bei einem Verfeinerungs- oder Vergröberungsschritt mit dem entsprechenden Element umzugehen ist. Für Connectables kommt hierbei die in Kapitel 9/10 beschriebene Projektionstechnik zur Anwendung. Connectives werden entsprechend der Sichtbarkeit ihrer Connectables dargestellt oder auch nicht.

---

[1] Die Grammatik sowie die Schnittstellen im Werkzeug lassen zwar mehr als zweiwertige Beziehungen zu, jedoch werden algorithmisch derzeit nur zweiwertige Beziehungen unterstützt.

Die im Konfigurationsskript enthaltene Information über die abstrakte Syntax der Sprache dient ausserdem als Basis, um bei der Erzeugung, Änderung, etc. von Elementen sicherzustellen, dass dies eine Aktion ist, deren Resultat mit der abstrakten Syntax der Sprache verträglich ist. Sind beispielsweise keine Konstrukte deklariert (siehe contains), die eingeschachtelt sein können, so ist es werkzeugseitig auch nicht möglich, ein neues Modellelement in diesem zu erzeugen oder ein existierendes Element in dieses Element hineinzubewegen.

Beispiel

Für das Beispiel sieht das Konfigurationsskript dann folgendermassen aus:

```
connectable    model.AbstractObject
   contains    model.AbstractObject
   {
   …
   }

connective     model.Relation
   contains    model.Relation
   connects    model.AbstractObject; model.AbstractObject
   {
   …
   }
```

Abbildung 88: *Konfigurationsskript für den einfachen ADORA-Editor (noch unvollständig)*

Deklariert sind hier das abstrakte Objekt, welches weitere abstrakte Objekte (repräsentiert durch Instanzen der Klasse model.AbstractObject) enthalten kann sowie die Beziehung, welche abstrakte Objekte verbindet und ihrerseits Unterbeziehungen haben kann (vgl. Kapitel 5).

14.1.4 Konkrete Syntax (zweiter Teil)

Um die Konfigurationsinformation für das Werkzeug vollständig zu machen, muss noch angegeben werden, von Objekten welcher Klasse die Sprachelemente eigentlich dargestellt werden und welche Komponente die Reaktionen auf Benutzereingaben entsprechend koordiniert.

Hierzu wird im Rumpf angegeben, wer – d.h. Objekte welcher Klasse – im entsprechenden MVC-Muster als View (siehe Abb. 89, view) und wer als Controller (siehe Abb. 89, controller) fungiert. Ferner wird angegeben, durch welches Symbol das Sprachkonstrukt in der Symbol-Leiste des Werkzeugs repräsentiert wird (siehe Abb. 89, icon) sowie der richtige Name des Elements (siehe Abb. 89, text) und ein kurzer Hilfetext (siehe Abb. 89, help), der ggf. als Tooltip angezeigt werden kann.

Grundsätzlich wäre es auch möglich gewesen, View und Controller nicht über das Konfigurationsskript anzugeben, sondern entsprechende Standardklassen bei der Strukturklasse (welche als Model fungiert) zu erfragen.[1] Dies würde die ansonsten im Konfigurationsskript sichtbare Information, wer für Darstellung und Steuerung zuständig ist, in den Code verlagern, was inbesondere bei Änderungen nicht unbedingt wünschenswert ist. Hiervon abgesehen, würde

[1] In der ersten Implementierung war dies so der Fall.

dies zu zusätzlichem Aufwand bei der Erstellung der Struktur- und Darstellungsklassen führen und schliesslich deren universelle Verwendbarkeit einschränken.

```
connectable   …
{
   view       <Name der Klasse, die für die Darstellung zuständig ist> size <Standardgrösse>
   controller <Name der Klasse, welche die Steuerung übernimmt>

   icon       <Symbol für die Elementerzeugung (vgl. Abb. 81 und 85)>
   text       <Name des Sprachkonstrukts>
   help       <Hilfetext>

   … Integritätsbedingungen ausgespart, siehe Kapitel 15 …

}
```

Abbildung 89: *Im Rumpf eines Eintrags wird angegeben, welche Klassen für die Darstellung des Elements und für die Steuerung der Benutzerinteraktion zuständig sind. Das Format des Rumpfs ist für Connectables und Connectives gleich.*

Beispiel

Doch nun wieder zurück zum Beispiel, welches mittlerweile bis auf die (Integration der) Integritätsbedingungen vollständig ist. Für das Beispiel sieht das entsprechend vervollständigte Konfigurationsskript folgendermassen aus.

```
connectable   model.AbstractObject
   contains   model.AbstractObject
   {
      view       view.NodeUI size 120 x 80;
      controller controller.ConnectableController;

      icon       "images/create_abstract_object.gif";
      text       "abstract object";
      help       "an abstract object is used to represent objects.
                  It can contain other abstract objects or states";

      … Integritätsbedingungen ausgespart, siehe Kapitel 15 …
   }

connective    model.Relation
   contains   model.Relation
   connects   model.AbstractObject; model.AbstractObject
   {
      view       view.GenericRelationUI size 60 x 20;
      controller controller.ConnectiveController;

      icon  "images/create_generic_relation.gif";
      text  "generic relation";
      help  "a generic relation connects two abstract objects";

      … Integritätsbedingungen ausgespart, siehe Kapitel 15 …
   }
```

Abbildung 90: *Fast vollständiges Konfigurationsskript für den einfachen ADORA-Editor.*

Für die beiden Sprachkonstrukte AbstractObject und Relation, die vom Werkzeug verarbeitet werden sollen, sind die Namen der Klassen angegeben, welche als View und als Controller vorgesehen sind, also jeweils für die Darstellung (siehe Abb. 90, Schlüsselwort view) und die Steuerung (siehe Abb. 90, Schlüsselwort controller) der Modellelemente zuständig sind. Ferner werden im Konfigurationsskript noch einige weitere Informationen abgelegt, die sich erfahrungsgemäss oft ändern können. Hierunter fallen der (für den Benutzer des Werkzeugs sichtbare) Name des Konstrukts (siehe Abb. 90, Schlüsselwort text), die Standardgrösse (siehe Abb. 90, Schlüsselwort size), etc. So müssen die zugrundeliegenden View-Klassen nicht allzu häufig geändert oder abgeleitet werden – zumindest nicht alleine deswegen, weil lediglich die Initialgrösse für ein Modellelement geändert werden soll, o.ä.

Mit diesem Skript ist das Werkzeug nun soweit konfiguriert, dass es (sinnvoll) lauffähig ist. Startet man das Werkzeug, so erhält man den bereits in Abb. 81 gezeigten, einfachen ADORA-Editor. Hiermit erstellte Modelle können aber immer noch bestimmte Inkonsistenzen aufweisen, die grundsätzlich werkzeugseitig bereits erkannt und verhindert werden könnten.

So wäre es beispielsweise ohne weiteres möglich, dass zwei abstrakte Objekte den gleichen Namen haben, obwohl sie Komponente der gleichen Komposition sind (siehe Abb. 91). Um solche Sachverhalte zu erkennen, enthält das Konfigurationsskript – in der bislang vorgestellten Form – noch nicht genügend Informationen.

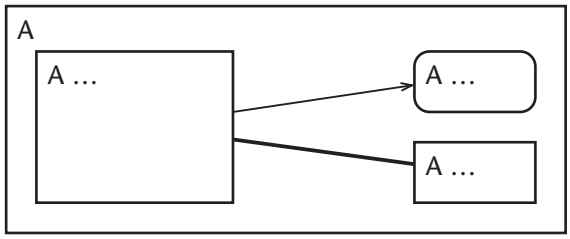


Abbildung 91: *Beispiel für ein fehlerhaftes ADORA-Modell.*
Während die illegale Transition (oben) im Gegensatz zur legalen Beziehung (unten) gar nicht erst möglich ist, da das Konfigurationsskript genügend Informationen enthält, damit legale Konstituenten festgestellt werden können, werden Namenskonflikte nicht erkannt. Das Konfigurationsskript ist hierfür nicht mächtig genug.

Wie mit dem Bereich Modellintegrität, Integritätsbedingungen, Invarianten, etc. umgegangen wird, ist Gegenstand des folgenden Kapitels 15, doch zuvor folgen noch einige Anmerkungen zu abstrakten Sprachkonstrukten (siehe nachfolgendes Kapitel 14.2). Aus Gründen der besseren Verständlichkeit wurde dies in den vorangegangenen Kapiteln aussen vor gelassen.

14.2 Abstrakte Konstrukte

Oft ist es zweckmässig, sog. *abstrakte Sprachkonstrukte* definieren zu können. Dies sind Sprachkonstrukte, wie beispielsweise das BasicElement, die selbst keine konkrete Ausprägung als Modellelement haben werden, sondern nur dazu dienen, gemeinsame Eigenschaften ansonsten unterschiedlicher Sprachkonstrukte zu bündeln.

Für die Konfiguration der grafischen Editorkomponente ist dies zwar nicht notwendig, da für derartige Konstrukte keine direkte Unterstützung notwendig ist. Sollen jedoch Integritätsbedingungen angegeben werden, wie z.B. ‘alle Modellelemente müssen einen Namen haben, der nicht leer sein darf’, so ist dies durchaus angebracht und manchmal mitunter sogar notwendig. Der Teil des Werkzeugs, welcher die Modellelemente verwaltet und letztendlich auch die Konsistenz des Modells überwacht bzw. sicherstellt (siehe Kapitel 15), muss wissen, welche Sprachkonstrukte abstrakt sind und welche Sprachkonstrukte von welchen Sprachkonstrukten abgeleitet sind.

Ein abstraktes Sprachkonstrukt wird im Konfigurationsskript durch das Schlüsselwort abstract angezeigt. Abb. 92 zeigt das Konfigurationsskript für das abstrakte Konstrukt BasicElement. Die Angaben zur Darstellungskomponente, etc. im Rumpf können stehen bleiben, dürfen aber auch weggelassen werden.

```
abstract        model.BasicElement
    {
        … ausgespart, vgl. Abb. 89
    }
```

Abbildung 92: *Beispiel für die Beschreibung abstrakter Konstrukte im Konfigurationsskript. Die Angaben im Rumpf zu Darstellung, Steuerung, etc. sind obsolet und daher optional (vgl. Kapitel 14.1.4).*

Ein Sprachkonstrukt, wie beispielsweise das abstrakte Objekt, welches de facto von einem anderen ggf. auch abstrakten Sprachkonstrukt abgeleitet ist, wird im Konfigurationsskript folgendermassen beschrieben (siehe Abb. 93):

```
connectable     model.AbstractObject
    extends     model.BasicElement
    contains    model.AbstractObject
    {
        … ausgespart, vgl. Abb. 90 …
    }

abstract connective
                model.Relation
    extends     model.BasicElement
    connects    model.AbstractObject
    …

connective      model.UsesRelation
    extends     model.Relation
    contains    model.UsesRelation
    …

connective      model.GeneralRelation
    extends     model.Relation
    contains    model.GeneralRelation; model.UsesRelation
    …
```

Abbildung 93: *Beispiel für die Beschreibung abgeleiteter Konstrukte im Konfigurationsskript.*

Anmerkung: Die Klassen, welche die Sprachkonstrukte repräsentieren (siehe Kapitel 14.1.1 und 14.1.2), müssen der Generalisierung/Spezialisierungs-Hierarchie folgen, wie sie durch das Konfigurationsskript vorgegeben ist.

Wie zu Anfang des Kapitels bereits gesagt: die Möglichkeit abstrakte Konstrukte anzugeben, ist weniger relevant für die grafische Editorkomponente, sondern hat hauptsächlich Auswirkungen auf den Bereich Modellintegrität und Integritätsbedingungen.

15 Integrität des Modells

Die Informationen, welche im Konfigurationsskript (so wie es bisher vorgestellt wurde) abgelegt sind, erlauben es dem Werkzeug, einfache, syntaktische Bildungsregeln zu erkennen und dementsprechend einzuhalten. Werden diese Regeln jedoch komplexer oder ist es zu ihrer Überprüfung notwendig, mehrere oder gar alle Modellelemente zu betrachten, wie beispielsweise bei der eindeutigen Benennung von Objekten (siehe Abb. 91), so ist dies allein aufgrund des Konfigurationsskripts nicht ohne weiteres möglich (vgl. auch [Dit+96], implizite vs. explizite Integritätsbedingung).

Abgesehen davon, dass man dies einfach akzeptieren könnte, gibt es zwei Möglichkeiten, das Problem anzugehen:

(1) Bei der Realisierung der entsprechenden (Model- oder Controller-) Klassen werden auch gleich die zugehörigen Integritätsbedingungen ausprogrammiert.

(2) Über eine weitere Notation wird die Formulierung von Integritätsbedingungen ermöglicht, wobei das Werkzeug in der Lage sein muss, die in dieser Notation formulierten Bedingungen automatisch zu überprüfen.

Der Hauptvorteil von Möglichkeit (1) ist, dass zur Konfiguration des Werkzeugs keine weitere Notation erlernt werden muss. Dem gegenüber stehen die Nachteile, dass der Code zur Überprüfung der Integritätsbedingungen hartverdrahtet im Code des Werkzeugs ist. Für jede Änderung, Ergänzung, etc. muss der Quellcode des Werkzeugs manuell geändert werden. Erschwerend kommt hinzu, dass nicht ohne weiteres alle Bedingungen ausschliesslich in den Model- oder Controller-Klassen verankert werden können, ohne dass massive, zusätzliche Kopplungen entstehen; sowohl unter diesen Klassen wie auch zwischen diesen Klassen und dem Repository. Der Code für die Bedingungen und der Code zur Bedingungsüberprüfung wird bzw. würde quer über den Code des Werkzeugs verteilt.

Um diese Probleme mit Kopplung und Verteilung zu entschärfen, kann zwar eine definierte Schnittstelle im Repository (siehe Klasse AdoraModel) vorgesehen werden, wo die Bedingungen ‘eingehängt’ werden können. Jedoch ist es zusätzlich notwendig, gewisse Konventionen einzuführen, die regeln wie und wo eine Integritätsbedingung auscodiert werden muss. Der Unterschied zu Möglichkeit (2) ist dann nicht mehr gross.

Hauptnachteil von Möglichkeit (2) ist – wie gesagt – die zusätzliche Notation. Diese muss zuerst entwickelt und anschliessend von der Person, welche das Werkzeug konfiguriert, erlernt werden. Jedoch können damit – vorausgesetzt die Notation ist entsprechend geeignet – Integritätsbedingungen einfacher und direkter formuliert werden. Ferner lässt sich die Kopplungs- und Verteilungsproblematik entschärfen. Konfigurationsskript und Integritätsbedingungen zusammen ergeben eine eigenständige Beschreibung derjenigen Sprache, die das Werkzeug aktuell verarbeitet. Der Code im Werkzeug zum Aufruf der Bedingungen hingegen muss der Person, die das Werkzeug konfiguriert, nicht bekannt sein.

Eine Voraussetzung für Möglichkeit (2) ist, dass die Bedingungen automatisch überprüft werden können. Hierzu muss es ein Ausführungsmodell und – analog zur Möglichkeit (1) – eine definierte Schnittstelle geben, wo die Bedingungen 'eingehängt' werden können. Bei geeigneter Wahl der Notation bietet sich der zusätzliche Vorteil, dass nicht nur werkzeugrelevante Integritätsbedingungen, sondern auch Teile der Semantik der Sprache ADORA-L mittels dieser Notation beschrieben werden können.

Wie unschwer zu erraten sein dürfte, wurde hier die Möglichkeit (2) favorisiert, d.h. es wurde eine spezielle Notation entwickelt, die es ermöglicht, Integritätsbedingungen für ADORA-Modelle zu formulieren und automatisch zu überprüfen. Bevor jedoch in Kapitel Kapitel 15.2 auf die eigentliche Notation und deren Ausführungsmodell eingegangen wird, werden nachfolgend in Kapitel 15.1 kurz existierende Ansätze vorgestellt.

15.1 Konfiguration des Werkzeugs – Integritätsbedingungen

Ansätze zur Formulierung und Formalisierung von Integritätsbedingungen finden sich in den unterschiedlichsten Bereichen. In [Sche98] werden drei typische Bereiche genauer betrachtet und die dortigen Ansätze bzw. Notationen auf ihre Eignung zur Formulierung von Integritätsbedingungen für ADORA-Modelle untersucht.

Betrachtet wurden folgende Notationen:

- OCL
- Concert-IB
- ECA-Regeln

Für die ausführliche Beschreibung, Vorstellung und Diskussion der Ansätze sei auf [Sche98] verwiesen. An dieser Stelle sollen nur die wichtigsten Ergebnisse kurz angeführt werden, um – wie gesagt – anschliessend in Kapitel 15.2 den hier verwendeten Ansatz vorzustellen.

OCL

OCL [IBM+97] ist eine getypte, deklarative Sprache, welche als Teil von UML neben der Formulierung von Guards sowie Vor- und Nachbedingungen für Operationen, Methoden, etc. auch die Formulierung von Integritätsbedingungen erlaubt.

```
AbstractObject  self.getType.allInstances ->
                    forAll( o | ( ( self.elementName = o.elementName ) implies ( self = o ) )
```

Abbildung 94: *Beispiel für einen OCL-Ausdruck. Hier die Integritätsbedingung, dass der Name eines abstrakten Objekts eindeutig sein muss.*

OCL ist nicht speziell zur Formulierung von Integritätsbedingungen entwickelt worden, was sich u.a. in diversen syntaktischen Unschönheiten niederschlägt. Hauptproblem an OCL ist jedoch nicht die Syntax, sondern dass es nicht ganz trivial ist, den Kontext eines OCL-Ausdrucks festzustellen (vgl. [Coo+99]) sowie die fehlende Fähigkeit, Ausdrücke zu schachteln (vgl. [Sch+96]), welche eine direkte Konsequenz des Kontextproblems ist.

Concert-IB

Die Notation Concert-IB [Schw95] stammt aus dem Bereich Konfigurationsmanagement und ist von der in [Wieb90] vorgestellten Notation ‘Jason’ abgeleitet. Die Notation dient ganz speziell und im Kontrast zu OCL der Formulierung von Integritätsbedingungen. Concert-IB ist bei Schwille [Schw95] eine Ergänzung der Entity-Relationship-Notation [Chen76], um Bedingungen zu beschreiben, die über das ER-Modell nicht oder nur sehr schlecht ausgedrückt werden können (vgl. [Rund96]). Concert-IB basiert auf Prädikatenlogik erster Ordnung und ist ebenso wie OCL deklarativ und getypt.

```
constraint NameIsUnique on ( anObject : AbstractObject )
   is  ( for-all otherObject in anObject.entitySet )
          ( ( anObject.name = otherObject.name ) implies ( anObject = otherObject ) )
```

Abbildung 95: *Beispiel für eine Integritätsbedingung in Concert-IB.*

Mit Concert-IB steht eine Notation zur Verfügung, welche speziell zur Formulierung von Integritätsbedingungen entwickelt wurde. Der Aufbau einer Integritätsbedingung in Concert-IB deckt sich mit der Definition, die auch Zehnder [Zeh+89] vorschlägt. Jede Integritätsbedingung verfügt über eine definierte Schnittstelle, die den Kontext der Bedingung dokumentiert. Ferner lassen sich in Concert-IB formulierte Bedingungen schachteln.

Aktive Datenbanksysteme

In aktiven Datenbanksystemen [Dit+95] [Dit+96] wird Integrität – hier ist es die Integrität einer Datenbank – über sogenannte ECA-Regeln sichergestellt. ECA steht für Event-Condition-Action, also für Ereignis, Bedingung und Aktion.

```
DEFINE RULE   UniqueName
ON AFTER      AbstractObject.changed
IF            ∃ o1, o2 in Repository •              // Notation für Bedingung nicht vorgegeben
                 ( o1.elementName = o2.elementName ∧ o1 ≠ o2 )
DO            {                                     // Notation für Bedingung nicht vorgegeben
              ... inform user ... or do something else
              }
```

Abbildung 96: *Beispiel für eine ECA-Regel.*

Der Schwerpunkt bei der formalen Beschreibung von ECA-Regeln liegt (leider) auf der Formalisierung der auslösenden Ereignisse und nicht auf der Formalisierung der Bedingungen oder der Aktionen.

In [Dit+96] wird zwar angemerkt, dass die Bedingung ein Prädikat über dem Datenbankzustand ist (welches natürlich auch über eine prädikatenlogische Formel repräsentiert werden könnte, vgl. [Dit+95]), aber obwohl die Bedingung ein zentraler Bestandteil einer ECA-Regel ist, wird zur Beschreibung der eigentlichen (Integritäts-)Bedingung und Aktion einer ECA-Regel keine bestimmte Notation angegeben, sondern lediglich vermerkt, mittels welcher (Programmier-)Sprachen dies geschehen kann.

15.2 ADORA-IB

Mit OCL, Concert-IB und ECA-Regeln wurden im letzen Kapitel unterschiedliche Ansätze für automatisch überprüfbare Integritätsbedingungen vorgestellt. Nun wird die Notation vorgestellt, mittels welcher Integritätsbedingungen für ADORA-Modelle beschrieben werden. Neben der Vorstellung dieser Notation wird in diesem Kapitel auch auf das Ausführungsmodell eingegangen, d.h. wie Integritätsbedingungen im Kontext eines ADORA-Modells automatisch überprüft werden können (siehe Kapitel 15.2.1). Ferner soll dargelegt werden, wie die mit dieser Notation formulierten Bedingungen in die Konfiguration und das Konfigurationsskript des Werkzeugs einbezogen werden (siehe Kapitel 15.2.2).

15.2.1 ADORA-IB

Die Notation, mittels welcher Integritätsbedingungen für ADORA-Modelle formuliert werden, wird ADORA-IB genannt. Diese Notation ergänzt die Konfigurationsnotation, so dass es möglich wird, Sachverhalte zu beschreiben, die sich über das Konfigurationsskript alleine nicht darstellen lassen. Wie gegen Ende von Kapitel 14.1 gezeigt, lassen sich alleine mittels der Konfigurationsnotation bestimmte Bedingungen nicht oder nur auf Umwegen beschreiben. Hierbei geht es um Bedingungen, die sich nicht mehr auf Basis isolierter Betrachtung eines Modellelements, sondern nur über die Betrachtung und den Vergleich mehrerer Elemente eines Modells entscheiden lassen. Beispiele hierfür sind Namenskonflikte (siehe Abb. 91, 95 sowie Kapitel 5.4.1 auf Seite 20) oder die Zyklenfreiheit der Teil-Ganzes-Hierarchie.

ADORA-IB ist keine Notation, die von Grund auf neu geschaffen wurde, sondern basiert auf der Notation Concert-IB von Schwille [Schw95] und ist somit eine Untermenge der Prädikatenlogik erster Stufe. Untermenge deswegen, weil nur geschlossene Formeln[1] zugelassen sind. Ausserdem darf jede Variable nur genau einmal in einer Formel quantifiziert werden. Es wurden ferner Anpassungen vorgenommen, wie beispielsweise eine Typzusicherung bei Quantifizierungen[2] (siehe Abb. 99), um sicherzustellen, dass sich eine Integritätsbedingung auf einer Zielplattform mit statischer Typprüfung – in diesem Fall Java – problemlos automatisch überprüfen lässt (mehr hierzu in Kapitel 15.2.1.2).

Aus OCL [IBM+97] wurden für ADORA-IB Anregungungen für einige der vordefinierten Operationen übernommen (siehe [Sche98]). So finden sich viele der in OCL vordefinierten Operationen ebenfalls als vordefinierte Operationen in ADORA-IB wieder, wenn auch in leicht abgewandelter Form.

Mittels dieser Notation konnte ein Grossteil der Integritätsbedingungen für ADORA-Modelle formuliert werden. Die bereits in ADORA-IB formulierten Integritätsbedingungen finden sich in [Sche98]. Sie werden daher hier nicht nochmals explizit angeführt (auch nicht im Anhang; stattdessen soll dort, um die Ausdruckstärke vom ADORA-IB zu demonstrieren, die Lösung einer Denksportaufgabe, des sogenannten Mister-X-Problems gezeigt werden).

---

1 Eine Formel heisst geschlossen, wenn sie nur gebundene Variablen enthält; vgl. [Scho95].

2 Concert-IB wurde für Smalltalk, also für eine dynamisch getypte Zielplattform geschaffen.

Im Anschluss wird die Notation ADORA-IB vorgestellt. Danach wird gezeigt, wie in ADORA-IB formulierte Bedingungen automatisch überprüft werden können und wie sich diese Bedingungen in das Konfigurationsskript integrieren.

15.2.1.1 Aufbau einer Integritätsbedingung

Eine Integritätsbedingung in ADORA-IB setzt sich zusammen aus dem *Bedingungsteil*, bestehend aus *Bedingungskopf und Bedingungsrumpf*, sowie dem *Aktionsteil*.

```
constraint  <Name der Integritätsbedingung>                    // Kopf
   on       <Formale Parameter>
   is       <prädikatenlogische Formel>                        // Rumpf
   ifTrue   <Java-Block>                                       // Aktionsteil
   ifFalse  <Java-Block>
   ifError  <Java-Block>
end
```

Abbildung 97: *Struktur einer Integritätsbedingung in ADORA-IB.*

Grundsätzlich wird eine Integritätsbedingung für ein bestimmtes Sprachkonstrukt formuliert und muss dann für alle Ausprägungen dieses Konstrukts gültig sein. Eine Integritätsbedingung gibt also eine bestimmte Art von Invariante für ein Modell an.

Bedingungskopf

Der Bedingungskopf gibt einen (für eine bestimmte Konfiguration) eindeutigen Namen für die Integritätsbedingung (siehe Abb. 97 oder 98, Schlüsselwort constraint) sowie die Schnittstelle in Form formaler Parameter an (siehe Abb. 97 oder 98, Schlüsselwort on).

```
constraint UniqueName
   on    ( elem : BasicElement; set : Set)
   …

constraint NameNotEmpty
   on    ( elem : BasicElement )
   …

constraint LegalName
   on    ( elem : BasicElement )
…
```

Abbildung 98: *Beispiel Bedingungskopf. Gezeigt werden die Bedingungsköpfe für die drei Integritätsbedingungen UniqueName, NameNotEmpty und LegalName. BasicElement ist eine Strukturklasse und Set ist ein (beliebiger) Container, welcher im Rumpf für die Quantifizierung verwendet werden kann. Damit wird die Bedingung UniqueName wiederverwendbar und ist unabhängig vom Kontext der Bedingung LegalName (vgl. hierzu Abb. 99).*

Der Bedingungskopf gibt über die formalen Parameter an, für welche Elemente eine Bedingung definiert ist, d.h. er gibt den Kontext der Bedingung an und schafft über die Möglichkeit zur eindeutigen Benennung einer Integritätsbedingung auch die Voraussetzung dafür, dass Integritätsbedingungen geschachtelt werden können (vgl. OCL und siehe Abb. 99).

Mit Ausnahme von Quantifizierungen und den durch die Strukturklassen (siehe Kapitel 13, S. 139) vorgegebenen Operationen bzw. Methoden kann im Bedingungsrumpf ausschliesslich auf die im Bedingungskopf deklarierten Bezeichner zugegriffen werden.

Bedingungsrumpf

Der Rumpf einer Bedingung besteht aus einer prädikatenlogischen Formel erster Stufe. Im Bedingungsrumpf steht somit die eigentliche Integritätsbedingung (siehe Abb. 97 oder 99, Schlüsselwort is).

Wenn die Integritätsbedingung erfüllt sein soll, muss diese Formel für alle Ausprägungen des Sprachkonstrukts, für welches die Bedingung definiert ist, gültig, also wahr sein. Praktisch wird die Gültigkeit auf einer endlichen Menge vom (Modell-)Elementen geprüft. Die Integritätsbedingung ist nicht erfüllt, wenn die Formel für mindestens ein Element falsch ist.

Anmerkung: Solange nicht sämtliche Elemente des Modells bestimmt werden müssen, die die Bedingung nicht erfüllen, bedeutet dies nicht zwingend, dass die Bedingung für alle Elemente überprüft werden muss.

Die Formel im Bedingungsrumpf bildet sich in bekannter Weise. Aus Termen werden atomare Formeln gebildet. Atomare Formeln werden zu zusammengesetzten Formeln (unäre, binäre und quantifizierte) kombiniert. Bereits definierte Bedingungen können eingebettet werden, so dass Integritätsbedingungen entsprechend geschachtelt werden können (siehe Abb. 99, Integritätsbedingung LegalName).

```
constraint UniqueName
    on      ( elem : BasicElement; set : Set )
    is      ( for-all other : BasicElement in set )
                ( ( elem.elementName().equals( other.elementName() ) = true )
                    implies ( elem = other ) )
    …

constraint NameNotEmpty
    on      ( elem : BasicElement )
    is      ( not ( elem.equals( "" ) = true ) )
    …

constraint LegalName
    on      ( elem : BasicElement )
    is      ( predicate NameNotEmpty( elem )
                and predicate UniqueName( elem, elem.getType().allInstances() ) )
    …
```

Abbildung 99: *Beispiel Bedingungsrumpf. Die Integritätsbedingung LegalName setzt sich hierbei aus den beiden Bedingungen UniqueName und NameNotEmpty zusammen.*
Gezeigt wird auch wie Methoden aus unterliegenden Strukturklassen aufgerufen werden können. So ist beispielsweise die Operation elementName(), in welcher in der Bedingung UniqueName benutzt wird, eine Methode der Klasse BasicElement. In der Bedingung UniqueName ist ausserdem die Typzusicherung bei der Quantifizierung zu sehen. Diese ist notwendig, wenn die Bedingungen auf Basis einer streng getypten Programmiersprache – wie z.B. Java – ausgeführt werden sollen.

Weiter soll an dieser Stelle nicht auf den Bedingungsrumpf eingegangen werden. Die vollständige EBNF für ADORA-IB findet sich im Anhang A. Eine ausführliche Beschreibung der Formeln, welche im Bedingungsrumpf zulässig sind, wie auch eine Beschreibung der vordefinierten Operationen, etc., findet sich in [Sche98] und wird daher nicht wiederholt.

Aktionsteil

Wie der Name schon sagt, gibt der Aktionsteil Aktionen an, die wahlweise ausgeführt werden, wenn die Auswertung einer Formel wahr bzw. falsch ist. Dafür besteht im Aktionsteil die Möglichkeit, je einen Aktionsblock anzugeben. Der eine wird ausgeführt, wenn die Formel gültig ist. Der andere, wenn die Formel nicht gültig ist (siehe Abb. 97 oder 99, Schlüsseworte ifTrue, ifFalse). Ausserdem besteht die Möglichkeit, darauf zu reagieren, dass eine Integritätsbedingung nicht (vollständig) überprüft oder ein Aktionsteil nicht (vollständig) ausgeführt werden konnte, weil beispielsweise die Evaluation der Formel zu einem Fehler geführt hat (siehe Abb. 97 oder 99, Schlüsselwort ifError). Schlägt auch die Ausführung des Fehlerbehandlungsblocks fehl, so wird bei geschachtelten Integritätsbedingungen der entsprechend der lexikalischen Struktur nächsthöherliegende Fehlerbehandlungsblock aktiv, also der Block der Bedingung, in welcher die Bedingung eingeschachtelt ist. Ist kein solcher Block vorhanden, wird die Auswertung ohne konkretes Resultat abgebrochen.

```
constraint LegalName
    …
    ifTrue    ( )
    ifFalse   ( inspection.Inspector.inspect( elem ); )    // öffne einen Inspektor auf dem 'faulen Ei'
    ifError   ( )
end
```

Abbildung 100: *Beispiel, Aktionsteil einer Integritätsbedingung.*

Zur Beschreibung der Aktionen ist keine spezielle Notation vorgesehen. Eine Aktion wird – zumindest derzeit – durch entsprechenden Quellcode in der Zielsprache des Werkzeugs angegeben, also hier durch einen Java-Code-Block (siehe Abb. 99). Dies macht eine in ADORA-IB formulierte Integritätsbedingung bzw. deren Aktionsteil zwar plattformabhängig, gibt dem Benutzer jedoch alle Freiheit, wie (im Werkzeug) auf die Auswertung einer Bedingung reagiert werden kann. Wie gesagt, 'lediglich' der Aktionsteil ist plattformabhängig, nicht jedoch der Bedingungsteil (vgl. Kapitel 15.1).

15.2.1.2 Automatische Überprüfung von Integritätsbedingungen

Das Ausführungsmodell für Integritätsbedingungen ist wie folgt: Der ADORA-IB-Quellcode einer Integritätsbedingung wird mittels eines Übersetzers (siehe [Sche98], IBCompiler) in Java-Quell-Code übersetzt. Dieser Java-Quellcode kann wiederum mittels eines normalen Java-Übersetzers in Java-Byte-Code übersetzt werden. Der Java-Byte-Code ist dann (zusammen mit einigen Hilfsklassen, die für die Quantifizierung gebraucht werden, vgl Abb. 101) auf einer Java VM lauffähig. Derzeit wird für jede Integritätsbedingung eine eigene Klasse mit einer entsprechenden Methode erzeugt, welche dann die eigentliche prädikatenlogische Formel wiederspiegelt.

```
public class LegalName
    {
    public boolean LegalName( BasicElement elemin )
        {
        boolean result =  true ;
        try
            {
            final BasicElement elem = elemin;
            result = ( ( ( new NameNotEmpty() ).NameNotEmpty( elem )
                && ( new UniqueName() ).UniqueName( elem, elem.getType().allInstances( ) ) ) );
            if ( result )
                { }
            else
                { inspection.Inspector.inspect( elem ); }
            }
        catch (Exception e )
            { }
        return result;
        }
    }


public class UniqueName
    {
    public boolean UniqueName( BasicElement elemin, Set setin )
        {
        boolean result =  true ;
        try
            {
            final BasicElement elem = elemin;
            final Set set = setin;
            result = new Quantification( set ).forAll( new Formula() {
                boolean eval( Object in )
                    {
                    final BasicElement other = ( BasicElement ) in ;
                    return ( ( ( elem.elementName( ).equals( other.elementName( ) ) == true )
                        ^  true  ) || ( elem == other )  );
                    }
                }
             );
            if ( result )
                { }
            else
                { }
            }
        catch ( Exception e )
            { }
        return result;
        }
    }
```

Abbildung 101: *Java-Quellcode, welcher vom IBCompiler aus den beiden ADORA-IB Integritätsbedingungen LegalName und UniqueName aus Abb. 99/100 erzeugt wird.*
Gezeigt wird speziell: (1) Wie die Operation elementName() 'nach unten' durchgereicht (siehe UniqueName) wird und zum Methodenaufruf eines Objekts der Klasse BasicElement wird (vgl. Abb. 82). (2) Wie die Schachtelung von Prädikaten übersetzt wird (siehe LegalName). (3) Welche Hilfsklassen für die Quantifizierung benötigt werden (siehe UniqueName).

Beispiel

Abb. 101 zeigt den (nachträglich zur besseren Lesbarkeit entsprechend formatierten) Java-Quellcode, welcher vom IBCompiler aus den in Abb. 99/100 gezeigten Integritätsbedingungen LegalName und UniqueName erzeugt wird.

Um eine Integritätsbedingung auszuführen bzw. zu überprüfen, muss vom Werkzeug eine Instanz eines Bedingungsobjekts erzeugt werden und für diese Instanz die entsprechende, namensgleiche Methode aufgerufen werden. Dafür muss dem Werkzeug noch bekanntgemacht werden, für welche Sprachkonstrukte welche Integritätsbedingungen gültig sein müssen. Bevor hierauf in Kapitel 15.2.2 eingegangen wird, folgt unmittelbar anschliessend noch eine Anmerkung zur Thematik Ausführung von prädikatenlogisch definierten Integritätsbedingungen und Entscheidbarkeit.

15.2.1.3 Entscheidbarkeit

Integritätsbedingungen werden hier mit den Mitteln der Prädikatenlogik, das heißt über prädikatenlogische Formeln definiert. Generell ist das Gültigkeitsproblem der Prädikatenlogik unentscheidbar (siehe [Scho95], d.h. es gibt keinen Algorithmus, der für eine beliebige Prädikatenlogische Formel erster Stufe in endlicher Zeit entscheiden kann, ob die Formel erfüllbar ist (Beweis von Church über die Reduktion auf das Post'sche Korrespondenzproblem). Somit könnte – grundsätzlich betrachtet – die automatische Überprüfung der Integritätsbedingungen problematisch werden. Da es sich bei ADORA-Modellen um endliche Modelle handelt, bzw. die Menge von Modellelementen zum Zeitpunkt der Überprüfung immer endlich ist, entschärft sich dieser Sachverhalt.

Tabelle 3: *Entscheidbarkeit prädikatenlogischer Formeln (grau hinterlegt = unentscheidbar/weiss hinterlegt = entscheidbar)*

<table>
<tr><td rowspan="2">gültige Formeln</td><td>erfüllbare aber nicht gültige Formeln mit unendlichen Modellen</td><td rowspan="2">unerfüllbare Formeln</td></tr>
<tr><td>erfüllbare aber nicht gültige Formeln mit endlichen Modellen</td></tr>
</table>

Die Erfüllbarkeit von Formeln F mit Strukturen $A = (U_A, I_A)$ mit einer *endlichen* Grundmenge U_A ist entscheidbar. So ist die Frage, ob die Formel F für die Struktur A erfüllbar ist oder nicht, hier immer entscheidbar, da die Grundmenge der zu prüfenden Struktur zum Zeitpunkt der Auswertung der Integritätsbedingung(en) immer endlich ist[1]. Es sei ausserdem noch angemerkt, dass auch das Gültigkeitsproblem von Formeln mit Strukturen mit endlicher Grundmenge entscheidbar ist.

[1] Hier können prinzipiell alle Quantifizierungen in eine Reihe von Disjunktionen bzw. Konjunktionen aufgelöst werden.

15.2.2 Integration der Bedingungen in das Konfigurationsskript

Nachdem die Notation ADORA-IB im vorangehenden Kapitel eingeführt wurde, gilt es nun noch darzulegen, wie die mittels dieser Notation formulierten Bedingungen in das Konfigurationsskript integriert werden.

Hier gibt es vorwiegend zwei Sachverhalte zu klären: (1) Für welche Modellelemente gelten welche Integritätsbedingungen und (2) wann genau soll ein Modell auf Einhaltung dieser Bedingungen geprüft werden. Beides wird im Konfigurationsskript im Constraint-Teil angegeben (siehe Abb. 102, Schlüsselwort constraints). Hierbei wird definiert, welches Ereignis die Überprüfung welcher Integritätsbedingung zur Folge hat.

Ereignisse, sog. ADORA-Model-Events (kurz AMEV), sind hierbei immer auf das Repository (Klasse AdoraModel) bzw. die dort abgelegten Modellelemente bezogen und betreffen das Hinzufügen, Ändern, Löschen, etc. eines Modellelements.

```
connectable   <Name der Klasse, die das Sprachkonstrukt repräsentiert>
   extends    <Name der Klasse, von welcher dieses Konstrukt abgeleitet ist>
   contains   <Name(n) der Klasse(n), die die Sprachkonstrukte repräsentieren, deren
                 Elemente enthalten sein können>
   {
      view        <Name der Klasse, die für die Darstellung zuständig ist> size <Standardgrösse>
      controller  <Name der Klasse, welche die Steuerung übernimmt>

      icon        <Symbol für die Elementerzeugung>
      text        <Name des Sprachkonstrukts>
      help        <Hilfetext>

      constraints
      {
         <Name des Adora-Model-Events> -> <Name der Integritätsbedingung>
         <Name des Adora-Model-Events> -> <Name der Integritätsbedingung>
         …
      }
   }
```

Abbildung 102: *Vollständige Struktur (incl. Integritätsbedingungen, vgl. Abb. 89 auf Seite 147) für einen connectable-Eintrag im Konfigurationsskript (analog für connectives).*

Wird eine Operation im Repository durchgeführt, so übermittelt das Repository das entsprechende AMEV an die Komponente, welche für die Überprüfung der Integritätsbedingungen zuständig ist. Diese Komponente, der AdoraModelChecker[1], findet in Abhängigkeit vom entsprechenden Ereignis und der aktuellen Konfiguration des Werkzeugs heraus, welche Integritätsbedingungen für ein Modellelement zu überprüfen sind und stösst anschliessend die automatische Überprüfung der Bedingungen an.

[1] Die Bezeichnung AdoraModelChecker ist etwas unglücklich gewählt, da ‘model checking’ mittlerweile in der Literatur eine andere, fixe Bedeutung hat. Der AdoraModelChecker führt kein ‘model checking’ im eigentlichen Sinn durch, sondern prüft ein ADORA-Modell gegen die im Konfigurationsskript angegebenen Integritätsbedingungen.

Welche AMEVs existieren und wann welches AMEV auftritt, ist vordefiniert. Das Werkzeug kann also nicht in dem Sinne konfiguriert werden, dass über das Konfigurationsskript definiert werden kann, welche Ereignisse existieren und wann diese auftreten. Derzeit kann auf folgende Ereignisse reagiert werden:

- Element zum Modell hinzufügen (AMEV_ADD)
- Modellelement ändern (AMEV_CHANGE)
- Element aus dem Modell entfernen (AMEV_REMOVE)
- No Operation (AMEV_NOP)

Während AMEV_ADD, AMEV_CHANGE und AMEV_REMOVE intuitiv klar sein sollten, ist AMEV_NOP erklärungsbedürftig. AMEV_NOP ist ein Dummy-Ereignis, um Bedingungen einbinden zu können, deren Prüfung nicht an ein auslösendes Ereignis gebunden ist, welches vom Repository nach einer entsprechenden Operation erzeugt wird. Solche Integritätsbedingungen werden dann überprüft, wenn die Überprüfung manuell ausgelöst wird. Da AMEV_NOP äusserst selten verwendet wird – hauptsächlich fürs Debugging – ist es lediglich vollständigkeitshalber mit aufgeführt.

Bei der Überprüfung der Bedingungen wird so vorgegangen, dass zuerst die eigentliche Operation – hinzufügen, ändern oder entfernen – im Repository durchgeführt wird. Anschliessend wird das entsprechende Ereignis generiert und so die Überprüfung der Bedingungen angestossen. Die Prüfung findet also ‘post mortem’ statt und zeigt an, dass das Modell momentan inkonsistent ist. Es ist jedoch ohne weiteres möglich, das Verhalten so zu ändern, dass, bevor die Operation durchgeführt wird, ein Ereignis generiert wird und ein weiteres im Anschluss an die Operation. So kann verhindert werden, dass sich das Repository (kurzfristig) in einem nicht konsistenten Zustand befindet.

In der Implementierung des Werkzeugs werden AMEVs durch Objekte der Klasse AdoraModelEvent repräsentiert. Sollen neue Ereignisse definiert werden, muss diese Klasse geändert und die Operationen des Repositories (Klasse AdoraModel) entsprechend angepasst werden. So dass dann beispielsweise anstatt AMEV_ADD die Ereignisse AMEV_ADDING und AMEV_ADDED bereitgestellt werden. Die Operation zum Hinzufügen eines Elements im Repository muss dann so geändert weden, dass der AdoraModelChecker einmal ‘angeworfen’ wird, bevor die eigentliche Operation durchgeführt wird und ein zweites mal unmittelbar im Anschluss an diese.

Grundsätzlich kann die Anregung zur Überprüfung von Integritätsbedingungen nur vom Repository ausgehen. Die Integritätsbedingungen selbst können direkt[1] keine Ereignisse erzeugen, welche die Überprüfung weiterer Bedingungen zur Folge hätten. Als Alternative hierzu können und sollten Integritätsbedingungen jedoch geschachtelt werden (siehe Kapitel 15.2.1). Diese Strategie – ein auslösendes Ereignis, danach Aufrufhierarchie der Bedingungen – bietet den

[1] Indirekt können sie dies (auf halb-legale Weise) dann, wenn im Aktionsteil einer Integritätsbedingung das Repository manipuliert wird.

Vorteil, dass jederzeit transparent ist und bleibt, was bei der Überprüfung abläuft. Sie ist so auch ein Beitrag zur Vermeidung nicht-terminierender Bedingungsketten.

Beispiel

Abb. 103 zeigt, wie in das Konfigurationsskript (siehe Abb. 90 auf Seite 147) eine Integritätsbedingung (siehe Abb. 99) integriert wird. Dies geschieht, indem Paare zwischen einem auslösenden Ereignis und zu überprüfender Bedingung gebildet werden.

```
connectable    model.AbstractObject
   contains    model.AbstractObject
   {
      … ausgespart, siehe Abb. 90 auf Seite 147 …

      constraints
      {
         AMEV_CHANGE -> LegalName
         AMEV_ADD -> LegalName
      }
   }
```

Abbildung 103: *Integration der Integritätsbedingungen (hier die Bedingung LegalName, siehe Abb. 99) in das Konfigurationsskript.*
Nach dem Hinzufügen, Ändern, etc. eines Elements im Repository wird das entsprechende AMEV sowie das entsprechende Element an den ModelChecker übermittelt. Dieser findet in Abhängigkeit von der Konfiguration des Werkzeugs heraus, welche Integritätsbedingung(en) zu überprüfen sind. Hier wäre das die Bedingung LegalName. Der ModelChecker stösst dann die Überprüfung dieser Bedingungen an.

16 Zusammenfassung & Diskussion

Schwerpunkt des Teils III dieser Arbeit war es, einige Prinzipien der Konzeption des Werkzeugs vorzustellen, welches das in Teil II vorgestellte Visualisierungskonzept umsetzt und ausserdem flexibel genug ist, dass es an zukünftige Änderungen von ADORA-L angepasst werden kann. Der hier vorgestellte Ansatz für ein solch flexibel konfigurierbares Werkzeug, welches das in Teil II vorgestellte Visualisierungskonzept umsetzt, ist für sich gesehen vollständig und abgeschlossen. Mit der Konzeption des grafischen Editors wurde begonnen, als die Sprache ADORA-L noch in der Entwicklung war. Daher ist der Editor (noch) nicht in ADORA-L spezifiziert, sondern in UML. So ist die Spezifikation der Werkzeuge bzw. des Editors in ADORA-L eine Aktivität, die nicht erfolgen konnte, als der Editor konzipiert wurde, aber sicherlich für die Zukunft noch ansteht[1].

Das Vorgehen, zur Anpassung des Werkzeugs eine Kombination aus ‘Framework Reuse’ und ‘Skripting’ zu fahren, ist eine Strategie, welche sich bereits in anderen Ansätzen, wie beispielsweise vis-A-vis [Lic+93], bewährt hat. Dieses Vorgehen erlaubt es, das Werkzeug an eine kontrollierte Evolution der Sprache ADORA-L anzupassen – erfordert allerdings das ‘händische Auscodieren’ der verwendeten Sprachkonstrukte. Es ist zwar ohne weiteres denkbar, auch hierfür eine massgeschneiderte Notation bereitzustellen und basierend auf den entsprechenden Beschreibungen Code zu generieren. Der zusätzliche Nutzen ist jedoch recht fragwürdig. Im Wesentlichen würde dabei auf einer sehr ähnlichen Abstraktionsebene nichts anderes getan, wie beim ‘händischen Auscodieren’ der einzelnen Komponenten in der Sprache der Zielplattform (hier Java).

Der kritische Punkt bei einer Anpassung oder Änderung des Werkzeugs ist meist nicht die Erstellung der MVC-Komponenten, welche die individuellen Sprachkonstrukte repräsentieren, sondern vielmehr die Beschreibung der Zusammenarbeit dieser Komponenten sowie das Sicherstellen der Integrität der Modell-Daten im Repository. Diese Probleme wurden mittels zweier Notationen angegangen – der Notation zur Konfiguration des Werkzeugs sowie der Notation ADORA-IB zur Formulierung von Integritätsbedingungen. Mittels dieser Notationen lassen sich sowohl die Bindungs- bzw. Bildungsregeln zwischen den einzelnen Sprachkonstrukten angeben, wie auch die Integritätsbedingungen für das entstehende Modell beschreiben, ohne dass dies werkzeugseitig manuell im Quellcode nachgeführt werden müsste. Prädikatenlogik erster Stufe als Mittel zur Beschreibung von Integritätsbedingungen zu verwenden, ist ein Ansatz, welcher sich schon in einigen anderen Bereichen – vgl. [IBM+97] [Schw95] [Wieb90] – bewährt hat. Die derzeit für ADORA-L relevanten Integritätsbedingungen konnten auf dieser Basis beschrieben werden (siehe [Sche98]).

Im Rahmen dieser Arbeit wurde der Bereich Modellformat und Modellpersistenz nicht angegangen[2]. Insbesondere wurde nicht untersucht, inwieweit bzw. wie elegant das Meta-Modell

[1] Der Editor war zum Zeitpunkt der Fertigstellung dieser Arbeit in einem Zustand, in welchem noch nicht alle Migrationsaktivitäten von Java 1.0.x auf Java 1.1.x abgeschlossen waren und die Umstellung auf Swing bevorstand.

von ADORA-L und die zugehörige Visualisierung in XML beschrieben werden können, so dass diese Beschreibung in einer ähnlichen Form wie das hier vorgestellte Konfigurationsskript als Ausgangspunkt bzw. Eingabe für die automatische Konfiguration des Werkzeugs dient. Ebenso könnte dann die Datenhaltung auf Basis eines XML-fähigen Repositories ablaufen. Auch wenn die Datenmenge, welche für ein ADORA-Modell anfällt, nicht so gross ist, dass die Verwendung kommerzieller Repositories ratsam wäre, besteht in der 'Variante XML' eine zukünftig ernsthaft zu prüfende Alternative, da hiermit der Anschluss an einen Standard und damit auch an Standardwerkzeuge geschaffen werden kann.

OCL [IBM+97] in seiner jetzigen Form zur Beschreibung von Integritätsbedingungen zu verwenden, wäre trotz der fehlenden Möglichkeit zur Schachtelung der Bedingungen zwar möglich, jedoch ist die automatische Überprüfung aufgrund des fehlenden Kontexts einer OCL-Bedingung eher problematisch. Sollten zukünftige Versionen von OCL diesen Mangel nicht mehr aufweisen, so könnte die Definition der Integritätsbedingungen auch über OCL erfolgen. Einen entsprechenden Übersetzer vorausgesetzt, welcher aus einer OCL-Bedingung ausführbaren Code für die Zielplattform – hier Java – generiert, kann die Integration von OCL-Bedingungen genauso erfolgen, wie für in ADORA-IB formulierte Bedingungen.

Die Prinzipien zur Beschreibung und Integration der Integritätsbedingungen sind in ihrer Gesamtheit an die Ideen der ECA-Regeln angelehnt, wie sie aus dem Bereich der Aktiven Datenbanksysteme bekannt sind. Dementsprechend führt auch hier eine exzessive bzw. unreflektierte Nutzung dieses Mechanismus' zu den aus diesem Bereich bereits bekannten Problemen betreffend das Laufzeit- und Terminationsverhalten. Etwas entschärft wird dieser Sachverhalt jedoch dadurch, dass es zumindest nicht direkt möglich ist, aus einer in ADORA-IB formulierten Bedingung (AMEV-)Ereignisse zu erzeugen, welche die Überprüfung weiterer Bedingungen zur Folge haben. So ist transparent – da statisch über die Schachtelung der Bedingungen nachvollziehbar – wie die Prüfung der Modellintegrität von statten geht. Probleme der Regeldistribution, also Fragen wie 'welche Bedingungen gehören zum Schema und welche zur Applikation?', sind hier nicht relevant, da es sich nicht um Datenbank plus Applikationen handelt, sondern um eine integrierte Werkzeugumgebung. Die Frage, ob eine Regel datenbank- oder applikationsspezifisch ist, stellt sich somit nicht.

[2] Um ein Modell zu speichern, wird der Serialisierungsmechanismus von Java verwendet, welcher das gesamte AdoraModel als Datei auf der Festplatte o.ä. speichert.

Anhang

A EBNF, ADORA-IB

Hier aufgeführt ist die EBNF für die Notation zur Formulierung von Integritätsbedingungen.

Constraint

Constraint = `constraint` ConstraintName `on` FormalParameters
`is` ConstraintBody
`ifTrue` JavaBlock
`ifFalse` JavaBlock
`ifError` JavaBlock
`end` .

Constraint Name

ConstraintName = DeclareIdentifier .

Formal Parameters

FormalParameters = `(` FormalParameterSequence `)` .
FormalParameterSequence = FormalParameterFinal { `;` FormalParameterFinal } .
FormalParameterFinal = FormalParameter .
FormalParameter = DeclareIdentifier `:` FormalParameterType .
FormalParameterType = Identifier .

DeclareIdentifier = Identifier .

Constraint Body

ConstraintBody = ConstraintInvocation | `(` Formula .

Constraint Invocation

ConstraintInvocation = `predicate` PredicateName ActualParameters .
PredicateName = DeclareIdentifier .

ActualParameters = `(` { ActualParameterSequence } `)` .
ActualParameterSequence = Term { `,` Term } .

Formula

Formula = QuantifiedFormula |
BinaryFormula |
UnaryFormula |
AtomicFormula .

Quantified Formula

QuantifiedFormula = Quantifier IterationVariable `in` Term `)`
ConstraintBody .

Quantifier = `for-all` | `there-exists` .
IterationVariable = FormalParameter .

Binary Formula

BinaryFormula = ConstraintBody Connective ConstraintBody `)` .

Connective = and | or | implies | iff .

Unary Formula

UnaryFormula = not ConstraintBody) .

Atomic Formula

AtomicFormula = Term RelOp Term) .
RelOp = = | <> | < | > | <= | >= .

Term

Term = (Variable | Literal) FunctionApplication .

Variable = DeclaredIdentifier .

Function Application

FunctionApplication = { . (TypeCast | FunctionName { ActualParameters }) } .

TypeCast = asType (Identifier) .
FunctionName = DeclaredIdentifier .

Identifiers

Character = A .. Z | a .. z .
DeclaredIdentifier = Identifier .
Digit = 0 .. 9 .
Identifier -> Character { Character | Digit } .
Literal = Number | String .
Number = { Digit } .
String = " { Character } " .

Java Block

JavaBlock = (*… siehe Java Language Specification [Gos+96]…*) .

B EBNF, KONFIGURATIONS-NOTATION

Hier aufgeführt ist die EBNF für die Notation zur Konfiguration des Werkzeugs.

Start -> SetupEntrySequence .

SetupEntry

SetupEntrySequence -> SetupEntry [SetupEntrySequence] .
SetupEntry -> `abstract` ElementDeclaration
| [`abstract`] `connectable` Connectable
| [`abstract`] `connective` Connective .

Connectables

Connectable -> ElementDeclaration
SetupEntryBody .

Connectives

Connective -> ElementDeclaration ConstituentsDeclaration
SetupEntryBody .
ConstituentsDeclaration -> `connects` DeclaredElement Delimiter DeclaredElement.

ElementDeclaration

ElementDeclaration -> ElementNameDeclaration
[ExtentsDeclaration] [ContainmentsDeclaration] .

ElementNameDeclaration -> Identifier .
ExtendsDeclaration -> `extends` DeclaredElement .
ContainmentDeclaration -> `contains` DeclaredElements .
DeclaredElements -> DeclaredElement [Delimiter DeclaredElements]
DeclaredElement -> Identifier .

SetupEntryBody

SetupEntryBody -> BracketOpen
View Controller Icon Text Help Constraints
BracketClosed .

View -> `view` Identifier Delimiter `size` number `x` number delimiter .
Controller -> `controller` Identifier Delimiter .
Icon -> `icon` Identifier Delimiter .
Text -> `text` BracketOpen Identifier BracketClosed Delimiter .
Help -> `help` BracketOpen Identifier BracketClosed Delimiter .

Constraints

Constraints -> `constraints` BracketOpen ConstraintAssociationSequence BracketClosed .
ConstraintAssociationSequence -> ??? .
…

Identifiers, Delimiters, Brackets, ...

```
BracketOpen -> { .
BracketClosed -> } .
Character -> A .. Z | a .. z .
CharacterSequence -> { Character | Space } .
Delimiter -> ; .
Digit -> 0 .. 9 .
Identifier -> Character { Character | Digit } .
Space -> _ .
```

C Eine Beispielkonfiguration

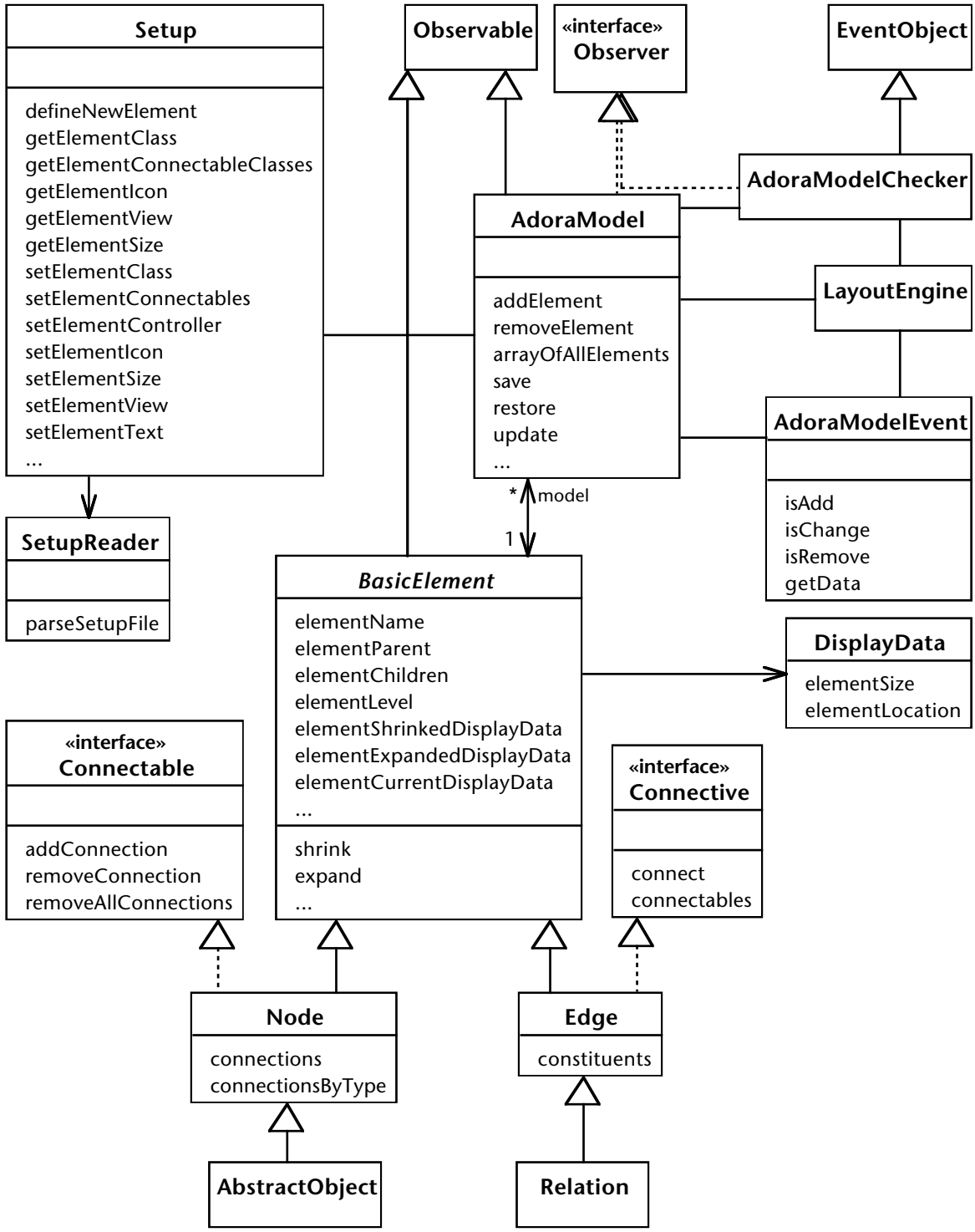
```
abstract        model.BasicElement
    {
        constraints
            {
            }
    }


connectable     model.AbstractObject
    extends     model.BasicElement
    contains    model.AbstractObject; model.State
    {
        view        view.AbstractObjectUI size 120 x 80;
        controller  controller.ConnectableController;

        icon        "images/create_abstract_object.gif";
        text        "abstract object";
        help        "an abstract object is used to represent objects. It acts as a placeholder
                    for instances and can can contain other abstract objects or states";

        constraints
            {
            }
    }


connectable     model.State
    extends     model.BasicElement
    contains    model.State
    {
        view        view.StateUI size 100 x 60;
        controller  controller.ConnectableController;

        icon        "images/create_state.gif";
        text        "state";
        help        "a state is used to describe discrete dynamic behavior and represents
                    a state of a finite state machine. A State can contain other states only";

    constraints
        {
        }
    }


connectable     model.StartState
    extends     model.State
    contains    model.State; model.StartState
    {
        view        view.StartStateUI size 100 x 60;
        controller  controller.ConnectableController;

        icon        "images/create_start_state.gif";
        text        "start state";
```

```
        help      "a start state of the finite state machine";

    constraints
        {
        }
    }


abstract connective model.Relation
    extends     model.BasicElement
    contains    model.Relation
    connects    model.AbstractObject; model.AbstractObject
    {
        constraints
        {
        }
    }


connective      model.GenericRelation
    extends     model.Relation
    contains    model.GenericRelation; model.UsesRelation
    connects    model.AbstractObject; model.AbstractObject
    {
        view          view.GenericRelationUI size 60 x 20;
        controller    controller.ConnectiveController;

        icon    "images/create_generic_relation.gif";
        text    "genric relation";
        help    "a generic relation represents an abitrary communication between abstract
                objects and connects two abstract objects. A generic relation can contain
                other generic relations or uses relations";

    constraints
        {
        }
    }


connective      model.UsesRelation
    extends     model.Relation
    contains    model.UsesRelation
    connects    model.AbstractObject; model.AbstractObject
    {
        view          view.UsesRelationUI size 60 x 20;
        controller    controller.ConnectiveController;

        icon    "images/create_uses_relation.gif";
        text    "uses relation";
        help    "a uses relation represents an client server communication between two
                abstract objects. A generic relation can contain uses relations only";

    constraints
        {
        }
    }
```

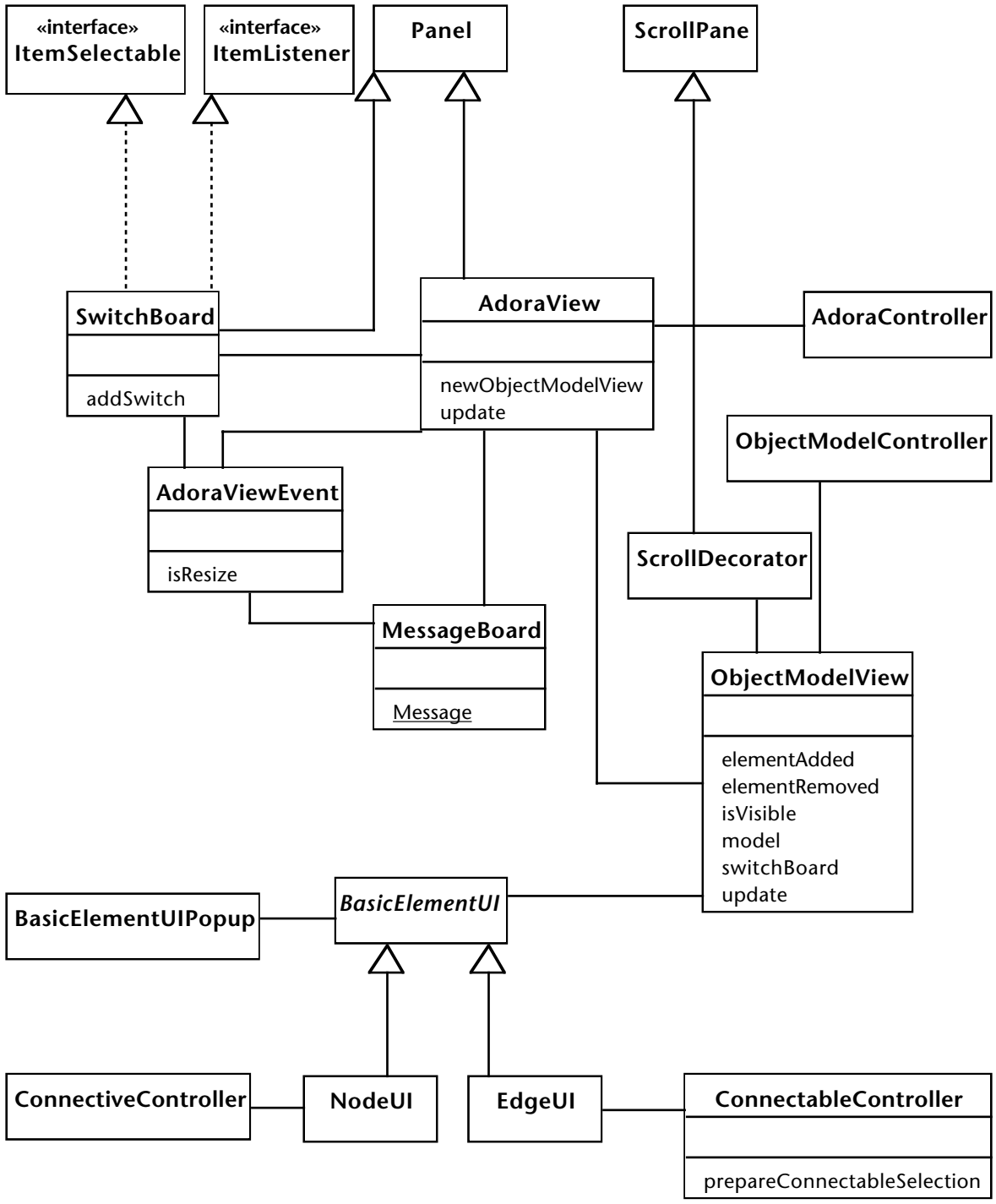
```
connective      model.StateTransition
    extends     model.BasicElement
    connects    model.State; model.State
    {
        view          view.StateTransitionUI size 80 x 40;
        controller    controller.ConnectiveController;

        icon    "images/create_state_transition.gif";
        text    "state transition";
        help    "a state transition represents a transition between two states. A transition
                is triggered by an event and can perform an action.";

    constraints
        {
        }
    }
```

D SYSTEMÜBERSICHT

Auszug aus dem Paket Model



Auszug aus den Pakenten View & Controller



E EIN ADORA-IB BEISPIEL

Mister-X-Problem. Mister X denkt sich 2 natürliche Zahlen zwischen 1 und 100 aus. Dabei sind die Zahlen weder 1 noch 100. Nun teilt er Susan die Summe der beiden Zahlen mit. Peter bekommt das Produkt der beiden Zahlen mitgeteilt. Die beiden sollen nun die beiden Zahlen herausfinden, die sich Mister X ausgedacht hat. Dabei dürfen sie sich ihre Zahlen nicht mitteilen. Nach einiger Zeit sagt Peter: “Ich kann nicht eindeutig sagen, welche die originalen Zahlen sind.” Susan entgegnet, dass sie auch nicht eindeutig sagen kann, welche die beiden Zahlen sind. Aber sie meint, dass sie vorher bereits wusste, dass Peter es nicht wissen kann. Peter antwortet: “Wirklich? Dann weiss ich jetzt die zu ratenden Zahlen.” Daraufhin Susan: “Jetzt weiss ich sie auch.” Frage: Welche zwei Zahlen hat sich Mister X ausgedacht? (Lösung siehe Index)

ADORA-IB Lösung des Mister-X-Problems

```
constraint MisterX
on ( model : Model )
is  ( there-exists pair : Pair in model.pairs() )
        predicate SusanAndPeter( pair, model )
ifTrue    ()
ifFalse   ()
ifError   ()
end

constraint SusanAndPeter
on ( pair : Pair; model : Model )
is  ((( predicate Peter1( pair, model )
        and   predicate Susan1( pair, model ))
        and   predicate Peter2( pair, model ))
        and   predicate Susan2( pair, model ))
ifTrue    ( System.out.println( pair ); )
ifFalse   ()
ifError   ()
end

constraint Peter1
on ( pair : Pair; model : Model )
is  ( model.pairsForProduct( pair.product() ).size() >= 2 )
ifTrue    ()
ifFalse   ()
ifError   ()
end

constraint Susan1
on ( pair : Pair; model : Model )
is  (( model.pairsForSum( pair.sum() ).size() >= 2 )
    and
    ( for-all pairForSum : Pair in model.pairsForSum(pair.sum()) )
        predicate Peter1( pairForSum, model) )
ifTrue    ()
ifFalse   ()
ifError   ()
end
```

```
constraint Peter2
on ( pair : Pair; model : Model )
is  (
    ( there-exists pairForProd : Pair in model.pairsForProduct( pair.product() ) )
        predicate Susan1( pairForProd, model )
    and
    ( for-all pair1 : Pair in model.pairsForProduct( pair.product() ) )
        ( for-all pair2 : Pair in model.pairsForProduct( pair.product() ) )
            ( ( predicate Susan1( pair1, model ) and predicate Susan1( pair2, model ) )
            implies
            ( pair1 = pair2 ))
    )
ifTrue    ()
ifFalse   ()
ifError   ()
end

constraint Susan2
on ( pair : Pair; model : Model )
is  (
    ( there-exists pairForSum : Pair in model.pairsForSum( pair.sum() ) )
        predicate Peter2( pairForSum, model )
    and
    ( for-all pair1 : Pair in model.pairsForSum( pair.sum() ) )
        ( for-all pair2 : Pair in model.pairsForSum( pair.sum() ) )
            (( predicate Peter2( pair1, model ) and predicate Peter2( pair2, model ) )
            implies
            ( pair1 = pair2 ))
    )
ifTrue    ()
ifFalse   ()
ifError   ()
end
```

Anmerkung: Model und Pair sind (Struktur-)Klassen mit folgender Spezifikation:

```
public class Model
    {
    /** Liefert alle Paare von 2 bis 99 */
    static Vector pairs();

    /** Antwortet mit den Paaren, so dass pair.product() = [product].*/
    static Vector pairsForProduct( int product );

    /** Antwortet mit dem Paaren, so dass pair.sum() = [sum].*/
    static Vector pairsForSum( int sum )
    }

public class Pair
    {
    /** Antwortet mit dem Produkt der beiden Zahlen, die das Paar bilden. */
    public int product()

    /** Antwortet mit der Summe der beiden Zahlen, die das Paar bilden. */
    public int sum()
    }
```

F Abkürzungen

Abb.	Abbildung
etc.	et cetera
ggf.	gegebenenfalls
i.d.R.	in der Regel
inkl.	inklusive
o.ä.	oder ähnlilches
o.B.d.A.	ohne Beschränkung der Allgemeinheit
o.g.	oben genannt
Tab.	Tabelle
usw.	und so weiter
u.U.	unter Umständen

G Referenzen

[Alle82] Allen, R. B. (1982): Patterns of Manuscript Revisions. *Behaviour and Information Technology,* Vol. 1; 1982. (pp.177-184)

[Aren96] Arena® Simulation-Tool: Systems Modeling Corporation, Sewickely PA, http://www.sm.com/simcontrol.htm; Nov. 1996.

[Bea+90] Beard, D. V., Walker II, J. Q. (1990): Navigational techniques to improve the display of large two-dimensional spaces. *Behaviour & Information Technology,* Vol. 9, No. 6; 1990. (pp. 451-466)

[Be+99a] Berner, S., N. Schett, N., Xia Y., Glinz M. (1999): *An Experimental Validation of the ADORA Language*. Technischer Bericht 99-07. Institut für Informatik der Universität Zurich; 1999.

[Be+99b] Berner, S., Glinz, M., Joos, S. (1999): A Classification of Stereotypes for Object-Oriented Modeling Languages. *Proceedings of the Second International Conference on the Unified Modeling Language*. Fort Collins, Colorado. Berlin, etc.: Springer; Okt. 1999. (pp. 249-264)

[Be+98a] Berner, S., Arnold, M., Joos., S., Glinz, M. (1998): *Visualizing ADORA Models*. Technischer Bericht 98-08. Institut für Informatik der Universität Zürich; 1998.

[Be+98b] Berner, S., Arnold, M., Joos., S., Glinz, M. (1998): A Visualization Concept for Hierarchical Object Models. *Proceedings of the 13th IEEE International Automated Software Engineering Conference (ASE 1998)*. Honolulu, Hawaii. Washington, etc.: IEEE Computer Society; Oct. 1998. (pp. 225-228)

[Booc94] Booch, G.: *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc.; 1994.

[Broo87] Brooks, F.P. (1987): No Silver Bullet – Essence and Accidents of Software Engineering. *IEEE Computer*, Vol 20, No. 4, April; 1987. (pp. 10-19)

[Bus+96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996): *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY, etc.: John Wiley & Sons, Inc.; (1996).

[Char94] Charwat, H.J. (1994): *Lexikon der Mensch-Maschine-Kommunikation*. 2. Auflage, München: Oldenburg; 1994.

[Chi+88] Chi, M. T. H., Glaser, R., Farr, M. J. (1988): *The Nature of Expertise*. Hillsdale, NJ: Erlbaum; 1988.

[Chi+91] Chimera, J., Wolman, K., Mark, S., Shneiderman, B. (1991): *Evaluation of three interfaces for browsing hierarchical tables of contents*. Technical Report CAR-TR-539, CS-TR-2620. University of Maryland, College Park; Feb. 1991.

[Chen76] Chen, P.P. (1976): The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, Vol. 1 (1); 1990. (pp. 9-36)

[Coo+99] Cook, S., Kleppe, A., Mitchell, R., Warner, J., Wills, A. (1999): Defining the Context of OCL Expressions. *Proceedings of the Second International Conference on the Unified Modeling Language*. Fort Collins, Colorado. Berlin, etc.: Springer; Okt. 1999.

[Dit+96] Dittrich, K.R., Gatziu, S. (1996): Aktive Datenbanksysteme, Konzepte und Mechanismen. *Thomson's Aktuelle Tutorien*, TAT 33; 1996.

[Dit+95] Dittrich, K. R., Gatziu, S. (1995): *Events in an Active, Object-Oriented Database System*. Hamburg: Verlag Dr. Kovac; 1995.

[Fein88] Feiner, S. (1988): Seeing the Forest for the Trees: Hierarchical Display of Hypertext Structure. *Proceedings of the Conference on Office Information Systems 1988*. Palo Alto; Mar 1988. (pp. 205-212)

[Fer+94] Ferstl, O. K., E. Sinz: *SOM Version 2.10*. Universität Bamberg, Lehrstuhl für Wirtschaftsinformatik; 1994.

[Fir+98] Firesmith, D., Henderson-Sellers, B. H., Graham, I., Page-Jones, M. (1998): *Open Modeling Language (OML) – Reference Manual*. SIGS reference library series. Cambridge, etc.: Cambridge University Press.

[Fru+91] Frühauf, K., Ludewig, J, Sandmayr, H. (1991): *Software-Projektmanagement und -Qualitätssicherung*. Stuttgart: Teubner; 1991. (pp. 93)

[Furn81] Furnas, G. W. (1981): *The fisheye view: a new look at structured files*. Technical Report, Bell Labs; 1981.

[Furn86] Furnas, G. W. (1986): Generalized fisheye views. *Proceedings of ACM CHI 86 Conference on Human Factors in Computing Systems*, Boston, Mass., Apr. 13-17, ACM Press, New York; 1986. (pp. 16-23)

[Gam+95] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995): *Design Patterns: Elements Of Reusable Object-Oriented Software*. Reading, Mass., etc.: Addison-Wesley; 1995.

[Gas+95] Gaskell, C., R. Phillips: Software Architecture of the Executable Graphical Specification Tool EGS1. *Software-Concepts and Tools* (1995) 16. (pp. 124-135)

[Glin93] Glinz, M. (1993): Hierarchische Verhaltensbeschreibung in objektorientierten Systemmodellen – eine Grundlage für modellbasiertes Prototyping. In H. Züllighoven, W. Altmann, E. H. Doberkat (Hrsg.), *Requirements Engineering 1993: Prototyping*, Bonn, Teubner Stuttgart. (pp. 175-192).

[Gli+01] Glinz, M., S. Berner, S. Joos, J. Ryser, N. Schett, Y. Xia, (2001). The ADORA Approach to Object-Oriented Modeling of Software. In K.R. Dittrich, A. Geppert and M.C. Norrie (eds.): *Advanced Information Systems Engineering, Proceedings of CAiSE 2001*, Interlaken, Switzerland. Lecture Notes in Computer Science Vol. 2068. Berlin: Springer.

[Gos+96] Gosling, J., Joy, B., Steele, G. (1996): *The Java Language Specification*. Reading, Mass., etc.: Addison-Wesley; 1996.

[Herc94] Herczeg, M. (1994): *Software Ergonomie – Grundlagen der Mensch-Computer–Kommunikation*. Bonn: Addison-Wesley; 1994.

[Hare87] Harel, D. (1987): Statecharts: *A Visual Formalism for Complex Systems*. Science of Computer Programming, Vol. 8; 1987. (pp. 231-274)

[Hol+89] Hollands, J. G., et al. (1989): Presenting a graphical Network: A Computing of Performance Using Fisheye and Scrolling Views. *Proceedings of the third International Conference on Human-Computer-Interaction*. Boston; Sept. 1989. (pp. 313-320)

[Hop+88] Hopcroft, J.E., Ullmann, J.D. (1988): *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie*. Reading, Mass.: Addison-Wesley; 1988.

[IBM+97] IBM et. al. (1997): *OCL Object Constraint Language Specification*; Version 1.1. http://www.software.ibm.com/ac/oc/; Aug. 1997.

[Ilg+87] Ilg, R., Ziegler, J. E. (1988): Interaktionstechniken. In K.-P. Fähnrich (ed.), *Software Ergonomie – State of the Art5*. Oldenburg; 1987. (pp. 106-107)

[Ilg+88] Ilg, R., Ziegler, J. E. (1988): Direkte Manipulation. In H. Balzert (ed.), *Einführung in die Software Ergonomie, Mensch Computer Kommunikation, Grundwissen 1*. Berlin, New York: de Gruyter; 1988. (pp. 175-194)

[Inno96] Innovator® CASE Workbench: MID GmbH, Nürnberg; 1996.

[Joh+91] Johnson, B., Shneiderman, B. (1991): Tree-maps: A space-filling approach to the visualization of hierarchical information structures. *Proceedings of IEEE Visualization '91*. San Diego, CA. Oct 1991. (pp.284-291)

[Joo+97] Joos, S., S. Berner, M. Glinz (1997): Hierarchische Zerlegung in objektorientierten Spezifikationsmodellen. *Softwaretechnik-Trends*, 17, 1; Feb. 1997. (pp. 29-37)

[Joos99] Joos, S. (1999): *ADORA-L – Eine Modellierungssprache zur Spezifikation von Software-Anforderungen*. Dissertation, Universität Zürich; 1999.

[Kad+78] Kadmon, N., Shlomi, E. (1978): A polyfocal projection for statistical surfaces. *The Cartographic Journal*, Vol. 15, No. 1; 1978. (pp. 36-41)

[Kaen96] Von Känel, R. (1996): *Visualisierungskonzepte von Teil-Ganzes-Hierarchien in objektorientierten Spezifikationsmodellen*. Diplomarbeit, Institut für Informatik der Universität Zürich; 1996.

[Kal+91] Kaltenbach, M., Robillard, F. and Frasson, C. (1991): Screen Management in Hypertext Systems with Rubber Sheet Layouts. *Proceedings of Hypertext '91*. San Antonio; Dec 1991. (pp. 91-105)

[Keah97] Keahey, A. T. (1997): *Nonlinear Magnification*. PhD. Thesis at the Department of Computer Science of the Indiana University; 1997.

[Keah98] Keahey, A.T. (1998): The Generalized Detail-In-Context Problem. *Proceedings of IEEE Visualization '98, Information Visualization Symposium*. Research Triangle Park, NC. Washington, etc.: IEEE Computer Society; Oct. 1998. (pp. 44-51)

[Kel+95] Keller, R. K. et al. (1995): Environment Support for Business Reengineering: The Macrotec Approach. *Software-Concepts and Tools.* (1995) 16; 1995 (pp. 31-40)

[Koe+96] Kölzer, A., I. Uhe (1996): *Benutzerhandbuch für das KOGGE-Tool BONSAI*. Interner Arbeitsbericht 4/96 der Univ. Koblenz, Inst. für Softwaretechnik; 1996. http://www.uni-koblenz.de/~ist/p_kogge.html; Nov. 1996.

[Koik95] Koike, H. (1995): Fractal Views: A Fractal-Based Method for Controlling Information Display. *ACM Transactions on Information Systems*, Vol. 13, No. 3; July 1995. (pp. 305-323)

[Leu+94] Leung, Y. K., Apperley, M. D. (1994): A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction*, Vol. 1, No. 2; June 1994. (pp. 126-160)

[Leun89] Leung, Y. K. (1989): Human-Computer Interface Techniques for Map based Diagrams (sic!). *Proceedings of the third International Conference on Human-Computer Interaction*. Boston; Sept 89. (pp. 361-368)

[Lic+93] Lichter, H., Schneider, K. (1993): vis-A-vis: Ein Objektorientiertes Applikation Framework für graphische Entwurfswerkzeuge. In Mayr, H.C. (ed.) / Wagner R. (ed.), *Objektorientierte Methoden für Informationssysteme, Informatik Aktuell*. Berlin, etc.: Springer; 1993. (pp. 187-207)

[Lich93] Lichter, H. (1993): *Entwicklung und Umsetzung von Architektur-Prototypen für Anwendungssoftware.* Dissertation, Universität Stuttgart. Zürich: vdf, Verlag der Fachvereine; (1993).

[Lude89] Ludewig, J. (1989): Modelle der Software-Entwicklung - Abbilder oder Vorbilder? *Softwaretechnik-Trends*, Oktober 1988

[Luft84] Luft, A. L. (1984): Zur Bedeutung von Modellen und Modellierungs-Schritten in der Softwaretechnik. *Angewandte Informatik*, Band 26, Heft 5; Mai 1984. (pp. 189-196)

[Mac+91] Mackinlay, J. D., Robertson, G. G. and Card, S. K. (1991): The perspective wall: detail and context smoothly integrated. *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*. New Orleans, Louisiana. ACM Press; Apr/May 1991. (pp. 173-179)

[Micr96] MicroSaint® Simulation-Tool: Micro Analysis & Design Inc., Boulder Colorado, http://www.madboulder.com/; Nov. 1996.

[Mit+97] Mitchell, K., Kennedy, J. (1997): The perspective tunnel: An inside view on smoothly integrating detail and context. *Eurographics Workshop on Visualization in Scientific Computing*; April 1997.

[Mös+97] Mössenböck, H. P., Sikora, H. (1997): Es muss nicht immer ASCII sein: Aktive Texte in der Praxis und im Unterricht. *Softwaretechnik Trends*, 17, 1; 1997.

[Noik93] Noik, E. G. (1993). Exploring large hyperdocuments: Fisheye views of nested networks. *Proceedings of the ACM Conference on Hypertext and Hypermedia*. Seattle, WA. ACM Press; Nov 93 (pp. 192-195)

[ObjT96] ObjecTime® Toolset: ObjectTime Limited, Kanada, Ontario/Canada, http://www.objectime.on.ca/ObjecTimeProduct.html; Nov. 1996.

[ObjY96] Objectory® products: Rational Software Corporation, Santa Clara CA, http://www.rational.com/pst/products/obj_overview.html, Nov. 1996.

[Petr95] Petre, M. (1995): Why looking isn't always seeing – Readership Skills and Graphical Programming. *Communications of the ACM*, Vol. 38, No. 6; June 1995. (pp. 33-43)

[Pet+93] Petre, M., Green, T. R. G. (1993): Learning to read graphics: some evidence that 'seeing' an information display is an aquired skill. *Journal of Visual Languages and Computing*, 4; 1993. (pp. 55-70)

[Rao+94] Rao, R., Card, S. K. (1994): The Table Lens: Merging graphical and symbolic representations in an interactive focus+context visualization for tabular information. *Proceedings of the ACM CHI'94 Conference on Human Factors in Computing Systems*. Boston. ACM Press; April 1994. (pp. 318-322)

[Rem+87] Remde, J. R., Gomez, L. M., Landauer, T. K. (1987): SuperBook: An Automatic Tool for Information Exploration - Hypertext? *Proceedings of the ACM Hypertext '87 Conference*. Chapel Hill, NC.; Nov. 1987. (pp. 175-188)

[Rob+91] Robertson, G. G., Mackinlay, J. D., Card, S. K. (1991): Cone Trees – Animated 3D Visualization of Hierarchical Information. *Proceedings of the ACM CHI'91 Conference on Human Factors in Computing Systems*. New Orleans. ACM Press; Apr. 1991. (pp. 189-194)

[Rob+93] Robertson, G. G., Mackinlay (1993): The document lens. *Proceedings of the ACM Symposium on User Interface Software and Technology.* ACM Press; 1993. (pp. 189-194)

[Rose96] Rational Rose® tools: Rational Software Corporation, Santa Clara CA, http://www.rational.com/pst/products/rosefamily.html; Nov. 1996.

[Rum+99] Rumbaugh, J., Jacobson, I., Booch, G. (1999): *The Unified Modeling Language Reference Manual*. Reading, Mass., etc.: Addison-Wesley.

[Rum+91] Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*. Englewood Cliffs, N. J.: Prentice Hall; 1991.

[Rund96] Rundshagen, M. (1995): *Computergestützte Konsistenzsicherung in der objektorientierten Systemanalyse*. Heidelberg: Physica Verlag; 1996.

[Sar+92] Sarkar, M., M. H. Brown (1992): Graphical Fisheye Views of Graphs. *Proceedings of ACM CHI 92 Conference on Human Factors in Computing Systems*, ACM Press, New York; 1992. (pp. 83-91)

[Sar+93] Sarkar, M., Snibbe, S., Tversky, O. and Reiss, S. (1993): Stretching the Rubber Sheet: A Metaphor for Visualizing Large Layouts on Small Screens. *Proceedings of UIST '93*. Atlanta; Nov 1993. (pp. 81-91)

[Sc+93a] Schaffer, D., Zuo, Z., Bartram, L., Dill, J., Dubs, S., Greenberg, S. and Roseman, M. (1993): Comparing Fisheye and Full-Zoom Techniques für Navigation of Hierarchically Clustered Networks. *Proceedings of Graphics Interface '93 (GI'93)*. Morgan-Kaufmann; 1993. (pp. 86-96)

[Sc+93b] Schaffer, D., Greenberg, S. (1993): Sifting through hierarchical information. *Proceedings of posters and short papers of the ACM INTERCHI '93 Conference on Human Factors in Computing Systems*. Amsterdam, Holland; 1993

[Sch+96] Schaffer, D., et al. (1996): Navigating Hierarchically Clustered Networks through Fisheye and Full-Zoom Methods. *ACM Transactions on Computer-Human Interaction*, Vol. 3, No. 2; Jun. 1996. (pp. 162-188)

[Sche98] Schett, N. (1998): *Konzeption & Realisierung einer Notation zur Formulierung von Integritätsbedingungen für ADORA-Modelle*. Diplomarbeit. Institut für Informatik der Universität Zürich; 1998.

[Scho95] Schöning, U. (1995): *Logik für Informatiker.* Oxford, etc.: Spektrum, Akad. Verlag; 1995.

[Schw95] Schwille, J. (1995): *Dokumenten- und Prozessmodelle für die Software-Verwaltung*. Dissertation, Universität Stuttgart. Hamburg: Verlag Dr. Kovac; 1995.

[Sedg92] Sedgewick, R. (1992): *Algorithms*, 3rd ed. Reading. Mass., etc.: Addision Wesley; 1992.

[Sel+94] Selic, B., G. Gullekson, P. T. Ward: *Real-Time Object-Oriented Modelling*. John Wiley & Sons; 1994.

[Spe+82] Spence, R., Apperley., M. (1982): Data-base navigation: An office environment for the professional. *Behavior and Information Technology*, Vol. 1, No. 1; 1982. (pp. 43-54)

[Stac73] Stachowiak, H. (1973): *Allgemeine Modelltheorie*. Wien: Springer-Verlag; 1973.

[Stat96] Statemate® statechart tool: i-Logix Inc., Andover MA, http://www.ilogix.com/; Nov. 1996.

[StP96] Software through Pictures® tools: Interactive Development Environments Inc. / AONIX, San Francisco CA, http://www.ide.com/; Nov. 1996.

[SysS96] SystemSpecs® petrinet-Tool: TnTech AG, CH-3018 Bern, http://www.thenet.ch/tntech/sysspecs.html; Nov. 1996.

[Wieb90] Wiebe, D. (1990): *Generic Software Configuration Management; Theory and Design*. University of Washington, Department of Computer Science; 1990.

[Zeh+89] Zehnder, C. A. (1989): *Informationssysteme und Datenbanken*; 5. Auflage. Stuttgart: vdf und B.G. Teubner; 1989.

H INDEX

NUMERICS

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Lebenslauf

Name:	Stefan Berner
Geburtsdatum:	1. März 1966
Staatsangehörigkeit:	deutsch
Geburtsort:	Stuttgart
e-mail:	berner@ifi.unizh.ch

ab 1972	Besuch der Grundschule in Stuttgart-Sillenbuch
ab 1977	Besuch des Geschwister-Scholl-Gymnasiums in Stuttgart-Sillenbuch
ab Juli 1986	Grundwehrdienst
ab Oktober 1986	Zivildienst
ab März 1988	Praktikum bei der Datenzentrale Baden-Württemberg
ab Oktober 1988	Studium der Informatik an der Universität Stuttgart
ab November 1994	Assistent, Institut für Informatik der Universität Zürich
ab April 2000	Software-Architekt, FJA Feilmeier & Junker AG
ab Mai 2002	Consultant, IBM Schweiz