Visualisierungsverfahren zur Interaktion mit objekt-relationalen Datenbanken

von
Dipl. Inf. Jens Thamm

Dissertation
vorgelegt am Fachbereich Mathematik/Informatik
der Universität Gesamthochschule Kassel
zur Erlangung des Doktorgrades der Naturwissenschaften
(Dr. rer. nat.)

Kassel 1999

Hiermit versichere ich, daß ich die vorliegende Dissertation selbständig und ohne unerlaubte Hilfe angefertigt und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Vom Fachbereich Mathematik/Informatik der Universität Gesamthochschule Kassel als Dissertation angenommen am 16. August 1999.

Erstgutachter: Prof. Dr. L. Wegner, Universität Gesamthochschule Kassel
Zweigutachter: Prof. Dr. K. Meyer-Wegener, Universität Dresden

Tag der mündlichen Prüfung: 27. September 1999

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Arbeit stellt ein neues, erweiterbares Visualisierungsverfahren zur Interaktion mit objekt-relationalen Datenbanken vor. Sie beschränkt sich dabei nicht nur auf die Konzeption der Visualisierungstechniken, sondern demonstriert in zahlreichen, vollständigen und implementierungsnahen Beispielen mögliche Realisierungen und zeigt die von ihnen erzeugten graphischen Ausgaben.

Die Arbeit will damit den Nachweis antreten, daß die Techniken der Selbstreferenzierung, der Speicherung von Visualisierungs-Meta-Daten in Tabellen und des Skriptings, bzw. deren Kombination in dem vorgestellten Verfahren, offen sind für Erweiterungen durch Anwender, z. B. Datenbankprogrammierer oder Gestalter graphischer Benutzerschnittstellen.

Sie soll auch belegen, daß dieser Ansatz gegenüber existierenden, geschlossenen Methoden der Gestaltung graphischer Benutzerschnittstellen (*graphical user interfaces*, GUIs) erhebliche Vorteile hat. Diese liegen in der Schnelligkeit und Leichtigkeit der Erstellung der Schnittstellen und der Anpassung an neue Anwendungen. Dabei weist der Ansatz eine ausreichende Effizienz bei der Generierung der visuellen Darstellung zur Laufzeit auf.

Obwohl die Codierungsbeispiele hier auf Ousterhouts Skriptsprache Tcl/Tk [Ous94] und deren Erweiterung Tcl/DB [Ahm98] basieren, ist die Realisierung der vorgestellten Konzepte nicht daran gebunden. Sie kann genauso gut in jeder anderen Umgebung vorgenommen werden, die eine interpretierte Sprache zur Verfügung stellt, mit der sowohl die Objekte der graphischen Benutzerschnittstelle als auch die strukturierten Datenobjekte manipuliert werden können. Die Modellierung von Datenstrukturen und Algorithmen ist hier in einer objekt-orientierten Sichtweise ausgeführt, die aber ebenfalls keine notwendige Anforderung ist. So wurde zum Nachweis der Realisierbarkeit eine prototypische Implementierung in Tcl/Tk vorgenommen. Dabei konnte auf die jüngeren, rein objekt-orientierten Erweiterungen von Tcl/Tk[1], [INCR TCL] und [INCR TK] [Har97], verzichtet werden. Die vorgestellten Algorithmen sind demnach allgemeingültig.

[1] Tcl/Tk und Tcl/DB beinhalten bereits objekt-orientierte Merkmale, z. B. die methodenartigen Klassenkommandos.

1.1 Interaktion mit Datenbanken

Unter einer Schnittstelle wird im Folgenden eine Mensch-Maschine-Benutzungsschnittstelle (im Englischen *man-machine-interface* (MMI), im Deutschen meist Benutzerschnittstelle)[2] verstanden.

Diese Schnittstelle erlaubt es dem Menschen, auf einem Rechner laufende Programme, auch Anwendungen oder Applikationen genannt, interaktiv zu beeinflussen. Die Erkenntnis, daß die Sprachen, Formen der Darstellung und Protokolle des Dialogs dieser Schnittstellen ein eigenes, wichtiges Forschungs- und Entwicklungsgebiet sind, stammt aus den siebziger Jahren. Sie ist eng verknüpft mit dem Aufkommen der graphischen Bildschirmausgabe. Speziell zu nennen sind die Arbeiten im Xerox Palo Alto Forschungszentrum (Palo Alto Research Center, PARC), etwa die Entwicklungen für den Xerox Star [JRV+89]. Diese führte später zur Entwicklung des Mac-Rechners bei Apple, der als Produkt erstmals eine Schnittstelle mit Fenstern und Icons präsentierte. Auf die Problematik der raschen und fehlerfreien Erzeugung ansprechender graphischer Benutzerschnittstellen wird in Abschnitt 2.2 eingegangen.

In den Mittachtzigern entwickelte sich daraus der Begriff *human-computer-interaction* (HCI und CHI) [Pre94, S. 7]. Die ACM Special Interest Group on CHI (SIGCHI) definiert dies so: „human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them“ [ACM92, S. 6]

Die vorliegende Arbeit beschäftigt sich mit speziellen Schnittstellen, nämlich Schnittstellen zu Datenbanken. Auf die besondere Art der Datenbanken, die dabei betrachtet werden, wird in Kapitel 2 eingegangen.

Hier soll zunächst die Notwendigkeit der Beschäftigung mit Schnittstellen für Datenbanken herausgestellt werden. Spaccapietra brachte dies 1994 auf den Punkt: „User Interfaces; Who Cares?“ [Spa94]. Er verweist darauf, daß bei allen Befragungen von Datenbankexperten nach vordringlichen Themen für Forschung und Entwicklung, eine Mehrheit der Befragten immer wieder „Benutzerschnittstellen“ (*user interfaces*) an erster Stelle nennt. Im Gegensatz zu diesem Wunsch steht dann die von Spaccapietra monierte Tatsache, daß sich kaum eine Datenbankarbeit mit Benutzerschnittstellen beschäftigt, auf der angesprochenen Konferenz VLDB 1994 in Chile z. B. gar keine.

Offensichtlich bezieht sich Spaccapietra dabei auf einen Beitrag von Stonebraker, Agrawal, Dayal, Neuhold und Reuter von 1993: „DBMS Research at a Crossroads: The Vienna Update“ [SAD+93]. Darin wird wiederum auf die sog. „Laguna Beach Befragung“ [BDD+89] verwiesen, nach der 1989 Benutzerschnittstellen und aktive Datenbanken als die wichtigsten Forschungsziele genannt wurden. Die Wiener Befragung hat die Benutzerschnittstellen erneut an die Spitze gesetzt. „The panelists were enthusiastic about new user interfaces,

[2] Laut DIN sind diese Schnittstellen Benutzungsschnittstellen, eingebürgert hat sich aber der Begriff Benutzerschnittstellen. Er soll auch hier verwendet werden.

such as workflow languages and collaboration tools. They lamented that progress in this area continues to be done by industry, and the research community has very little impact on this important topic.“ [SAD+93, S. 689].

An dieser Einschätzung hat sich bis heute nichts geändert: die Tagungsbände von drei der wichtigsten Datenbankkonferenzen, VLDB, SIGMOD und PODS, enthalten in den Jahren 1995–1999 zusammen rund 1000 Artikel und Berichte zu Vorführungen. Davon beschäftigen sich höchstens 10 mit Benutzerschnittstellen!

Dies ist aus mehrerlei Hinsicht ärgerlich. Zum einen verlangen gerade die neueren Arten von Datenbanken, wie z. B. die objekt-orientierten Systeme (siehe Abschnitt 2.1.2), nach Ad-hoc-Anfragesystemen, die einfach und intuitiv benutzbar sind („Thou shall have a simple way of querying data“ [ABJ+90, S. 234]).

Zum zweiten haben sich die Anwenderkreise für Datenbanken völlig verändert. Früher wurden Datenbanken im wesentlichen für Online-Zugriffe großer Verwaltungen eingesetzt. Anwender greifen dabei auf die Daten über spezielle Anwendungsprogramme zu, die Daten aus der Datenbank extrahieren und in Form von Masken anbieten. Ad-hoc-Anfragen und das Stöbern in Datenbanken sind in dieser Umgebung eher selten und Spezialisten vorbehalten. Die vorherrschenden Datentypen sind alpha-numerischer Art.

Im Gegensatz dazu werden ab Anfang der neunziger Jahre Datensammlungen auch auf CD-ROM und über das Internet angeboten. Die Anwender sind DV-Laien und greifen wieder über spezielle Benutzerschnittstellen auf die Datenbank zu. Weiterhin werden alternative Zugriffsmöglichkeiten angeboten, wie z. B. das Suchen (Stichwortsuche) und insbesondere das Stöbern (*browse*). Die Informationen werden nicht mehr nur alpha-numerisch angeboten, sondern enthalten auch Graphiken, bewegte Bilder (Animationen und Videosequenzen) sowie Audiodaten. Häufig werden die Informationen in einer Sammlung von HTML-Seiten auf einem WWW-Server angebotenen. Dieser kann seinerseits auf Informationen aus Datenbanken zurückgreifen, bzw. die HTML-Seiten können mit Informationen aus Datenbanken generiert worden sein.

Die Datenautobahn (*information highway*) geht demnach mitten durchs Wohnzimmer, wie es Silberschatz, Stonebraker und Ullman ausdrücken: „The information superhighway just rolled through your living room“ [SSU96, S. 56]. In Anlehnung an den Begriff der Datenautobahn fragt auch DeWitt 1995, ob Datenbanken bei dieser Entwicklung nicht unter die Räder geraten („Database Systems: Roadkill on the Information Superhighway?“ [DGN95, S. xi; Invited Talk]), weil sie sich mit ihren inflexiblen Datentypen und eingeschränkten Anfragemöglichkeiten nicht an die neuen Formen der Informationsverarbeitung anpassen.

Konkret verbergen sich dahinter mehrere Phänomene. Dazu gehört die Rückkehr der Navigation, die in der Datenbankwelt durch den Übergang von den hierarchischen und Netzwerkmodellen zu den relationalen Datenbanken eigentlich überwunden war. Für den „naiven“ Anwender ist es oft leichter, sich navigierend in der Datenlandschaft zu bewegen und sich zum Ort des Informationsangebotes (virtuell) hin zu bewegen. Es wird dagegen als unnatürlich empfunden, die Information aus mehreren Quellen zusammenzusetzen und zu

filtern, wie das in relationalen Datenbanken durch fortschreitende Projektion und Selektion mit Joins zwischen mehreren Tabellen üblich ist.

Ein Beispiel dafür ist die Suche nach der Information, ob in einem vorgegebenen Zeitraum eine Zugverbindung existiert. In einem SQL-basierten System könnte eine Anfrage wie folgt aussehen:

```
SELECT Abfahrt.An AS Abfahrt, Ankunft.An AS Ankunft, ...
  FROM Züge, Zugläufe Abfahrt, Zugläufe Ankunft,
       Bahnhöfe Bhf1, Bahnhöfe Bhf2
 WHERE Ankunft.Zug = Abfahrt.Zug
   AND Abfahrt.Bhf = Bhf1.Nr
   AND Ankunft.Bhf = Bhf2.Nr
   AND Züge.Nr = Abfahrt.Zug
   AND Züge.Art = 'ICE'
   AND Abfahrt.An > 16:00 AND Abfahrt.An < 17:00
   AND Bhf1.Ort = 'Kassel'
   AND Bhf2.Ort = 'Dresden'
   AND ...
```

In einer menü-geführten Anfrage müßte man die Stadt, das Ziel, den Tag, die Uhrzeit(en), den Zugtyp, usw. eingeben oder zumindest auswählen. Diese Eingaben würden dann in die Anfrage eingesetzt werden. In einem virtuellen Bahnhof könnte die Information durch Navigation in der virtuellen Abfahrtstafel einer Stadt bestimmt werden.

Weitere Phänomene sind die schwache Strukturierung der Daten (schemalose Informationsangebote), Redundanzen, Zyklen in Verweisketten, eine Vielzahl an Datentypen, usw. Auf Anwenderseite ist das herausstechende Merkmal die Sprachlosigkeit des Anfragers. Natürlichsprachliche Zugänge („Wann fährt denn der ICE von Kassel nach Dresden, zwischen vier am Nachmittag und fünfe, sonntags?“) sind noch nicht machbar. Eine weitgehend formalisierte, von allen universell beherrschte Informations- und Kommandosprache gibt es nicht und wird es wegen des hohen Anspruchs vermutlich auch nie geben[3].

An die Stelle der sprachlichen Kommunikation ist demnach deiktisches Handeln [BHPR97] getreten. Dem Zeigen auf Dinge mit dem Finger oder einem Zeigestock entsprechen an der Benutzerschnittstelle das Bewegen des Cursors, das Anklicken, die Navigation mit Weiter- und Zurück-Tasten, das Blättern in Analogie zu einem Buch oder das Rollen in der Schriftrollen-Metapher. Zum Thema „Benutzerschnittstellen“ gibt es eine praktisch unerschöpfliche Anzahl an Literaturstellen. Allein die WWW-Seiten `http://www.aw.com/DTUI` zum Buch „Designing the User Interface“ von Ben Shneiderman [Shn92] enthalten über 500 Querverweise zu anderen Projekten, WWW-Servern und Publikationen.

[3] Anzumerken ist, daß es für einzelne Anwendungsgebiete durchaus spezialisierte, formalisierte Sprachen gibt, die universell akzeptiert sind. So hatten bereits die Armeen des 18. und 19. Jahrhunderts wegen der heterogenen Vielvölkerzusammensetzungen der Soldaten Kommandosprachen mit ca. 30 ausgewählten Kommandos entwickelt, die jeder Rekrut zu lernen hatte.

Die oben gemachte Aussage, daß die zentralen Publikationen der Datenbankforscher sich nicht genügend mit Benutzerschnittstellen beschäftigen, steht demnach in deutlichem Widerspruch zur enormen Bedeutung dieser Form des Informationszugangs und zu der großen Zahl an Veröffentlichungen zum Thema Benutzerschnittstellen. In der Tat wird man erst fündig, wenn man in eher speziellen Publikationsreihen gezielt nach dem Begriff „Benutzerschnittstellen für Datenbanken“ sucht.

Zu diesen gehören z. B. die Tagungsbände der Konferenzen „Visual Database Systems“ (VDB: Tokyo 1989, Budapest 1992, Lausane 1995, L'Aquila 1998, Fukuoka 2000) und „Advanced Visual Interfaces“ (AVI, zwei-jährlich seit 1992 in Italien), das seit 1989 bestehende „Journal of Visual Languages and Computing“, die „IEEE Multimedia“ Zeitschrift und die Publikationen der ACM SIGCHI.

Der jüngste „große Überblick“ zu Benutzerschnittstellen für Datenbanken in der einschlägigen Datenbankliteratur ist 1992 als Hauptthema zu fortschrittlichen Benutzerschnittstellen für Datenbanksysteme in der Zeitschrift ACM SIGMOD Record (Band 21, Nummer 1, März 1992, S. 4–64: „Advanced User Interfaces for Database Systems“) veröffentlicht und damit schon recht alt. Ioannidis schreibt in seiner Einführung [Ioa92], daß ein Wandel der erfolgsbestimmenden Faktoren für Datenbanksysteme stattgefunden hat. Zwar seien die drei wichtigsten Faktoren immer noch „performance, performance, performance!“, aber zumindest ein Performance-Faktor müsse inzwischen bzgl. Benutzerproduktivität interpretiert werden. Aus der Ungeeignetheit von Datenbanksprachen wie SQL für viele Benutzer folgert er: „The need for more intuitive and easier to learn and use interfaces to database systems is always current.“

Dieser Zeitraum um 1992 herum markiert einen gewissen Höhepunkt in der Entwicklung. In Deutschland liegt dieser eher etwas später. Ein Indiz dafür ist die 1994 von Wegner für die Fachgruppe 2.5.1 „Datenbanken“ der Gesellschaft für Informatik in Kassel veranstaltete Arbeitstagung „Benutzungsschnittstellen für Datenbanksysteme“ [Weg94]. Diese war mit 350 Teilnehmern eine der am besten besuchten Arbeitstagungen, die von der Fachgruppe je veranstaltet wurden.

Der Anfang der neunziger Jahre ist auch deshalb markant für die Entwicklung von graphischen Benutzerschnittstellen, weil sich in dieser Zeit Microsoft Windows stark verbreitet. Die sog. „WIMP-Oberfläche“ (WIMP = *windows + icons + menues + pointers*) bietet sich als Schnittstelle für Datenbankanfragen offensichtlich an. Der, auf der angesprochenen Arbeitstagung vorgestellte, Oracle Data Browser ist ein typischer Vertreter dieser Generation von Benutzerschnittstellen für Datenbanken [MW94].

Im Folgenden wird eine grobe Klassifizierung von graphischen Benutzerschnittstellen für Datenbanken vorgenommen, um die sehr breit gefächerte Entwicklung mit vielen Veröffentlichungen und einigen Prototypen weiter verfolgen zu können.

Visuelle Anfrageformulierung (*visual query formulation*)

Catarci et al. [CCLB97] geben einen Überblick über visuelle Anfragesysteme (*visual*

query systems, VQSs), die visuelle Repräsentationen von Interessengebieten benutzen, um Anfragen nach verwandter bzw. ähnlicher Information zu stellen. In dem Artikel wird eine Einteilung der VQSs nach visuellen Repräsentationsformen und Interaktionsstrategien, sowie eine Zuordnung zu verschiedenen Benutzergruppen vorgenommen. Visuelle Anfragesysteme werden hier nicht weiter betrachtet; vielmehr soll das Browsen in Anfrageergebnissen im Vordergrund stehen.

Eher abschreckende Beispiele für eine graphenbasierte Darstellung von Anfrageergebnissen in dem VQS G^+/GraphLog sind die Abbildungen in [CCM92].

Inhaltsbasierte Anfragen (*image query, query by content, feature extraction and indexing, content-based search and retrieval*)
Inhaltsbasierte Anfragen nach visuellen Informationen sind solche nach bzw. mit Bildern und Bildinhalten. Der Tagungsband [Leu97] der ersten Konferenz mit dem Thema „Visuelle Informationssysteme“ (*visual information systems*) enthält Beiträge, die eine Übersicht über die Verarbeitung visueller Daten in Informationssystemen geben. Darunter werden dort Anfragen nach bzw. mit Bildern und Bildinhalten sowie Extraktion und Indizierung von Bildmerkmalen verstanden. Die angesprochenen Techniken sind hier nicht relevant. Das in dieser Arbeit vorgestellte Visualisierungsverfahren kann aber zur Konstruktion von Benutzerschnittstellen für solche Informationssysteme benutzt werden, insbesondere weil es erweiterbar ist und die Darstellung von multimedialen Datentypen erlaubt.

Datenvisualisierungen (*data visualization, data maps*)
Unter Datenvisualisierung wird die Visualisierung von dynamischen Prozeßdaten, z. B. von Simulationen oder in Prozeßwarten, durch eine graphische Benutzerschnittstelle verstanden. Datenbanken können in diesem Modell als Schicht zwischen datengenerierendem Prozeß und datenvisualisierender Anwendung gesehen werden, wodurch ein gewisses Maß an Unabhängigkeit zwischen diesen beiden ensteht.

Als gegenständliche (Daten-) Visualisierungen im Sinn von [RS99, S. 90] können auch aus Daten generierte, graphische Darstellungen von Objekten interpretiert werden, z. B ein Konstruktionsgegenstand in einem CAD-System. Das in dieser Arbeit vorgestellte Visualisierungsverfahren unterstützt diese Art von Visualisierungen zwar nicht direkt. In Abschnitt 5.3.3 wird aber beispielhaft gezeigt, daß es durch Definition spezieller Visualisierungsfunktionen entsprechend erweitert werden kann.

Informationsrückgewinnung (*visual information retrieval*)
Die Informationsrückgewinnung mit visuellen Daten ist eng verwandt mit inhaltsbasierte Anfragen, so daß die dort gemachten Aussagen analog gelten. Die Ausgabe der Zeitschrift Communications of the ACM mit dem Hauptthema „visuelles Informationsmanagement“ (Band 40, Nummer 12, Dezember 1997, S. 30–80: „Visual Information Management“) und einer Einleitung dazu von Jain [Jai97] enthält Artikel, die das Finden und Präsentieren von visuellen Daten (Bilder und Film) behandeln.

Formularbasierte Visualisierungen (*forms*)

Das Konzept der Formulare war auch schon vor Aufkommen der graphischen Benutzerschnittstellen verbreitet (z. B. FADS [RS82]), wobei damals die einzelnen Felder eines Formulars zyklisch durchlaufen werden mußten (siehe auch Abschnitt 2.2.7). Formulare sind heute noch, vor allem in pre-relationalen und relationalen Datenbanken, weit verbreitet, werden aber auch in neueren Entwicklungen eingesetzt. In [CB91] wird z. B. ein auf dem OODBS O_2 (siehe Abschnitt 1.2) basierendes System beschrieben, das die interaktive Erstellung formularbasierter Benutzerschnittstellen für Datenbankanwendungen erlaubt.

Dort zeigt sich aber auch, daß die Darstellung hierarchischer Strukturen durch die Schachtelung von Formularen unübersichtlich ist [CB91, S. 437]. In dieser Arbeit wird die Eigenschaft von Formularen, zu jedem Zeitpunkt genau ein Element einer Kollektion von Datensätzen darstellen, als beispielhafte Erweiterung der tabellarischen Darstellung betrachtet und in diese integriert (siehe Abschnitt 5.8.2).

Diagramme in Visualisierungen

Diagramme als abstrakte Visualisierungen, d. h. als Visualisierungen ohne Bindung an physikalische Gegenstände, z. B. Balken-, Torten-, Netzwerk- oder Liniendiagramme, enthalten oft eine Vielzahl von Informationen. „[...] Interpretationen, Such- und Vergleichsoperationen [werden] wesentlich erleichtert [...].“ Aber: „Die hohe Effektivität von [diagrammartigen] Visualisierungen bringt gleichzeitig eine Beschränkung in ihrer Ausdrucksmächtigkeit mit sich.“ [RS99, S. 91] Oft müssen in Diagrammen eingesetzte Symbole bzw. Symboliken, z. B. Farben und Texturen, in einer Legende erklärt werden. Kamps analysiert in [Kam99] systematisch Diagramme als visuelle Repräsentationen von faktischem Wissen.

Visualisierung und Design von Schemata (*schema visualization and modelling*)

Schemata werden meist auf Basis des Entity-Relationship-Modells oder Erweiterungen davon dargestellt (z. B. ENIAM [Cre89]). Haber et al. schlagen eine baumartigen Darstellung vor, wie sie für Schemata verbreitet ist und vergleichen sie in einer tabellarischen Übersicht mit anderen Ansätzen [HIL95, S. 537].

Die Übertragung einer diagramm- oder baumartigen Darstellung, wie sie für Schema- und Typinformation sehr angebracht ist und auch in dieser Arbeit dafür verwendet wird, auf Instanzdaten, ist selten sinnvoll. Die durch die Schemata beschriebenen und meist großen Datenmengen, z. B. sehr umfangreiche Kollektionen, können mit Bäumen oder Graphen schlecht visualisiert werden: Knoten werden zu klein, es entsteht ein Gewirr von Kanten, bzw. Ästen in Bäumen, ein Graph muß stark auseinandergezogen werden, damit er alle Knoten aufnehmen kann. Für textuelle Information muß oft eine kaum lesbare, sehr kleine Schriftart verwendet werden, wenn sie in oder an Knoten dargestellt werden soll.

Trotzdem findet sich dieser Vorschlag immer wieder, z. B. in [Kun92, WMJ+99]. In Kuntz' Beitrag über einen Browser für geschachtelte Relationen werden nur sehr

kleine Objekte visualisiert, im Artikel über einen Browser für Digitale Bibliotheken von Witten et al. ist das Beispiel eher abschreckend [WMJ+99, S. 78].

Daten- und Informationsraum (*data space, information space*)
: Durch eine zu große Fülle von Information oder eine zu schlechte Strukturierung der Information kann beim Benutzer das Gefühl des „Verlorenseins im Informationsraum“ entstehen (*„lost in information space“, [Gou97]*). Hier kann die Aufbereitung der Informationen, z. B. durch Synthese bestimmter Eigenschaften, hilfreich sein. Chen et al. schreiben: „However, even the viewing of a very-well organized set of documents is usually limited by its sequential representation on a computer screen. Such representation does not present the whole picture when users are dealing with large data collections. New ways of representing information [...] are vital to synthesizing diverse data.“ [CNJOT98, S. 78]

  Die Darstellung von Daten in einer streng hierarchischen Struktur wie den geschachtelten Tabellen gibt einem „naiven“ Benutzer eine wesentlich bessere Übersicht eines Datenobjekts und seiner Sub-Objekte als z. B. eine graphische Darstellung durch einen Graphen. Letztere ist vielleicht für einen „geübten“ Anwender eher geeignet; diese Diskrepanz zwischen dem Entwickler einer Benutzerschnittstelle (der geübte Anwender) und dem Benutzer der Schnittstelle (dem naiven Benutzer) ist, wie sich auch in der Datenbanksprache SQL zeigt, weit verbreitet.

Benutzerschnittstellen für das WWW (*Web interfaces, Web data space*)
: Das WWW ist unbestritten eine der bedeutendsten Technologien der Informationsverarbeitung in der heutigen Zeit. Auch aktuelle Artikel zu Benutzerschnittstellen für das WWW sehen anhaltenden Bedarf an Fortschritten in Technologien wie Browsing und Navigation (*„Major advances are necessary in browsers, navigation, and information management* [...]“, [Nie99, S. 72]). Das WWW ermöglicht, geeignete Benutzerschnittstellen vorausgesetzt, z. B. auch kooperatives Arbeiten mit Multimedia-Anwendungen [NGP+99].

  WWW-Browser als Benutzerschnittstellen für Datenbanken haben stets mit der Zustandslosigkeit des zugrunde liegenden Protokolls HTTP (*hypertext transfer protocol*) zu kämpfen, das transaktionsorientierte Anwendungen nur durch zusätzliche Mechanismen ermöglicht [Loe97, Tur99].

  Bei neueren Entwicklungen spielen Visualisierungen ebenfalls eine Rolle, z. B. im Bereich der WWW-Browser: „Hyperbolische Browser, die eine drehbare, gewölbte Bildschirmdarstellung erlauben, sollen mit den Eigenschaften hierarchischer Browser [...] kombiniert werden. Die hyperbolische Browser-Technologie [...] ermöglicht das Ein- und Auszoomen auf einen Seiteninhalt. Für die Speicherung und Organisation von Dokumenten wird Microsoft ihre Technologie „Data Mountain“ weiter ausbauen [...]. Dabei werden Seiten durch perspektivische Darstellung zu einem keilförmigen Gebirge angeordnet.“ [Bon99]

Selbst wenn in dieser Arbeit das WWW nicht explizit als Grundlage oder Rahmenbedingung des entwickelten Visualisierungsverfahrens betrachtet wird, sollte es nicht unbeachtet bleiben. In Abschnitt 6.3 wird diesem Aspekt rückblickend die Aufmerksamkeit gewidmet.

Kooperative Benutzerschnittstellen (*computer supported cooperative work*, CSCW)
: Viele Anwendungen für kooperatives Arbeiten erzeugen in wachsendem Maß (Un-) Mengen von Daten, z. B. Systeme für virtuelle Treffen, die alle ausgetauschten Informationen protokollieren. Diese Daten werden nutzlos, wenn sie nicht mit geeigneten Mechanismen gesammelt, analysiert, verarbeitet und verstanden werden können. Chen et al. beschreiben in [CNJ+98] ein System zur Visualisierung von Diskussionsinhalten, das Eigenschaften von Themen wie z. B. Häufigkeit der Nennung oder Länge der Diskussion mit Attributen wie z. B. Färbung oder Größe assoziiert, und verschiedene Darstellungsarten wie z. B. Daten-Landkarten oder Hypertexte enthält. Ziele sind dabei, Schlüsselworte zu synthetisieren, das Browsen in Dokumentsammlungen zu ermöglichen, und ähnliche Dokumente visuell zu Bündeln (*visual clustering*)

Virtuelle Realität (*virtual reality*)
: Virtuelle Realität versucht Metaphern physisch existierender Objekte wie z. B. Räume, Gebäude oder Bücherregale möglichst real als Benutzerschnittstelle abzubilden. Im Datenbankbereich wird sie meist nur für sehr spezielle Ausprägungen als Benutzerschnittstelle eingesetzt (siehe z. B. [AJ94, YH97, POS+98]). Hier soll explizit ein generisches Visualisierungsverfahren entwickelt werden, das deshalb von den Methoden und Konzepten der virtuellen Realität zwar beeinflußt sein kann, aber diesem Bereich nicht zuzuordnen ist.

Werkzeuge für Benutzerschnittstellen (*user interface tools*)
: Ein Überblick der historischen Entwicklung der Programmierung graphischer Benutzerschnittstellen, bei der auch die dabei verwendeten Werkzeuge angesprochen werden, wird in Abschnitt 2.2 gegeben. Dort wird auch Bezug auf das in dieser Arbeit vorgestellte Visualisierungsverfahren genommen. Die Ausgabe der Zeitschrift IEEE Software mit dem Hauptthema „Erstellen effektiver Benutzerschnittstellen“ (Band 14, Nummer 4, Juli/August 1997, S. 21–72: „Creating Effective Interfaces“) enthält in der Einführung von Sears und Lund eine ausführliche Übersicht weiterer Quellen zum Thema [SL97, S. 23].

Zusammenfassend läßt sich sagen, daß tabellenartige Darstellungen für die meisten Anwendungen, die hierarchisch strukturierte, textuelle Daten enthalten und diese mit ihren Beziehungen darstellen wollen, sehr gut geeignet sind. Man denke z. B. an Fahr- und Flugpläne, Vorlesungsverzeichnisse oder Raumbelegungspläne. Dabei ist eine Schachtelung hilfreich, die großvolumige Datenmengen in verschiedenen Granularitäten partitioniert. Für Fahrpläne kann die Schachtelung z. B. nach Zeitintervallen, für Vorlesungsverzeichnisse nach

Fachbereichen, usw. erfolgen. Die Schachtelung der äußeren Ebene kann auch durch physische Partitionierung erfolgen, d. h. durch Aufteilung in mehrere Tabellen; z. B. kann das Vorlesungsverzeichnis einer Universität nach Fachbereichen getrennt angeboten werden.

Die tabellarische Darstellung ist auch deswegen gut geeignet, weil sie wenige sog. „arbiträre Vereinbarungen“ benötigt („Die intuitive, unmittelbare Verständlichkeit von Visualisierungen begründet sich in der Tatsache, daß die Information mit Hilfe weniger, im Gegensatz zur Sprache weder arbiträrer noch diskreter Vereinbarungen kommuniziert wird.“ [RS99, S. 92]). Gleiches läßt sich auch über die Schachtelung von Tabellen für die Darstellung hierarchisch strukturierter Daten sagen.

Visualisierungen der oben genannten Art bauen auf Metaphern wie „Anschlagtafel“, „Schriftrolle“, „Buch“ und „Faltblatt“ auf und orientieren sich am herkömmlichen, gebräuchlichen Umgang mit Information. Visualisierungen, die auf anderen Metaphern aufbauen, z. B. Gebäuden, in denen man sich virtuell zu Räumen hinbewegt, um dort nach vermuteter Information zu suchen, sind z. T. auch sehr intuitiv, aber wesentlich aufwendiger in der Realisierung. Für geübte Benutzer, die hohe Anforderungen an die Interaktivität einer Anwendung stellen, sind die Reaktionszeiten oft zu langsam.

Spezielle Metaphern, wie z. B. ein „Bücherregal“, sind oft nur für spezielle Anwendungen geeignet, das Bücherregal z. B. für die Dokumenten-Recherche; allerdings sind sie für diese dann meist auch besonders gut geeignet. Datenvisualisierungen, die Datenobjekte in einer virtuellen Landschaft plazieren, z. B. indem sie Suchbegriff-Cluster als Punkte einer Landkarte darstellen, sind gut geeignet, die Existenz der Datenobjekte, z. B. eines relevanten Dokuments, zu zeigen. Sie erlauben aber nicht, die zugehörigen textuellen Daten des Objektes mitanzuzeigen; um diese zu sehen, muß dann stets ein Metaphernwechsel erfolgen.

Diagramme wie z. B. das Streckennetz einer Fluglinie, das auf eine Weltkarte projiziert wird [TTW96], können ebenfalls nur die Existenz von Datenobjekten oder Beziehungen zwischen diese zeigen, wie das Bestehen einer direkten oder indirekten Flugverbindung. Mehr als minimale Zusatzinformationen, wie die konkreten Angaben zu einem Flug, müssen wieder aus Tabellen entnommen werden.

Häufig wird man sich eine Kombination der Techniken wünschen, z. B. eine Diagrammdarstellung, aus der direkt tabellarisch dargestellte Zusatzinformationen abgerufen werden können, oder ein tabellarische Darstellung, aus der eine diagrammartige Übersicht aufgerufen werden kann. Tabellarische Darstellungen, in der Details ausgeblendet sind, erlauben eine intuitive Erfassung auch großer Datenräume. Z. B. könnte ein Vorlesungsverzeichnis nach Fachbereichen, Veranstaltungen für Vor- und Hauptdiplom sowie Fächern untergliedert sein. Für einen Fachbereich ließe sich so die Gewichtung von Vor- und Hauptdiplomsveranstaltungen, das Angebot insgesamt und die relative Stärke der einzelnen Fächer leicht überschauen. Die nähere Inspektion einer Veranstaltung, und z. B. das elektronische Einschreiben in einen Kurs, kann in einer Formulardarstellung erfolgen.

Eine Datenbankvisualisierung sollte eine flexible Schnittstelle anbieten, in die auch multimediale Inhalte integriert werden können, wie z. B. das Bild eines Dozenten in der Vorle-

sungsankündigung, ein Verweis in einen Gebäudeplan für den Veranstaltungsort, usw. Für die Interaktion über mobile Geräte, z. B. neuere Entwicklungen im Bereich der Mobiltelefone, werden Informationen auch als Audiodaten zur Verfügung stehen müssen.

Auf viele dieser Punkte kann in dieser Arbeit nur andeutungsweise eingegangen werden. Durch ein möglichst flexibles Visualisierungsverfahren, das unter den Gesichtspunkten der Konfigurierbarkeit und Erweiterbarkeit entwickelt wird, soll die Grundlage für ihre Untersuchung in weiteren Forschungsarbeiten gelegt werden. Nicht behandelt werden hier Themen wie inhaltsbasierte Anfragen und Inhaltsrückgewinnung. Zwar können multimediale Daten (Bilder, Video, usw.) durch das vorgestellte Visualisierungsverfahren dargestellt werden, die sehr spezialisierten Verfahren für die inhaltliche Bewertung dieser Daten, die prinzipiell auch integrierbar wären, müssen aber ausgespart bleiben.

Betrachtet man abschließend den sog. „Asilomar-Bericht“ von 1998 („The Asilomar Report on Database Research“ [BBC+98][4]) als jüngsten in der Reihe der Datenbankstatusberichte, so stellt man wiederum einen Umschwung fest. In Anlehnung an die beiden oben genannten Berichte versucht er erneut, den Stand der Forschung und die neuen Herausforderungen für die Datenbankwelt zu beschreiben. Zwar enthält der Bericht im Gegensatz zu seinen Vorgängern weder eine Umfrage noch eine Liste von Forschungszielen, herausragendes Thema ist aber jetzt eindeutig das WWW, das die eher konservativen Datenbankforscher durch sein rasantes Wachstum und die immensen Möglichkeiten offenbar überrascht hat: „The Web Changes Everything“ [BBC+98, S. 75].

Benutzerschnittstellen sind kein explizites Thema mehr. Es ist sogar so, daß einige der zukünftig angeschlossenen Gerätschaften („gizmos“), etwa intelligente Lichtschalter in einem Gebäude, das über das WWW gemanagt wird, überhaupt keine sichtbare Benutzerschnittstelle mehr haben. Datenbanken werden vermehrt zum Speichern und Verarbeiten von Daten *und* Programmlogik verwendet, erfüllen aber die Anforderungen dazu noch nicht gut genug („The requirements of repositories, such as [...] browsing are not well-served in most current systems“ [BBC+98, S. 76]).

Indirekt ist das Problem des Umgangs mit neuen Formen von Daten in neuen (WWW-basierten) Umgebungen überall enthalten. Prozedurale Anfragen, wie sie vor 25 Jahren populär waren, werden durch den Entwurf der Anfragesprache in XML [BP98, DFF+98] wieder in Mode kommen. Eine weitere Folge der Entwicklung von XML als Erweiterung von HTML zur besseren Beschreibung von strukturierten Daten ist die Zunahme komplexer Datenobjekte. Deren Form wird eher hierarchisch als relational oder objekt-orientiert strukturiert sein, und sie werden insbesondere tief geschachtelt sein („The database research community should undertake the substantial effort of unifying web and database technologies, [...] Representative issues include handling sets of [...] potentially deeply nested objects [...]“ [BBC+98, S. 79]). Die Visualisierung solcher hierarchisch strukturierten und tief geschachtelten, komplexen Datenobjekte ist das zentrale Thema dieser Arbeit.

[4] Autoren sind die bekannten Datenbankforscher Bernstein, Brodie, Ceri, DeWitt, Franklin, Garcia-Molina, Gray, Held, Hellerstein, Jagadish, Lesk, Maier, Naughton, Pirahesh, Stonebraker und Ullman

1.2 Objekt-Orientierung und Datenbanken

Wie oben angedeutet, ist der Einfluß der „Objekt-Orientierung“ in der Datenbankwelt der zweite große Grund, warum den Benutzerschnittstellen noch mehr Bedeutung zukommen sollte. In ihrem Beitrag „Of Objects and Databases: A Decade of Turmoil“ [CD96] gaben Carey und DeWitt 1996 eine Übersicht der Entwicklung im letzten Jahrzehnt und einen Ausblick auf das kommende Jahrzehnt, d. h. der Zeit der post-relationalen Datenbanktechnologie. Die folgende Betrachtung ist stark an diesen Artikel angelehnt[5].

Die Mitte der achtziger Jahre war eine unruhige Zeit für die Datenbankforschung. Die Unruhe stand im Gegensatz zur kommerziellen Entwicklung. Rund 15 Jahre nach Codds grundlegender Arbeit zum Relationenmodell für Datenbanken [Cod70], und rund zehn Jahre nach der Beschreibung der fundamentalen Implementierung des „System R“ [ABC+76], wurden relationale Datenbanksysteme in der Form von IBM SQL/DS, Oracle, Ingres und Adabas kommerziell in größerem Umfang eingeführt. Die Normalisierungstheorie war gut entwickelt, mit dem Entity-Relationship-Modell [Che76] existierte eine ausreichende höhere Modellierungsschicht, SQL wurde erstmals standardisiert [Int87] (siehe Abschnitt 2.1.4).

Die Unruhe entstand durch die Beobachtung verschiedener Mängel. Erstens waren relationale Datenbanken bei gewissen Formen der Anfragen, die über sequentielles Durchsuchen abgearbeitet wurden, recht langsam. Dies hing auch mit der starken Verteilung der Daten auf mehrere Tabellen zusammen, die eine inhärente Folge der Normalisierung ist. Aus Sicht des Anwenders wurde sie als unnatürlich empfunden. Besonders bei der Behandlung technisch-wissenschaftlicher Aufgaben, bei denen z. B. Konstruktionsdaten auszuwerten waren, entstanden unbefriedigende Laufzeiten und sehr komplexe Anfragen. Diese resultierten u. a. aus der Notwendigkeit, über Joins logisch zusammenhängende Daten aus physisch getrennten Relationen zusammensetzen zu müssen. Ein bekanntes Beispiel ist die Verarbeitung geometrischer Daten, bei der Punkte, Linien, Polygone, usw. in getrennten Tabellen gehalten werden [KW87]. Lösungsansätze beruhen auf eher speziellen Konzepten, z. B. Cluster-Indizes als Zugriffsverfahren für logisch zusammenhängende, aber physisch getrennt gespeicherte geometrische Daten [Zir92].

Zweitens paßte die mengenorientierte Verarbeitungsform, wie sie natürlicherweise durch die Relationen-Algebra in der Ausprägung von SQL für den Umgang mit relationalen Datenbanken angeboten wird, schlecht zur satzweisen (*one-tuple-at-a-time*) Verarbeitungsform der Programmiersprachen. Diese bis heute gültige Diskrepanz wird mit dem Schlagwort der Impedanzunverträglichkeit (*impedance mismatch* [Ban88]) bezeichnet. Der für die Verbindung von SQL und Programmiersprachen eingeführte Daten-Cursor ist dabei eher ein Fremdkörper innerhalb der Relationenalgebra.

[5] Auch Carey und DeWitt sehen die Notwendigkeit der Navigation an der Oberfläche, auch wenn sie den Begriff der „Oberfläche“ nicht unbedingt als Benutzerschnittstelle im Sinne der HCI verstehen: „At the surface [...] good mappings and programming interfaces will be needed [...] with fully integrated object query support in addition to navigation.“ [CD96, S. 11]

Noch sehr viel radikaler waren Überlegungen im Zusammenhang mit den neuentstandenen[6] objekt-orientierten Programmiersprachen. Diese führten Begriffe wie Datenabstraktion, Datenkapselung und Vererbung ein. Durch eine zusätzliche Eigenschaft, die Persistenz, sollte der Impedance Mismatch ausgeräumt werden. Die transienten, im flüchtigen Hauptspeicher abgelegten Programmiersprachen-Objekte sollten durch entsprechende Zusatzvereinbarungen persistent gemacht werden können, also auf nicht-flüchtige Datenträger geschrieben werden. Von dort sollten sie transparent, d. h. ohne zusätzliche Anweisungen, wieder in Anwendungsprogramme einfließen können [AB87]. Frühe Entwicklungen sind z. B. Gemstone auf der Basis von Smalltalk [CM84], Vbase auf der Basis einer CLU-artigen Sprache [AH90] und Orion auf der Basis von CLOS [BCG+87].

Zusätzlich zu den Merkmalen der objekt-orientierten Wirtssprache wurden wiederum Konzepte wie Navigation, Versions-Management, andere Anfragemöglichkeiten und Indexbildung diskutiert. Dies führte dazu, daß man die entstehenden Systeme als objekt-orientierte Datenbanksysteme (OODBS bzw. OODBMS) bezeichnete.

Generell entstand der Eindruck, Datenbanken müßten „aktiver“ werden. Neben der Eigenschaft, Daten konsistent und redundanzfrei abzuspeichern, sollten sie aktive Elemente wie Trigger, Regeln, und Programmstücke enthalten. Ein Beispiel dafür ist der *„procedure as a data type“* genannte Ansatz [Sto86].

Zuletzt sei noch die Forderung nach Erweiterbarkeit erwähnt, die davon ausgeht, daß kein Datenbanksystem die Anforderungen aller Anwender gleich gut erfüllen kann. Die Idee ist dabei, einen Satz von Komponenten zur Verfügung zu stellen, z. B. einen Speichermanager, einen Anfrageoptimierer, einen Manager für einen erweiterten Transaktionsbegriff (lange und/oder geschachtelte Transaktionen), usw. Dem Anwender wird dadurch eine Plattform für Eigenentwicklungen gegeben, die gemäß den speziellen Anforderungen maßgeschneidert konstruiert werden können. In einigen Projekten wurden erweiterbare Datenbanken untersucht und Prototypen entwickelt, z. B. EXODUS [CDF+86], GENESIS [Bat86], DASDBS [DPS86], AIM-P [Dad88] und Starburst [SCF+86].

Insgesamt kann die Entwicklung der achtziger Jahre in vier Entwicklungslinien eingeteilt werden:

- Erweiterte relationale Datenbanksysteme,
- persistente Programmiersprachen,
- OODBS und
- Datenbankbausätze und -komponenten.

Davon hat sich nur eine Linie als kommerziell erfolgreich erwiesen. Datenbankbausätze und die persistenten Programmiersprachen hatten keinen dauerhaften und weitreichenden Durchbruch, obwohl sie zu vielen interessanten und auch für die anderen Linien bedeutenden Forschungsergebnissen führten. Davon ist z. B. die navigierende Programmierschnitt-

[6] Die Wurzeln der Objekt-Orientierung liegen mit Simula schon Anfang der sechziger Jahre, der Name „C++“ stammt bereits aus dem Jahr 1983 [PKP98, S. 25].

stelle von OODBS zu nennen. Auch letztere, die OODBS, führen heute nur ein Nischendasein. Zwar unterstützen inzwischen Sprachhüllen um relationale Systeme herum in Client-Server-Umgebungen auf der Client-Seite objekt-orientierte Anwendungen. Dies ist aber eher ein Ansatz der neunziger Jahre in Zusammenhang mit CORBA, OLE, Java und dem Begriff der „Middleware“ als Schicht zwischen Datenbank und Anwendung. Er ist nicht Teil der oben genannten vier Entwicklungslinien.

Als bedeutendsten Überlebenden der achtziger Jahre können die erweiterten relationalen Systeme, die heute unter dem Begriff „objekt-relationale Datenbanken“ [Sto96, SB98] firmieren, betrachtet werden.

Die Gründe für das (vermeintliche oder tatsächliche) Scheitern der anderen Linien sind vielfältig. Die hohen Ansprüche der OODBS wurden z. B. 1989 von einigen der führenden Forscher auf diesem Gebiet, nämlich Atkinson, Bancilhon, DeWitt, Dittrich, Maier and Zdonik, im „Object-Oriented Database System Manifesto“ [ABJ+90] festgehalten. Als notwendige Elemente werden dort aufgeführt:

- Unterstützung für komplexe Objekte,
- Objektidentität,
- Kapselung (*encapsulation*),
- Klassen- und Typenkonzept mit Vererbung und Substituierbarkeit,
- Überschreiben, Überladen und spätes Binden (*overriding, overloading, late binding*) von Operationen (Methoden),
- vollständige Berechenbarkeit der Methoden[7],
- Erweiterbarkeit,
- Persistenz,
- Sekundärspeicher-Management,
- Kontrolle der Nebenläufigkeit (*concurrency*),
- Wiederherstellbarkeit bei Absturz (*recovery*) und
- Ad-hoc-Anfragen.

Als optionale Elemente werden aufgeführt:

- Multiple versus singuläre Vererbung,
- statische versus dynamische Typprüfung,
- Verteilung,
- lange und geschachtelte Transaktionen und
- Versions-Management.

Offen gelassen wurden u. a. das Programmiersprachen-Paradigma und die Details des Typsystems bzw. Objektmodells. Nicht erwähnt wird z. B. ein Sicherheitskonzept.

Neben den drei anfänglichen Forschungssystemen hatten auch Systeme wie O_2 [BBB+88, BDK92], ObjectStore [LLOW91], und ODE [AG89] entscheidenden Einfluß auf die Entwicklung. Als Produkte existieren heute u. a. GemStone [BOS91], Objectivity [Obj98], ObjectStore, Jasmine [Com99], O_2, Poet [POE99] und Versant [Ver99]. Die Object Database

[7] Im Sinne der Sprachmächtigkeit, d. h. Typ-0 Berechenbarkeit.

Management Group (ODMG) definiert seit den frühen Neunzigern Standards für Objektdatenbanksysteme, darunter die Object Data Language (ODL), die Object Query Language (OQL), sowie Schnittstellen zu Smalltalk, C++ und Java (siehe Abschnitt 2.1.2).

Trotz dieser zahlreichen Produkte und eines De-facto-Standards ist die Entwicklung unbefriedigend. Als Mängel gelten:

- Uneinheitliche Einhaltung der ODMG Standardisierungsvorschläge,
- eingeschränktes oder nicht vorhandenes Sichtenkonzept,
- schwierige Schemaentwicklung (*schema evolution*),
- unzureichende Zuverlässigkeit,
- zu enge Programmiersprachen-Bindung und
- eine fehlende einheitliche theoretische Fundierung.

Speziell erfordern alle Änderungen eine Wiederholung der Kodieren-Übersetzen-Linken-Schleife. Als Alternative sind ODBC-Anbindungen an relationale Datenbanken entstanden, die ähnlich wie eingebettetes SQL eingesetzt werden können.

Zu den heutigen objekt-relationalen Datenbanksystemen gehören die Produkte von CA-Ingres (Ingres II), IBM (DB2/CS V2), Informix[8] (Dynamic Server), UniSQL (UniSQL Server) und Oracle (Oracle8). Ihre Entwicklungsrichtung wurde 1990 von Stonebraker, Rowe, Lindsay, Gray, Carey, Brodie, Bernstein und Beechim im „Third-Generation Database System Manifesto“ [SRL+90] skizziert. Die Autoren verstehen es als Antwort auf [ABJ+90] und fordern darin:

- Ein reichhaltiges Typsystem für strukturierte Objekte,
- Vererbung,
- Funktionen und Kapselung,
- optionale eindeutige Objektidentifikatoren,
- Regeln und Trigger,
- eine höhere Anfrageschnittstelle,
- gespeicherte und virtuelle Kollektionen,
- änderbare Sichten (*updatable views*),
- Trennung von Datenmodell und Leistungsmerkmalen (Indexe, *clustering*, usw.),
- Zugang über mehrere Sprachen,
- geschichtete, persistenzorientierte Sprachanbindungen,
- Unterstützung von SQL und
- eine Client-Server-Schnittstelle, die das Verschicken von Anfragen unterstützt (*query shipping*).

Objekt-relationale Systeme setzen auf dem relationalen Modell und seiner Realisierung durch SQL auf. Frühe Systeme unterschieden zwei Arten von Objekttypen. Erstens die benutzerdefinierten Basistypen im Sinne abstrakter Datentypen (ADTs). Dazu gehören

[8] Informix hat Illustra, einen ehem. Marktführer in diesem Bereich, aufgekauft.

insbesondere nicht-alpha-numerische Typen wie Audio, Video, Zeitreihen, „langer Text“ (z. B. ein Lebenslauf als Attributwert in einem Angestellten-Tupel), Pixelbild (z. B. das Paßfoto des Angestellten) und geometrische Datentypen wie Punkt, Linie und Polygon.

Zweiten gibt es sog. „Tupeltypen“ (*row types*), eine Erweiterung des rein relationalen Tupelkonzeptes auf benannte Typen und Funktionen bzw. Methoden. Dazu gehören auch Referenzen, die eigentlich als „Pointer Spaghetti“ verpönten Zeiger, und komplexwertige Attribute (Mengen von Tupeln, Multimengen, Felder und Listen). Auf der obersten Ebene wird als Datenbankschema weiterhin eine Sammlung von benannten Tabellen gebildet. Dieses enthält aber jetzt, in Analogie zu den OODBS, zusätzlich ein reiches Angebot an Objekttypen.

Zur Behandlung mittels SQL werden Pfadausdrücke, methodenartige Funktionsaufrufe und geschachtelte Klauseln im FROM-Teil zur Behandlung mengenwertiger Attribute eingeführt. SQL3 hat hier mittlerweile eine Standardisierung gebracht [Mel96] (siehe Abschnitt 2.1.4). Die Anpassung deklarativer Anfragesprachen wie SQL an die Handhabung komplex strukturierter Objekte macht diese Sprachen aber noch komplizierter und damit noch ungeeigneter für viele Benutzer [Ioa92].

Ein weiterer Ansatz, objekt-orientierte Konzepte für die Erstellung von Datenbankanwendungen verfügbar zu machen, sind die bereits erwähnten client-seitigen Sprachhüllen (*client wrappers*). Diese werden beim Client eingerichtet und bieten dort objekt-orientierte Funktionalität an, nutzen aber den traditionellen Datenzugriff auf bestehende relationale Server. Leider läßt die Funktionalität dieser Wrapper zu wünschen übrig. Speziell im Anfragebereich zwingen sie den Anwender, Ad-hoc-Anfragen in SQL zu formulieren, was wieder zum bekannten Impedance Mismatch führt. Unklar ist, ob die Programmlogik besser in den Regeln, Triggern und Prozeduren der Datenbank aufgehoben ist oder in die objekt-orientierte Benutzerschnittstelle hineinprogrammiert werden soll.

CORBA (*common object request broker architecture*) [MZ95, Pop98] ist ein Standard [OMG98] der Object Management Group (OMG)[9] für die Interaktion zwischen verschiedenen Objektwelten. Dessen Rolle für OODBS und objekt-relationale Datenbanken wird als nicht zukunftsträchtig eingeschätzt („[in factorable object services] we predict that OMG will fail [...]“, [CD96, S. 10]. Zwar ist der angebotene Mechanismus zum Aufruf entfernter Methoden (*remote procedure call*, RPC) für die Inanspruchnahme von Diensten auf fremden Rechnern nützlich, und die Registrierung von Diensten in verteilten Umgebungen von Interesse. Aber es ist zu erwarten, daß sich Dienste für Persistenz, Indexbildung oder Transaktionen nicht bewähren werden, da es in all den Jahren der Datenbankforschung noch nicht gelungen ist, diese isoliert anzubieten. Ein weiterer De-Facto-Standard im Bereich Objektverteilung ist das von der Firma Microsoft etablierte OLE (*object linking and embedding*) zusammen mit dessen Objektmodell COM bzw. DCOM (verteiltes COM, Microsofts Antwort auf CORBA). Wegen der enormen Verbreitung der Produkte von Microsoft arbeiten viele Firmen an Schnittstellen für ihre Datenbank-Server, die Objekte gemäß OLE bzw.

[9] Nicht zu verwechseln mit der Object Database Management Group (ODMG).

COM, im Sinne abstrakter Datentypen, beim Endanwender unterstützen.

Mit der im objekt-orientierten Umfeld wichtigen Programmiersprache Java lassen sich sowohl auf der Server- als auch auf der Client-Seite die gerade genannten komplexen Objekte (ADTs) ideal realisieren. Für Java existieren bereits verschiedene Kopplungen mit Datenbanken, z.B. JDBC und SQLJ [Mal98] (siehe Abschnitt 2.1.2). Außerdem läßt Java sich gut ins WWW integrieren, so daß diese Entwicklung sicher zukunftsträchtig ist. Alternativ können Lösungen aber auch mit einer anderen Skriptsprache realisiert werden, mit der sich komplexe Datenbankobjekte behandeln lassen. Wenn sie ein Plug-In für WWW-Browser anbietet, das übers Netz empfangene Programmstücke in einer sicheren Interpreterumgebung ausführen kann, ist auch eine WWW-Integration gegeben. In dieser Arbeit wird Tcl [Ous94] als eine solche alternative Sprache vorgestellt und verwendet.

Die Zukunft wird den objekt-relationalen Datenbanksystemen gehören. Diese werden ihre objekt-orientierten Fähigkeiten ausweiten. Zu den zu erwartenden Verbesserungen gehören

- Bibliotheken von ADTs mit Vererbungskonzept,
- transparente Verteilung von Daten in Zwischenspeicher (*caches*) auf dem Client-Rechner und verteilte Ausführung von Methoden,
- Anpassung von SQL an objekt-orientierte Konzepte und speziell komplexe Objekte,
- Anbindung an die Programmiersprachen C++, Smalltalk und Java und
- zusätzlichen Schnittstellen für deskriptive Objektanfragen und die Navigation auf den Daten.

Viele dieser Aspekte werden von den Standards SQL3 und ODMG-2.0 abgedeckt (siehe Abschnitte 2.1.4 und 2.1.2). Der alles übergreifende Schlüsselbegriff in dieser Liste ist sicherlich Erweiterbarkeit, da die Liste der möglichen Anwendungen und Interaktionsformen so schnell wächst (nichts ist in der Informatik so beständig wie der Wandel), daß man nicht mit einem Datenmodell für jetzt und alle Zeiten rechnen kann.

1.3 Der Datenbank-Editor ESCHER

Es ist interessant, die Geschichte des Kasseler ESCHER-Projektes[10] im Licht der Entwicklung von objekt-relationalen Datenbanken zu betrachten.

Die Ursprünge von ESCHER führen auf das Jahr 1986 zurück. Am Wissenschaftlichen Zentrum der IBM in Heidelberg wurde in den achtziger Jahren von Dadam, Küspert und anderen ein Prototyp zum fortschrittlichen Informations-Management (*Advanced Information Management Prototype*, AIM-P) entwickelt [DKA+86]. Dieser sollte im Wesentlichen das eNF^2-Datenmodell realisieren, also aus heutiger Sicht strukturell komplexe Objekte.

[10] Der Namenspatron von ESCHER ist der holländischen Künstler M. C. Escher (1898–1972), der für seine selbstreferenzierenden Graphiken bekannt ist.

Ähnliche Entwicklungen liefen zu dieser Zeit in Darmstadt mit Scheks[11] DASDBS [SPS87, SPSW90], in Clausthal-Zellerfeld mit Heuers OSCAR [Heu89] und in Kaiserslautern mit dem PRIMA System von Härder und Meyer-Wegener [HMW+87].

Im Ausland waren zu dieser Zeit Bubba [BAC+90], das Gamma Project [DGS+90], Iris [WLH90], O_2 [D+90], ORION [KGBW90], POSTGRES [SRH90] und Starburst [HCL+90] in der Entwicklung. Die Spezialausgabe zu prototypischen Datenbanksystemen der IEEE Transactions on Knowledge and Data Engineering (Band 2, Nummer 1, März 1990: „Special Issue on Database Prototype Systems“) mit der Einleitung von Stonebraker [Sto90] gibt zu den oben genannten Systemen einen Überblick.

Im Rahmen des AIM-P Projektes wurden u.a. effiziente Verfahren zur Sortierung und Duplikatseliminierung entwickelt, woran auch Wegner aus Kassel maßgeblich beteiligt war [SLPW89, KSW89, WT89, TW91]. Dafür wurde eine Plattform geschaffen, mit der sich Dateneingabe, Sortierverfahren und Resultatausgabe verwirklichen ließen.

Auf Mehrbenutzerfähigkeit kam es dabei nicht an, AIM-P selbst lief zunächst nur auf einem Großrechner unter VM und war in Kassel nicht verfügbar. Deshalb wurde dort ein entsprechendes System entwickelt, das unter DOS auf einem PC geschachtelte relationale Tabellen speichern und darstellen konnte. Die Benutzerschnittstelle wurde mit dem textbasierten Fenstersystem Turbo Vision von Borland entwickelt, was damals noch Stand der Technik war. Sie kann damit als textuelle Ganzseiten-Bildschirmschnittstelle bezeichnet werden.

Turbo Vision mit überlappenden Fenstern, Pull-Down-Menüs und Mausunterstützung ist eine Mischung aus reaktivem Programmierparadigma (Ereignisschleife zum Abfragen von Bildschirm- und Tastaturereignissen) und prozeduralem Programmieren. Es kommt noch heute in vielen kleineren kommerziellen Anwendungen zum Einsatz. In ESCHER hatte die Form der pseudo-graphischen, textorientierten Ausgabe durch mosaikartiges Kombinieren der Graphikelemente des erweiterten IBM Zeichensatzes (die sog. „Blockgraphik“) zur Folge, daß alle Größenberechnungen in einem Spalten-/Zeilenraster vorgenommen wurden. Dies wurde auch in späteren Visualisierungskonzepten übernommen, die eigentlich auf Berechnungen in der Bildschirmeinheit Pixel aufbauen, z. B. dem Visualisierungskonzept des gegenwärtigen, auf OSF Motif basierenden, Prototypen von ESCHER. Aus dieser Zeit stammt auch die feste Kopplung von Visualisierungsdaten mit den eigentlichen Objektdaten.

Sehr schnell stellte sich heraus, daß insbesondere die folgenden Aspekte von ESCHER interessante Forschungsthemen per se darstellten:

- Die Interaktion mit geschachtelten Daten,
- die Darstellung eines begrenzten Ausschnitts einer im Verhältnis zum kleinen Bildschirm sehr großen geschachtelten Tabelle und
- die adäquaten Speicherung der Daten.

[11] Ursprünglich war auch Schek an der Heidelberger Forschung beteiligt.

Die Konzepte der Interaktion mit geschachtelten Datenobjekten durch Cursor, in ESCHER „Finger“ genannt (siehe Abschnitt 2.3.2), und eines Meta-Schemas als Schema aller Schemata inklusive seiner selbst (siehe Abschnitt 2.3.3), wurden von Wegner erstmals 1989 in Tokyo publiziert („ESCHER — Interactive, Visual Handling of Complex Objects in the Extended NF^2-Database Model“ [Weg89]). Die dortige Konferenz wurde zur ersten aus der bereits in Abschnitt 1.1 erwähnten Reihe „Visual Database Systems“. Sie war insofern wegweisend, als sie erstmals die visuellen Aspekte von Datenbanken, sowohl was die Formen des Umgangs mit den Daten als auch die Art der Daten selbst anbetrifft, in den Vordergrund stellte. Viele Autoren des Konferenzberichtes [Kun89] haben die Entwicklung im Bereich Multimedia und komplexe Objekte massiv beeinflußt, unter ihnen Castelli, Hsiao, Jungert, Kambyashi, Klas, Klinger, Kunii, Lum, Meier, Meyer-Wegener, Neuholt, Paredaens, Rabitti, Stucki, Wegner und Wiederhold.

Etwa ab 1990 wurde im ESCHER-Projekt auch die Modellierung und Darstellung geometrischer Daten untersucht. Dieses Thema war bereits früher im AIM-P Projekt bearbeitet worden [DTLS91]. Insbesondere die Modellierung geometrischer Daten als Beispiel für die sog. „Nicht-Standard-Anwendungen“ war Gegenstand vieler anderer Forschungsarbeiten, siehe z. B. [KW87]. Für ESCHER wurde eine duale Benutzerschnittstelle konzipiert. Die tabellarische Darstellung komplexer Objekte wurde dabei an eine graphische Darstellung der geometrischen Objekte gekoppelt. Eingaben konnten in beiden Darstellungen vorgenommen werden. Damit war es z. B. möglich, die Koordinaten einer Ecke eines Polygons auf zwei Arten zu verändern:

- Entweder wurden die Werte direkt in der tabellarischen Darstellung eingegeben oder
- in der graphischen Darstellung wurde die Ecke bewegt, z. B. durch Ziehen mit der Maus (*drag & drop*).

Beide Varianten hatten die gleichen Auswirkungen. Die Veränderung der graphischen Darstellung bewirkte die Änderung der Datenbankwerte und umgekehrt. Ein Beispiel hierfür findet sich in [Weg91a], eine weitere Beschreibung in [WPC92].

Letztere Arbeit enthält auch Vorschläge für den Umgang mit varianten Tupeln, eine Idee, die später von Kalus und Dadam nochmals aufgegriffen [KD95] und in neueren Arbeiten des ESCHER-Projektes ebenfalls weiter verfolgt wurde [Pau94, The96].

Die Systemarchitektur von ESCHER, die bis heute in etwa beibehalten wurde, ist ebenfalls in [Weg91a] beschrieben. Sie umfaßt für die unteren drei Schichten eine Aufteilung in

- Record Manager (RM),
- Data Manager (DM) und
- Object Manager (OM).

Der Record Manager [Weg90] ist für die persistente Speicherung von Byte-Folgen zuständig und vergibt für diese persistenten „Objekte“ Identifikatoren. Dabei wurde das aus System R [ABC+76] bekannte Konzept der invarianten Tupel-Identifikatoren (*tuple identifier*, TID)

übernommen, für das im Record Manager der Name Record Identifier (RID) verwendet wird. Record Identifier können als Objektidentifikatoren im Sinne von Abschnitt 1.2 betrachtet werden, sie sind aber implementationsnäher ausgelegt. Aus heutiger Sicht würde man diese Schicht als Objektspeicher (*object repository*) bezeichnen, wobei ein Transaktionskonzept, das Mehrbenutzerfähigkeit durch Kontrolle der Nebenläufigkeit (*concurrency*), Wiederherstellbarkeit bei Absturz (*recovery*), usw. ermöglichen würde, fehlt. Insofern kann es nicht mit anderen experimentellen Objekt-Speichern wie etwa Shore [CDN+94] oder EOS [Bil92a, Bil92b] auf eine Stufe gestellt werden.

Auf der nächsthöheren Ebene ist der Data Manager [Weg91b] für die Strukturierung der Byte-Folgen verantwortlich. Er greift anhand von RIDs auf die Byte-Folgen zu und interpretiert diese. Dazu hat er ein Repertoire an elementaren Speichertypen (z. B. Integer, String) und verwaltet diese in baumartigen Strukturen, die bereits eine Implementierung des NF^2-Datenmodells darstellen.

Die Behandlung „großer“ Objekte, d. h. solcher Objekte, die nicht in einer Seite als kleinster Transfereinheit zwischen Hauptspeicher und Sekundärspeicher untergebracht werden können, war ursprünglich zwischen Record Manager (lange Byte-Folgen) und Data Manager (Kollektionen großer Kardinalität) aufgeteilt. Inzwischen hat diese Aufgabe vollständig der Data Manager übernommen [WTZ97], was auch mit der Umsetzung des aus persistenten objekt-orientierten Programmiersprachen bekannten Konzepts des *pointer swizzling* (siehe z. B. [KK93, Mos92]) zusammenhängt [WPTT96].

Über dem Data Manager liegt schließlich der Object Manager, auf dessen Fähigkeiten in dieser Arbeit ganz wesentlich aufgesetzt wird. Der Object Manager versteht die komplexen Datentypen, die dem Anwender an der DBMS-Schnittstelle angeboten werden. Er kann z. B. zwischen einer (geordneten) Liste und einer Menge unterscheiden, kann einer Komponente eines Tupels seine physische Position im Vektor, der das Tupel realisiert, zuweisen, usw. Er kennt Finger und bietet eine Fülle von Funktionen an, die der Navigation und Datenmanipulation mit diesen „Objektzeigern“ dienen.

Interessanterweise haben sich die in [Weg91a, S. 346] aufgelisteten Funktionen und Sprechweisen bis heute als ausreichend und im Wesentlichen nicht veränderungsbedürftig erwiesen. Abschnitt 2.3.4, der sich mit den Operationen auf Fingern in Tcl/DB befaßt, wird auf diesen Befehlssatz zurückkommen.

Die unteren drei Schichten von ESCHER implementieren insgesamt das eNF^2-Datenmodell, wobei in der mittleren Schicht, dem Data Manager, der Übergang zwischen relationalem Datenmodell und komplexen Objekten liegt.

Die Zeit von 1992 bis 1995 war auch für ESCHER von der oben beschriebenen Unruhe in der Datenbankwelt geprägt. Das Interesse an NF^2-Strukturen ging stark zurück. Der Umgang damit war zunächst zwar intuitiv, weil Zusammengehöriges zusammen dargestellt wurde; die Details sind aber schwierig. Normalformen für geschachtelte Relationen sind undurchsichtig [MNE96a], die Berücksichtigung der mehrwertigen funktionalen Abhängigkeiten (*multi-valued dependencies*, vgl. 4NF [Fag77] und 5NF [Vin95]) ist schwierig

[OY87, OY89], und die Entwicklung geeigneter Speicherstrukturen, die komplexe Objekte zusammengefaßt (*clustered*) abspeichern, ist aufwendig (siehe z. B. [Muß92, Tha94]).

Leichter erschien es, persistente komplexe Objekte über geeignete Programmiersprachen zu realisieren, die ein Klassenkonzept, Vererbung, selbstdefinierte Typen, usw. anbieten sowie Persistenz integrieren. D. h. die oben angesprochene Objekt-Orientierung durch den Übersetzer (*compiler*) war attraktiver.

Für ESCHER stand eine Re-Implementierung an, da sich das oben beschriebene PC-System auf Basis von DOS und Turbo Vision als wenig ausbaufähig erwies. Zu entscheiden war u. a., ob jetzt auf einer der existierenden relationalen bzw. neu angebotenen objekt-orientierten Datenbanken aufgesetzt, oder im Stil von O_2 mit einer persistenten objekt-orientierten Sprache gearbeitet werden sollte. Die Wahl fiel sehr zurückhaltend aus[12]: Portierung der Eigenentwicklung, insbesondere auch der unteren Schichten, nach C unter UNIX (AIX) mit OSF Motif als Basis für die graphische Benutzerschnittstelle. Durch die weitere Nutzung des eigenständig entwickelten Speicher-Subsystems, das persistente komplexe Objekte realisiert, wurde für ESCHER die Unabhängigkeit von konkreten Datenbanksystemen gewahrt. Konzepte der Objekt-Orientierung wurden zunächst nicht berücksichtigt.

Die derzeitige prototypische Implementierung des ESCHER-Projektes mit seiner auf OSF Motif basierenden graphischen Benutzerschnittstelle wird in dieser Arbeit X-ESCHER genannt.

Eine Folge der Umstellung war, daß die duale Interaktion für geometrische Objekte, deren Implementierung auf speziellen Mechanismen von Turbo Vision aufgesetzt hatte, zunächst entfiel. Mit der vorliegenden Arbeit wird dieses Konzept wieder aufgegriffen und verallgemeinert. Andererseits wurden multimediale Datentypen wie Pixelbilder (siehe Abschnitt 2.3.4) und eine QBE-artige Anfrageschnittstelle [Lie95, WTWL96, TTW96] integriert. Weiterhin wurde das Interaktionsparadigma auf die mausbasierte Positionierung von Fingern und Markierung von Objekten ausgedehnt [Wil96a, Wil96b]. Durch eine Umsetzung des *pointer swizzling* Konzeptes als Software-Lösung konnte nachgewiesen werden, daß sich eine persistente Datenhaltung ohne Kopieren und Transformieren von Formaten auch ohne die, z. B. in ObjectStore üblichen, sehr maschinenabhängigen Speichermanagementfunktionen realisieren läßt [WPTT96].

Bei all diesen Erweiterungen zeigte sich, daß die historisch gewachsenen Strukturen und Besonderheiten der ESCHER-Implementierung, die inzwischen einige zehntausend Zeilen Code umfaßte, und die Eigenheiten von OSF Motif zu einer sehr mühselig Weiterentwicklung des Systems führten. Zur Entwicklung der Benutzerschnittstelle waren keine höheren Konzepte (z. B. Interface Builder oder UIMS, siehe Abschnitt 2.2) anwendbar, da die Visualisierungen datengetrieben und dynamisch sind.

Sprachlösungen, die sowohl auf Datendefinitions- als auch Datenmanipulationsebene mit komplexen Objekten umgehen können und auf modernen Klassenkonzepten basieren, wur-

[12] Die Wahl könnte aus heutiger Sicht als post-relational charakterisiert werden.

den in den Dissertationen von Paul [Pau94] und Thelemann [The96] vorgeschlagen, mangels Implementierung aber nicht eingesetzt — anders als z. B. der für AIM-P implementierte Vorschlag HDBL [ALPS87], der stark in SQL3 einging. Einerseits wurde dadurch für ESCHER nie eine objekt-orientierte Sprachebene erreicht. Andererseits wäre eine von den Sprachstandards der ODMG (ODL und OQL, siehe Abschnitt 1.2) losgelöste Entwicklung langfristig unklug gewesen, wie sich durch die enttäuschten Erwartungen, die an die OODBS gestellt wurden, herausstellt hat.

Vielmehr könnte sich die Programmierung der für Objekte benötigten Methoden mittels einer kleinen, einfachen und typarmen Skriptsprache als richtig erweisen, wie es z. B. Ousterhout, der Vater von Tcl, propagiert („Scripting: Higher-level Programming for the 21st Century“, [Ous98]).

Die Implementierung einer aktiven Komponente auf der Basis kleiner, interpretierter Programmstücke existiert für ESCHER seit 1997. Unter dem Namen Tcl/DB[13] wurde Ousterhouts Skriptsprache Tcl um Kommandos für die Manipulation von und die Navigation in komplexen Objekten durch die Kopplung an die Programmierschnittstelle von ESCHERs Object Manager erweitert [WWT97, TW98, Tha98, Ahm98].

Ob die Entscheidung der Wirtssprache heute zugunsten Java ausfallen würde, das immer mehr an Bedeutung zu gewinnen scheint, ist unklar. Vielleicht erweist sich aber die eher konservative Wahl von Tcl wie im Fall der OODBS als langfristig richtig. Eine gewisse Ermüdung über die ständig neuen Merkmale und die Kämpfe um die Normung bei Java läßt das vermuten (siehe z. B. [Lew98], [Fra98] wird auf dem Titelblatt mit „Java: Half Empty?“ angekündigt). Auch Äußerungen wie die von Kernighan, einem der Väter des Betriebssystems UNIX und der Programmiersprache C, daß er momentan viel mit Tcl programmiert („Much of my programming in the past few years has been in Tcl/Tk [...]“ [Ker99, S. 67]), lassen sich als Bestärkung der Skriptidee lesen.

Sehr elegant, wenn auch schwierig in der Realisierung, ist das Konzept der Selbstreferenzierung, das ESCHER mit dem Meta-Schema von Anfang an verfolgt hat und das ihm seinen Namen gab (siehe Fußnote 10 auf Seite 17). Eine weitere Anwendung dieses Konzepts wäre das Speichern der oben genannten „kleinen interpretierten Programmstücke“ in Form von Tcl-Skripten als Datenobjekte in Tabellen. Die Idee, eine Visualisierung als Meta-Daten aufzufassen und in Tabellen des Systems abzulegen, wurde in [WWT97] beschrieben. Sie geht auf alte Konzepte der Handhabung von Meta-Daten [MR86] zurück und wird heute im Zusammenhang mit dem un- bzw. semistrukturierten Datenangebot im WWW verstärkt diskutiert. Im letzteren Fall müssen die Datenseiten als Meta-Information z. B. Interpretationshinweise transportieren („Wie muß diese Seite gelesen werden?“).

Eine Kooperation mit der SAP AG, Walldorf, hat dazu geführt, daß das Interaktionskonzept von ESCHER für die Objektbehandlung in ABAP übernommen wurde[14]. Das ist ein

[13] In anderen Arbeiten werden auch die Namen TclDB und Tcl-DB verwendet. In Anlehnung an die Schreibweise Tcl/Tk wird hier Tcl/DB verwendet.

[14] ABAP (*Advanced Business Application Programming*) ist eine Programmiersprache; das Interaktionskonzept wurde also genau genommen in den Editor der integrierten Entwicklungsumgebung übernommen.

Indiz dafür, daß Benutzerschnittstellen für komplexe Objekte, die im Hintergrund auf post-relationalen DBMS aufbauen, auch kommerziell an Bedeutung gewinnen.

Zusammengefaßt kann also festgestellt werden, daß sich die eher evolutionäre als revolutionäre Entstehungsgeschichte von ESCHER im Nachhinein als recht stabil herausgestellt hat. Sie steht damit im Einklang mit den objekt-relationalen Entwicklungen, die strukturelle Objekt-Orientierung in den Vordergrund stellen und auf eine allzu enge Kopplung an objekt-orientierte Programmiersprachen verzichten.

Die derzeitigen Erweiterungen von ESCHER, z. B. die Interaktion über eine WWW-Schnittstelle [TWW99, ATW, GWZ], zeigen, daß die grundlegenden Konzepte erweiterungsfähig sind. Eine Voraussetzung für weitere Entwicklungen ist die Lösung der sehr starren Kopplung von Visualisierung und Datenhaltung. Ein Ziel dieser Arbeit ist, diese Voraussetzung durch ein neues, erweiterbares Visualisierungsverfahren zu erfüllen.

Kapitel 2

Grundlagen

In diesem Kapitel werden Grundlagen dieser Arbeit beschrieben. Zunächst wird der Aspekt der strukturellen Komplexität in Datenmodellen erläutert (Abschnitt 2.1). Im einzelnen werden das eNF^2-, das objekt-orientierte und das objekt-relationale Datenmodell sowie SQL3 angesprochen. Dann wird ein Überblick über die Entwicklung der Programmierung graphischer Benutzerschnittstellen gegeben (Abschnitt 2.2). Schließlich wird auf einige Konzepte des graphischen Datenbank-Editors für komplexe Objekte, ESCHER, eingegangen (Abschnitt 2.3). An einigen Stellen werden in diesem Kapitel Anforderungen an das Visualisierungsverfahren formuliert, die in den folgenden Kapiteln bei dessen Entwicklung berücksichtigen werden sollen.

2.1 Strukturelle Komplexität in Datenmodellen

Ein **Datenmodell** bestimmt, wie sich Objekte und deren Verhalten, Beziehungen zwischen den Objekten, sowie Abläufe von Prozessen der (realen) zu modellierenden Welt in Datenobjekte und Algorithmen der (virtuellen) Rechnerwelt abbilden lassen. Die Abbildung soll möglichst eindeutig und elegant sein. Die Datenmodelle[1], die sich heute für Datenbanken sowohl im wissenschaftlichen als auch im kommerziellen Bereich durchgesetzt haben, sind das relationale Datenmodell und dessen Nachfolger. Das relationale Datenmodell [Cod70] hat dabei das hierarchische Datenmodell [TL76] und das Netzwerk-Datenmodell [TF76] abgelöst, obwohl diese beiden „Urahnen“ immer noch kommerzielle Bedeutung haben (Stichwort *legacy systems*: IDS, DMS, IMS). Als Nachfolger des relationalen Datenmodells, d. h. als post-relationale Datenmodelle, werden hier NF^2-Datenmodell und dessen Erweiterung, das eNF^2-Datenmodell, sowie objekt-orientierte Datenmodelle und die Mischform aus relationalem Datenmodell und objekt-orientierten Konzepten, das objekt-relationale Datenmodell

[1] Mit Datenmodellen sind hier immer Datenbankmodelle, also Datenmodelle von Datenbanken, gemeint; Datenmodelle gibt es z. B. auch für Programmiersprachen, wobei dort eher von Programmierparadigmen und Typsystemen gesprochen wird.

angesehen. Hier sollen explizit nur die Datenmodelle betrachten werden, die als direkte Vorlagen für die Realisierung existierender Datenbanksysteme angesehen werden können. Im Gegensatz dazu dienen die semantischen Datenmodelle, z.B. das Entity-Relationship- [Che76, BCN92] oder das IFO-Modell [AH84, AH87], meist nur dem konzeptuellen Entwurf eines Datenbankschemas und werden nach diesem in ein implementationsnäheres Datenmodell transformiert.

In der Literatur (z.B, [Bro84, Nav92, LK86, Heu92]) hat sich für die Definition von Datenmodellen eine Aufteilung in mehrere Komponenten herauskristallisiert:

- Im **strukturellen Teil** wird beschrieben, wie die Datenobjekte und die Beziehungen zwischen diesen modelliert werden können,
- der **operationale Teil** beschreibt die Manipulation der Datenobjekte, und
- zusätzlich werden die **Integritätsbedingungen** zur Beschreibung der Konsistenz betrachtet.

Diese Arbeit bezieht sich hauptsächlich auf den strukturellen Teil der Datenmodelle. Der operationale Teil hat hier eine geringere Bedeutung, da die navigierende Interaktion mit den Datenobjekten nicht auf den deskriptiven, sprachorientierten Ansätze der Datenmodelle beruht. Integritätsbedingungen werden in dieser Arbeit höchstens am Rande betrachtet.

Das relationale Datenmodell soll gegenüber den post-relationalen Datenmodellen als einfaches Datenmodell abgegrenzt werden. Es wird deshalb als „einfach“ bezeichnet, weil es in seiner ursprünglichen Form nur einfache Datentypen (elementare oder atomare, alphanumerische Datentypen wie Zahlen, Daten, Währungen, Zeichenketten, usw.) und fest vorgegebene Strukturierungen (Relationen sind Mengen von Tupeln) erlaubt. Im Gegensatz dazu erlauben komplexe Datenmodelle die Strukturierung von komplexen Objekten der realen Welt und deren Beziehungen in einer „natürlichen“ Art und Weise. Man muß sich jedoch bewußt sein, daß prinzipiell jedes komplexe Datenmodell, zumindest im strukturellen Teil, durch ein einfaches Datenmodell implementiert werden kann.

Die Hervorhebung der komplexen Datenobjekte beruht auf der Beobachtung, daß in fast allen Anwendungen die modellierten Entitäten strukturiert sind [Cat91]. Selbst ein so elementares Objekt wie eine Bestellung setzt sich strukturell aus Liefer- und Rechnungsadresse, die selbst wieder strukturiert sind, sowie einer Liste von Bestellposten, die ebenfalls strukturiert sind, zusammen. Bei nicht-kommerziellen Anwendungen, etwa in Wissenschaft und Technik, treten komplexe Strukturierungen sogar besonders häufig auf. Die einfachen Mittel des relationalen Datenmodells sind eher geeignet, kaufmännische und administrative Anwendungen adäquat zu modellieren und in RDBMSs effizient zu implementieren. Insbesondere eine stark hierarchisch ausgeprägte Struktur von Objekten führt, durch Normalisierung zur Vermeidung von Redundanz, im relationalen Datenmodell dazu, daß die Daten auf viele Relationen verteilt werden. Heutige kommerzielle Implementierungen von

RDBMS setzen bereits viele Ergebnisse aus der Forschung um, z. B. Hash-Joins in Verbindung mit regel- und kostenbasierter Anfrageoptimierung zur effizienteren Auswertung von Joins. Zusammen mit der gestiegenen Performance und Verfügbarkeit der Hardware werden so die Effizienzprobleme der RDBMS in Verbindung mit komplexen Datenobjekten abgeschwächt. Andere, modellinhärente Probleme, wie das Fehlen eines transitiven Abschlusses und der Rekursion, bleiben aber bestehen.

2.1.1 Das eNF^2-Datenmodell

Aus Erfahrungen mit Unzulänglichkeiten des relationalen Datenmodells hat sich die Bemühung ergeben, das relationale Datenmodell zu erweitern bzw. zu verallgemeinern. Eine Hauptrichtung war dabei die Verallgemeinerung durch Aufgabe der Forderung der sog. „ersten Normalform“ (*first normal form*, 1NF) für die grundlegende Struktur des relationalen Datenmodells, der Relation selber.

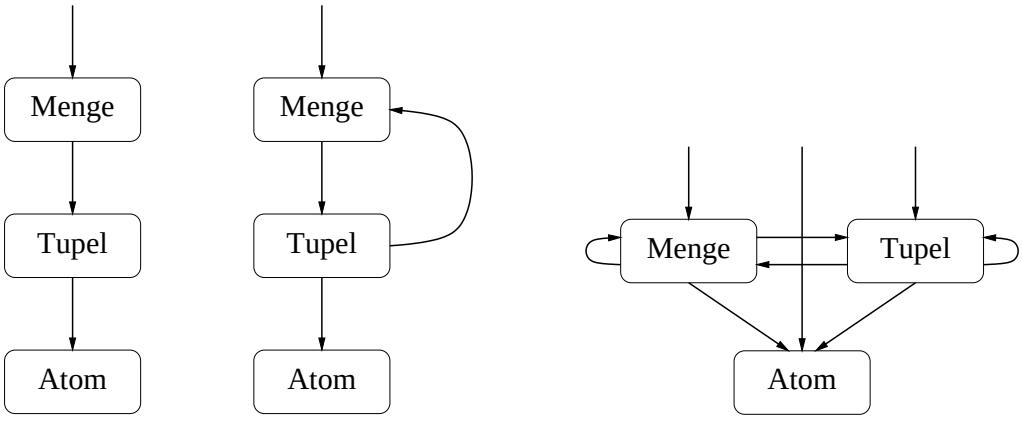
Die Forderung der 1NF für Relationen besagt, daß diese nur Mengen von Tupeln sein dürfen[2]. Die Tupel selbst bestehen aus Komponenten, deren Domänen nur einfache (atomare, skalare) Werte enthalten. Solche Relationen können immer in einer tabellarischen Art dargestellt werden, indem die Namen der Tupelkomponenten als Spaltenüberschriften angegeben werden, und die einzelnen Tupel dann untereinander aufgelistet werden. Dadurch hat sich für Relationen auch der Name Tabellen eingebürgert (siehe z. B. [KS91, S. 53 ff.]).

Durch die Aufgabe der 1NF wurde der Begriff „nicht erste Normalform“ (*non first normal form*, NFNF bzw. NF^2) für das Datenmodell geprägt. Hier wird auf eine formale Darstellung des NF^2-Datenmodells verzichtet und im Folgenden eine eher informelle Beschreibung der Konzepte gegeben. Für die formalen Aspekte sei auf [JS82, AB84, AFS89] verwiesen.

In einer NF^2-Relation wird für die Komponenten der Tupel zugelassen, daß sie selbst wieder Relationen sind. Also wird auf die Forderung nach der Einfachheit der Domänen verzichtet. Die Anordnung der strukturierenden Elemente, nämlich der Relationen, wird aber vorgegeben: Eine NF^2-Relation ist eine Menge von Tupeln, deren Komponenten atomar sind oder selbst wieder Relationen. Dabei ist zunächst nur von einer einfachen „Schachtelung“ der Relationen ausgegangen worden [Mak77], d. h. die Komponenten der Tupel waren atomar oder 1NF-Relationen. Später wurde sie rekursiv angewendet, so daß die Struktur einer NF^2-Relation wie folgt definiert ist: Eine NF^2-Relation ist eine Menge von Tupeln, deren Komponenten atomar oder selbst wieder NF^2-Relationen sind [SS86, TF86].

Durch die Vorgabe, daß eine Relation stets eine Menge von Tupeln ist, ist die Reihenfolge der strukturierenden Elemente (Mengen und Tupel) immer noch sehr eingeschränkt. Gibt man diese Forderung auf, gelangt man zum erweiterten NF^2-Datenmodell. Die strukturierenden Elemente werden als **Konstruktoren** bezeichnet. Relationales, NF^2- und eNF^2-

[2] Hier wird von der pragmatischen, tupelorientierten Sichtweise, und nicht von der eher theoretisch motivierten Abbildungssichtweise einer Relation ausgegangen [Ull88, Seite 43 f.].



(a) 1NF (b) NF^2 (c) eNF^2

Abbildung 2.1: Reihenfolge der Konstruktoren

Datenmodell unterscheiden sich also strukturell darin, in welcher Reihenfolge die Konstruktoren angeordnet werden dürfen. Graphisch ist der Unterschied in Abb. 2.1 dargestellt[3]. In Abb. 2.1(c) wird durch die drei „Einstiegspunkte“ deutlich, daß die Daten des eNF^2-Datenmodells nicht mehr nur Relationen von Tupeln sind, sondern prinzipiell beliebige Strukturen sein können, im Extremfall sogar einfache atomare Objekte.

Eine weitere Neuerung des eNF^2-Datenmodells ist die Unterscheidung von verschiedenen Konstruktoren für Sammlungen von gleichartigen Objekten: Neben der Menge (im mathematischen Sinn, d. h. ohne Ordnung und Duplikate) wurden Listen und Multimengen zugelassen. Diese Konstruktoren werden unter dem Oberbegriff **Kollektionen** zusammengefaßt.

Eine Implementierung des NF^2-Datenmodells wurde z. B. mit DASDBS [DOP+85, DPS86, Pau88] realisiert, das eNF^2-Datenmodell wurde in AIM-P [DKA+86, Dad88, SLPW89] umgesetzt.

2.1.2 Objekt-orientierte Datenmodelle

Das Aufkommen der objekt-orientierten Programmiersprachen mit Merkmalen wie Vererbung und Kapselung (siehe Abschnitt 1.2) bildete die Basis für die objekt-orientierten Datenmodelle[4] in der Datenbankwelt. Sie stellten zunächst keine evolutionäre Weiterent-

[3] Diese Darstellung ist aus der im AIM-P Projekt entwickelten Darstellung zum Vergleich der Datenmodelle abgeleitet worden (siehe z. B. [Dad88]).

[4] Bei objekt-orientierten Datenmodellen ist der Plural tatsächlich angemessen, da, anders als beim relationalen Datenmodell, eine Fülle von z. T. erheblich differierenden Vorschlägen existieren. Es gibt einige Ansätze, Gemeinsamkeiten herzustellen und Standards zu etablieren [ABJ+90, Cat94, Cat96].

wicklung des relationalen Datenmodells dar, sondern waren ein revolutionärer Neuanfang und erlebten in der Wissenschaft eine geradezu stürmische Entwicklung.

Es zeigte sich jedoch, daß die als Erweiterungen objekt-orientierter Programmiersprachen realisierten Systeme den Anforderungen an eine Datenbank nicht gerecht wurden, da sie grundlegende Konzepte nicht verwirklichten (z. B. fehlten meist nicht-prozedurale Anfragesprachen, Meta-Daten-Verwaltung, Sichten, u. a.). Sie werden daher, etwas abfällig, oft auch als *„persistent storage manager“* bezeichnet [Kim95, S. 5]. Die Forschung ist heute auf Datenbanksysteme gerichtet, die die Konzepte von Datenbanken (Transaktionen, Nebenläufigkeit, physische Datenunabhängigkeit, Dateiorganisation und Zugriffspfade, Optimierbarkeit, usw.) mit den Konzepten der Objekt-Orientierung (Einkapselung, Vererbung, Objektidentität, strukturierte und verschachtelte Objekte) verschmelzen, und nicht das Eine um das Andere erweitern. Dadurch soll eine bessere Integration der beiden Ansätze erreicht werden.

Ein grundlegendes Konzept aller objekt-orientierten Datenmodelle ist die Unterstützung komplexer Objekte (auch „strukturelle Objekt-Orientierung“ genannt [Dit86]). Betrachtet man die strukturelle Komponente objekt-orientierter Datenmodelle genauer, so stellt man fest, daß sie im Wesentlichen nicht über die Erweiterung des relationalen Datenmodells durch das eNF^2-Datenmodell hinausgehen. Im Gegenteil wird insbesondere die Konstruktion von Kollektionen oftmals unzureichend unterstützt, indem nur Listen von Komponentenobjekten zugelassen werden [KDD95, S. 243 ff.]. Diese Unzulänglichkeit resultiert eindeutig aus dem Ursprung in den objekt-orientierten Programmiersprachen, die selbst keine beliebigen Kollektionen erlauben.

Die Object Management Group (OMG), ein Industrie-Konsortium, das inzwischen aus über 400 Firmen besteht und einerseits zur Förderung der Objekt-Orientierung im Software-Bereich, andererseits zur Entwicklung und Definition von Modellen und Schnittstellen für objekt-orientierte Technologien in großen, verteilten Anwendungen und Systemen etabliert wurde, hat ein Objektmodell als Grundlage für seine Arbeit, insbesondere auch in Verbindung mit einer allgemeinen Makler-Architektur für Objektanfragen (*common object request broker architecture*, CORBA) entwickelt. Dieses Kernobjektmodell (*core object model*) basiert auf einigen wenigen Konzepten: Objekte, Operationen, Typen und Subtyp-Bildung. Komplexe Typen können als Strukturen (Tupel) und Vereinigungen (Kollektionen), mit Schablonentypen für Sequenzen und Strings, sowie mit Feldern (*arrays*) gebildet werden.

Von der Object Database Management Group (ODMG), ebenfalls einem Industriekonsortium, wurde ein Objektmodell für Objektdatenbanksysteme als auf dem Kernobjektmodell aufbauendes Profil (*ODBMS profile*) entwickelt. Dieses floß in den Object Database Standard ODMG-93 in der Version 1.2 ein [Cat96] und findet sich auch in der aktuellsten Version ODMG-2.0 [Cat97]. Es erlaubt die Konstruktion komplexer Typen durch Konstruktoren (dort *generators* genannt) für Kollektionen (*set, bag, list* und *array*) und Strukturen (*tuple*). Letztere, die Strukturen, können jedoch nur als sog. „literale Typen“ (*literal types*) definiert werden und besitzen somit keine Objektidentität.

Die objekt-orientierte Programmiersprache Java, die nicht nur wegen ihrer Plattformunabhängigkeit immer mehr an Bedeutung gewinnt, wird auch als Host-Sprache für objekt-orientierte Datenbanken eingesetzt. Die Java Anwendungsschnittstelle für Datenbanken (*Java database connectivity*, JDBC) dient dem Verbindungsaufbau mit Datenbanken, Versenden von SQL-Befehlen und Bearbeiten der Anfrageergebnisse [HCF97, S. 6]. Mit SQLJ [BS99] sollen SQL und Java enger, als nur durch eine Aufrufschnittstelle, verbunden werden. Dabei werden die Möglichkeiten von SQL3 (siehe Abschnitt 2.1.4) zur Definition von komplexen Typen mittels orthogonal anwendbarer Typkonstruktoren noch erweitert, z. B. indem Java-Klassen zur Typdefinition verwendet werden können [HR99, S. 533]. In ODMG-2.0 wurde ebenfalls ein Standard für Persistenz in Java (*Java persistence standard*) aufgenommen.

2.1.3 Das objekt-relationale Datenmodell

Die objekt-relationalen Datenbanken sind die konsequente Fortführung der Entwicklung von erweitert-relationalen Datenbanksystemen unter Einbeziehung einiger Konzepte der Objekt-Orientierung (objekt-relationale Datenbanken werden sogar als Umbenennung der erweitert-relationalen Datenbanken bezeichnet [HR99, S. 533]). Viele sehen als Grundlage der post-relationalen Datenbanktechnologie eine Vereinigung von relationalen und objekt-orientierten Systemen [Kim95, S. 5 ff.].

Stonebraker ordnet objekt-relationalen DBMS vier Hauptmerkmale zu [Sto96, S. xii]:

- Erweiterbarkeit der Basistypen,
- komplexe Objekte,
- Vererbung und
- ein Produktionsregelsystem (*production rule system*).

Die Erweiterbarkeit der Basistypen muß integraler Bestandteil der objekt-relationalen Datenbank sein und sich in dessen Konzepte einfügen, d. h. ein neu definierter Basistyp muß in Bezug auf Ein- und Ausgabe, Optimierung, usw. den eingebauten Basistypen gegenüber gleichwertig sein. Für die neuen Basistypen müssen vom Benutzer Funktionen und Operatoren sowie geeignete Zugriffsverfahren definiert werden können [Sto96, S. 21 ff.].

Um komplexe Objekte zu konstruieren, werden die folgenden Typkonstruktoren als notwendig erachtet[5]:

- Tupel (*record*),
- Kollektion (*set*) und
- Zeiger (*reference*).

Weitere nützliche Typkonstruktoren sind solche für Listen (*lists*), Stapel (*stacks*), Warteschlangen (*queues*) und Vektoren (*arrays*). Komplexe Objekte sollen Parameter und Resultatwerte von benutzerdefinierten Funktionen und Argumente von Operatoren sein können.

[5] Die Typkonstruktoren müssen orthogonal anwendbar sein.

Für komplexe Objekte können Pfadausdrücke (in [Sto96] „kaskadierte Punktnotation“, *cascaded dot notation*, genannt) benutzt werden, um Komponenten zu spezifizieren.

Die Erweiterbarkeit von Basistypen wird von der Definition komplexer Typen aus folgenden Gründen unterschieden:

- Typen ohne innere Struktur, wie z. B. ein Typ für positive, ganze Zahlen, sollen direkt als Basistyp, nicht über den Umweg eines komplexen Objektes, definierbar sein.
- Instanzen komplexer Typen ist eine Objektidentität zugeordnet. Sie können daher referenziert werden, verursachen aber einen zusätzlichen Speicheraufwand.
- Basistypen mit definierten Vergleichsoperatoren können durch traditionelle Zugriffspfade unterstützt werden.

Die Bedeutung der objekt-relationalen Technologie wird auch durch die Aktivitäten der ODMG unterstrichen, die 1998 eine neue Arbeitsgruppe zum Thema „Objekt-relationale (OR) Abbildung“ (*object/relational (OR) mapping*) ins Leben gerufen hat.

2.1.4 SQL3

Die Entwicklung moderner Datenbanksysteme ist eng mit der Entwicklung der Datenbanksprache SQL verknüpft. Ursprünglich wurde SQL als Anfragesprache von „System R“ [ABC+76], einem experimentellen Datenbanksystem, an dem die Realisierbarkeit des relationalen Datenmodells untersucht wurde, unter dem Namen SEQUEL entwickelt [AC75, CGY81]. Der Begriff „SQL“ ist als Abkürzung für „Structured Query Language“ entstanden und ist inzwischen der Name eines Standards geworden. Die Sprache SQL enthält neben Sprachelementen zur Abfrage (*query*) und zum Ändern von Daten (*data manipulation language*, DML) auch solche zum Definieren und Verwalten einer Datenbank (*data definition language*, DDL). Das ist auch der Grund, warum SQL3 als modernste Version von SQL hier in einer Reihe mit den bereits betrachteten Datenmodellen genannt wird.

SQL, das auch als *„universal data speak“* bezeichnet wird, hat sich von seinem einstigen Status eines „De-facto-Standard“ zum offiziellen Standard relationaler Datenbanksprachen entwickelt. Bereits 1986/87 wurde die erste Version von ANSI und ISO verabschiedet, die oft als SQL/86[6] bezeichnet wird. Sie wurde im Laufe der Zeit mehrfach erweitert. 1992 wurde dann die zweite Version, bekannt als SQL/2 oder SQL/92, von der ISO ratifiziert, welche wiederum verschiedene Erweiterungen erfuhr, bzw. immer noch erfährt [DD98], z. B. [Int96]. Diese Version wurde auch als DIN 66315 übernommen. Die Weiterentwicklung führte zu SQL3, das in mehrere Teile untergliedert wurde und dessen Standardisierungsprozeß

[6] Die Schreibweisen der verschiedenen SQL-Versionen differieren in der Literatur erheblich. Hier werden die Schreibweisen aus [DD98] verwendet.

noch nicht abgeschlossen ist[7]. Aber auch von SQL4 ist bereits die Rede — zumeist dann, wenn für irgendwelche Aspekte in SQL3 (noch) keine Einigung erzielt werden konnte. Die Behandlung objekt-orientierter Aspekte in SQL3 wird in [Pis93] untersucht.

Der zweite Teil von SQL3 ist hier am interessantesten (Part 2: SQL Foundation). In ihm werden neue Datentypen, Sub-Tabellen (*subtables*) und ein ADT-Konzept (dort *user defined types*, UDTs, genannt) beschrieben. Erstmals wird in SQL die Konstruktion von komplexen Typen integriert. Mit Typkonstruktoren für Kollektionen können Mengen, Multimengen und Listen (*varying arrays*) definiert werden. Sie sind, wie der Tupel-Konstruktor, orthogonal verwendbar. Dadurch wird die bekannte Tabellendefinition

```
CREATE TABLE ...
```

gleichbedeutend mit der folgenden Typdeklaration [DD98, S. 542]:

```
MULTISET ( ROW ( ... ) )
```

Referenztypen erlauben eine eindeutige Identifikation von Tupeln und unterstützen das direkte Navigieren über Pfadausdrücke. Ein benutzerdefinierter Typ kann als sog. „Tupeltyp“ (*row type*) einer Basisrelation zugeordnet werden. Da die Tupel eines Tupeltyps eine Identität besitzen, können Referenztypen dazu benutzt werden, um auf solche Tupel zu verweisen [HR99, S. 531].

SQL3 wird auch als Ansatz verstanden, die vielfältigen Probleme der heute weitverbreiteten heterogenen Datenquellen zu lösen [HR99, S. 531 f.]. Als Standard für ein Datenmodell, das auch die Strukturierung komplexer Objekte erlaubt, erfüllt es die Anforderungen des Modells der universellen Speicherung (*universal storage model*), bei dem alle Daten in einem DBMS integriert gespeichert und verwaltet werden. „Dieser [Ansatz] verlangt eine vorherige Transformation, Konversion oder vereinheitlichte Darstellung aller verschiedenartigen Datentypen, die oft eine komplexe Struktur und benutzerdefinierte Semantik besitzen.“ [HR99, S. 530] Aber auch das Modell des universellen Zugriffs (*universal access model*), bei dem durch eine Middleware-Komponente eine homogenisierte Sicht auf die heterogenen Datenquellen abgeleitet wird, beschränkt sich auf die Vereinheitlichung der strukturellen Heterogenität, da die benutzerdefinierte semantische Heterogenität (noch) nicht handhabbar ist [HR99, S. 536 ff.].

2.1.5 ESCHER$^+$

Viele objekt-orientierte Datenmodelle haben Schwächen im strukturellen Bereich, die die Modellierung komplexer Objekte erschweren. Ein relativ einfaches Datenmodell, das bereits die meisten strukturellen Aspekte von komplexen Objekten abdeckt, ist das eNF^2-Datenmodell [DKA+86, PA86, PD89].

[7] Der Arbeitsentwurf (*working draft*) von ISO und ANSI wurde bereits 1995 veröffentlicht [Mel95], 1996 dann der Entwurf der Kommission (*committee draft*) [Mel96]. In [EM98] wurde die Annahme als internationaler Standard für Anfang 1999 vorausgesagt [HR99, S. 531].

Diese Arbeit baut auf das Datenmodell ESCHER$^+$ [The96], das eine Weiterentwicklung des eNF^2-Datenmodells ist, auf (siehe Abschnitt 2.3.1), und betrachtet strukturell komplexe Objekte gemäß Abb. 2.1(c) auf Seite 28. ESCHER$^+$ subsumiert im Wesentlichen auch die strukturellen Gestaltungsmöglichkeiten objekt-relationaler Datenmodelle, insbesondere auch die des SQL3-Standards. Die Anforderung an das Visualisierungsverfahren ist demnach, daß es alle gemäß ESCHER$^+$ komplex strukturierten Datenobjekte darstellen kann.

2.2 GUI-Techniken

Im Folgenden wird einen kurzer Überblick auf die historische Entwicklung der Programmierung graphischer Benutzerschnittstellen gegeben. Natürlich gibt es eine ganze Reihe von Übersichtsartikeln, die die Entwicklung von graphischen Benutzerschnittstellen, bzw. meist allgemeiner der Mensch-Maschine-Schnittstelle, und der zugehörigen Werkzeuge beschreiben. Hier eine nur unvollständige Liste:

- [MHC96] und [Mye96] geben einen historischen Überblick der Techniken, Anwendungen und Werkzeuge der HCI. Es wird explizit die wichtige Rolle der universitären Forschung in diesem Bereich betont. Als grundlegende Interaktionskonzepte werden die direkte Manipulation graphischer Objekte (1963)[8], die Maus (1965) und das Fensterkonzept (1968) angesprochen. Als Arten von Anwendungen werden Text-Editoren (1962), Zeichen- und Tabellenkalkulationsprogramme (1963 bzw. 1977), Hypertext- und CAD-Systeme (1945/1965 bzw. 1963) sowie Computerspiele (*video games*, 1962) genannt. In der Wissenschaft noch immer aktuelle Konzepte sind die Erkennung von Gestik (*gesture recognition*, 1963), Multimedia (1968), dreidimensionale Darstellungen (1963), virtuelle Welten (*virtual reality* und *augmented reality*, 1965), kooperatives Arbeiten (*computer supported cooperative work*, CSCW, 1968) und die Verarbeitung natürlicher Sprache (1980). Die genannten Konzepte für Werkzeuge sind UIMS (*user interface management systems*, 1966/1982), Toolkits (1974/1984), Interface Builder (1979) und Komponenten (*component architectures*, 1988).

- [Mye94] ist eine ausführliche Klassifizierung von Werkzeugen für Benutzerschnittstellen mit Beispielen sowohl aus dem kommerziellen, als auch dem wissenschaftlichen Bereich. Der Technical Report enthält außerdem eine Diskussion des Erfolgs der einzelnen Werkzeugklassen. Zunächst werden drei Schichten unterschieden: Fenstersystem, Toolkits und höhere Werkzeuge. Toolkits werden als Aufrufschnittstellen, Komponentenbibliotheken (*widget sets*[9]) oder virtuelle Toolkits klassifiziert. Zu den höheren Werkzeugen zählen sprachbasierte Ansätze, *application frameworks* [Mye94, S. 24] und solche mit direkter graphischer Spezifikation.

[8] Die Jahreszahlen geben jeweils den ungefähren Zeitpunkt der Erfindung bzw. ersten Publizierung an.

[9] Das englische Kunstwort *widget* ist als Zusammenziehung aus dem Begriff *window gadget* entstanden. Dieser kann mit „Fenster Kinkerlitzchen“ übersetzt werden und deutet auf das Konzept der Fenster als grundlegenden Baustein graphischer Benutzerschnittstellen hin.

- [BGBG95, Pre93, Gol88, RLPM96] sind historische Betrachtungen zur Entwicklung der HCI, insbesondere auch graphischer Benutzerschnittstellen. Sie betrachten Konzepte und Systeme wie z.B. MEMEX (1945), Whirlwind (1950), Sketchpad (1963), Xerox Star (1981), Apple Macintosh (1984), WYSIWYG (*what you see is what you get*), Netzwerke, usw.

2.2.1 Vorgeschichtliches

Bevor graphische Benutzerschnittstellen aufkamen, die die starre, zeilen- und spaltenorientierte Textausgabe („ASCII-Terminal“) durch eine Ausgabe mit Pixeladressierung ersetzten und damit die Darstellung beliebiger Polygonzüge und Flächenfärbungen erlaubten, waren kommandozeilenorientierte und maskenbasierte Dialoge die bestimmenden Elemente der Mensch-Maschine-Kommunikation. Die vergleichsweise einfache Programmierung dieser Systeme orientiert sich im Wesentlichen an der Eingabe-Verarbeitung-Ausgabe-Schleife, die die Abläufe bestimmt. Die heutige, kompliziertere weil ereignisgesteuerte Programmierung ist eine Folge der graphischen Benutzerschnittstellen, bei denen man „immer überall hin klicken“ kann („In jeder Situation kann fast alles passieren — was für den Benutzer Komfort bedeutet, ist für den Programmierer eine erschreckende Vorstellung.“ [GKKZ92, S. 23]).

2.2.2 Fenstersysteme

Fenstersysteme (*window systems*) sind die Grundlage graphischer Benutzerschnittstellen. Sie unterteilen den Bildschirm in mehrere logische Bereiche (Fenster), die einen eigenen Kontext haben. Sie gehen auf den Anfang der 70er Jahre zurück (COPILOT, EMACS) und sind mit Namen wie Smalltalk [Kay93], Cedar Window Manager [SZBH86] und „Xerox Star“ [JRV+89] verbunden. Die erste Art der Programmierung graphischer Benutzerschnittstellen, und damit der Startpunkt der historischen Entwicklung, war sicherlich die direkte Programmierung des Fenstersystems über dessen Programmierschnittstelle (*application programming interface*, API). Auch heute noch wird ein großer Teil der graphischen Benutzerschnittstellen direkt, ohne die Hilfe spezieller Tools oder Techniken, programmiert (26% nach [MR92]). Auch wenn sehr spezielle Teile von Benutzerschnittstellen implementiert werden, muß oftmals auf die grundlegenden Funktionen der Fenstersysteme zurückgegriffen werden. Heute verbreitete Fenstersysteme sind z.B. Microsoft Windows, das X Window System im UNIX-Umfeld, der Apple Macintosh und IBMs OS/2 Presentation Manager.

2.2.3 Toolkits

Eine höhere Abstraktionsebene wurde mit der Einführung von **Toolkits** geschaffen. Ein Toolkit ist eine Sammlung von Funktionen und Vereinbarungen, deren Grundelemente jetzt

nicht mehr Polygone, Färbungen und Tastaturcodes sind, sondern die dem Benutzer angebotenen Elemente graphischer Benutzerschnittstellen, wie Rollbalken, Menüs, Knöpfe, usw. Diese Elemente können über ein Programmierschnittstelle kombiniert werden, wodurch dann die graphische Benutzerschnittstelle konstruiert wird. Toolkits können sehr unterschiedliche Abstraktionsniveaus haben, von einfachen graphischen Elementen wie Knöpfen bis zu komplexen Konstruktionen wie kompletten Text-Editoren. Seit der Entstehung von Toolkits werden mit der Entwicklung im Programmiersprachen-Bereich auch immer wieder neue Toolkits entworfen, z. B. als objekt-orientierte Klassenbibliotheken. Beispiele für Toolkits sind

- X-Windows: Athena Widget Set, OSF Motif, OpenLook, Tcl/Tk, Latitude
- Apple Macintosh: Toolbox, CPLAT, Tcl/Tk, Latitude
- Microsoft Windows: MFC (Microsoft Foundation Class library), OWL (Borland Object Windows Library), CPLAT, Tcl/Tk

Um die Programmierung plattformübergreifender Benutzerschnittstellen zu erleichtern, sind einige Toolkits auf mehreren Plattformen verfügbar, z. B. CPLAT für Microsoft Windows und Macintosh oder Latitude für X Window System und Macintosh. Es wird dann auch von sog. „virtuellen Toolkits“ (*virtual toolkits*) gesprochen. Insbesondere ist hier die Skriptsprache Tcl/Tk [Ous94] zu erwähnen, die mit wenigen Einschränkungen auf allen drei oben genannten Plattformen sogar im Quellcode kompatibel ist. Nach [MR92] werden heute 34% der Benutzerschnittstellen durch die Verwendung von Toolkits programmiert. Ein Problem bei der Verwendung von Toolkits ist deren oftmals sehr komplexe Schnittstelle. Trotz der vielen Funktionen, die zur Verfügung gestellt werden, sind spezielle Anforderungen an Benutzerschnittstellen manchmal nicht, oder wieder nur durch die direkte Programmierung der Fenstersystemschnittstelle, zu realisieren.

2.2.4 Interface Builder

Basierend auf Toolkits wurden **Interface Builder** entwickelt. Ein Interface Builder ist quasi ein CAD-System zur Konstruktion graphischer Benutzerschnittstellen. Dazu stellt es dem Designer einen graphischen WYSIWYG-Editor zur Verfügung, mit dem er die Elemente der zu erzeugenden Benutzerschnittstelle aus den Komponenten eines Toolkits auswählen, positionieren und konfigurieren kann. Viele Interface Builder können auch von Nicht-Programmierern benutzt werden, wodurch das Erstellen der graphischen Benutzerschnittstelle auch Nicht-Fachleuten wie Arbeitswissenschaftlern, Ergonomen und Psychologen zugänglich wird. Interface Builder sind meist gut geeignet, um das Layout der statischen Komponenten einer Benutzerschnittstelle zu generieren. Die dynamischen Aspekte können oft gar nicht, oder nur unzureichend, modelliert werden. Ein häufig verwendeter Ansatz ist hier, daß ein Interface Builder Quellcode in einer bestimmten Zielsprache generiert. Der Quellcode benutzt das verwendete Toolkit und enthält bei der Spezifikation der dynamischen Anteile nur Coderümpfe, die dann von Hand ausprogrammiert werden müssen. Die Programmierung dieser sog. „Rückrufprozeduren“ (*call-back procedures*) ist meist

schwierig („[...] but things become difficult as soon as the designer has to write code, for example, to provide the behaviour for the interface elements." [Ous98, S. 28]). Einige Interface Builder erlauben die Generierung von Quellcode für verschiedene Plattformen, so daß plattformübergreifende Benutzerschnittstellen erzeugt werden können [Chi95]. Interface Builder sind nur mit einem relativ geringen Anteil von 14% bei der Programmierung graphischer Benutzerschnittstellen vertreten [MR92]. Existierende Interface Builder sind MacFlow, Prototyper (Macintosh), UIMX, XF (UNIX, X-Windows), WindowsMAKER (Microsoft Windows) und der NeXT Interface Builder.

Als spezielle Interface Builder sollen hier noch Werkzeuge zur Programmierung von Benutzerschnittstellen für Datenbanken erwähnt werden (siehe auch Abschnitt 2.2.7). Diese generieren meist formularbasierte Datenbankanwendungen, die der Struktur einer Datenbank angepaßt sind. Es gibt produktabhängige (z. B. ORACLE Forms) und -unabhängige (z. B. Uniface) Ausprägungen dieser speziellen Interface Builder.

2.2.5 User Interface Management Systems

Ein weitergehender Ansatz sind die **User Interface Management Systems** (UIMS)[10]. Die Analogie des Namens zu den Datenbanksystemen (Database Management Systems, DBMS) ist nicht von ungefähr: Die Schaffung einer logischen Unabhängigkeit von Anwendungen und ihren Benutzerschnittstellen ist eines der Ziele von UIMS, was sich in einer Architektur widerspiegelt, die an die klassischen Drei-Ebenen-Architektur von DBMS angelehnt ist. Ein UIMS ist ein umfassenderes Werkzeug als ein Interface Builder. Es soll eine automatische Generierung einer Benutzerschnittstelle, basierend auf einer Beschreibung auf hohem Abstraktionsniveau, erlauben. Diese Benutzerschnittstelle soll ohne Veränderung der zugrunde liegenden Anwendung modifiziert bzw. weiterentwickelt werden können. Das UIMS soll sowohl die Entwicklung als auch die Ausführung der Benutzerschnittstelle unterstützen, d. h. es soll sowohl in der Entwurfs- und Test- als auch in der Laufzeitphase verwendbar sein. Dazu kann es Komponenten enthalten, die zu verschiedenen Zeitpunkten operieren (*design time*, *run-time*, *after run-time*[11]). Knapp ein Drittel (27%) aller graphischen Benutzerschnittstellen werden mit UIMS erstellt [MR92]. Beispiele für UIMS sind Hypercard und Easel.

2.2.6 Weitere Konzepte

Ein Zwischenschritt auf dem Weg von Toolkits zu UIMS, bzw. ein Teil der UIMS, sind die **User Interface Languages** (UIL). Diese Sprachen wurden entwickelt, um Benutzerschnittstellen und ihr Layout auf einem hohen Abstraktionsniveau zu beschreiben. Aus

[10] Der Begriff UIMS geht auf den Seeheim-Workshop [End84, Mac85, Pla85] 1983 zurück [Kli96, S. 175].
[11] Nach Ausführung eines Programms, z. B. Fehlerbehebung oder Evaluation [Mye94, S. 19].

diesen Beschreibungen sollen die konkreten Benutzerschnittstellen dann automatisch generiert werden. Ein Beispiel ist die OSF Motif UIL.

Weiterhin gibt es Spezialformen und Symbiosen der oben genannten Ansätze. Der UIMX Interface Builder etwa generiert eine Beschreibung der Benutzerschnittstellen in UIL, und integriert einen Interpreter, um dynamische Aspekte von Benutzerschnittstellen abzubilden. Der Interface Builder XF ist mit Tcl/Tk erstellt und generiert selber Tcl/Tk-Code. Dadurch ist er prinzipiell beliebig erweiterbar, und die generierten Benutzerschnittstellen können beim Erstellen sofort getestet werden.

Einen Spezialfall bilden Generatoren für Benutzerschnittstellen von Datenvisualisierungen. Darunter wird die Visualisierung von dynamischen Prozeßdaten, z. B. von Simulationen oder in Prozeßwarten, durch eine graphische Benutzerschnittstelle verstanden. Hierfür existieren spezielle Werkzeuge, die geeignete Schnittstellenelemente zur Verfügung stellen.

Auch Werkzeuge zum Erstellen von HTML-Dokumenten können als Interface Builder bezeichnet werden, wenn sie die heutigen Erweiterungen und Fähigkeiten von HTML integrieren (*„Because Web browsers can double as GUI builders across multiple platforms, many people have created quite complex applications that run within a browser window.“ [Nie99, S. 68]*).

Vielen Entwicklungen im Bereich der Programmierung graphischer Benutzerschnittstellen ist gemeinsam, daß eine automatische Generierung der Benutzerschnittstellen angestrebt wird. Es gibt eine große Anzahl von Forschungsansätzen, die Teile eines UIMS mehr oder weniger umfassend behandeln. Genereller Nachteil dieser Systeme ist, daß die generierten Benutzerschnittstellen so gut wie immer einer manuellen Nachbesserung bedürfen. Diese ist jedoch in generierten Programmen oft sehr umständlich. Außerdem erfordern die meisten Systeme das Erlernen einer speziellen Sprache, in der die Benutzerschnittstelle beschrieben werden muß. Im Folgenden wird einen kurzer Überblick über einige Forschungsansätze gegeben:

- RBE [KZ95a] ist ein Domänenkalkül über Komponenten von Benutzerschnittstellen. Mit dieser Sprache ist eine deklarative Spezifikation der Semantik von Benutzerinteraktionen möglich.
- Jade [VZM90] generiert Benutzerschnittstellen automatisch, aber erlaubt die nachträgliche Änderung mit einem graphischen Editor.
- Das User Interface Design Environment (UIDE, [SFG93]) basiert auf einer speziellen sprachlichen Spezifikation der Anwendungssemantik, generiert daraus Pre- und Post-Konditionen für Operationen und schließlich die Benutzerschnittstelle. UIDE erlaubt Transformationen der Benutzerschnittstelle unter Erhaltung von deren Konsistenz.
- ITS [WBB+90] trennt Inhalt und Stil der Benutzerschnittstelle durch Einteilung in mehrere Schichten und realisiert damit eine regelbasierte Generierung von Interaktionstechniken.

- Rendezvous [HBP+93] ist ein Toolkit, das auf die Programmierung verteilter Benutzerschnittstellen spezialisiert ist, die synchrones Arbeiten unterstützen.

- Amulet [MMM+97] ist ein erweiterbares, auf Constraints basierendes Toolkit (*extensible constraint based user interface toolkit*) für C++. Es kann in X Window System, Microsoft Windows und Macintosh Umgebungen eingesetzt werden. Amulet enthält Merkmale wie ein sog. „Prototyp-Instanz Objektmodell“ (*prototype-instance object model*), höhere Konzepte der Eingabebehandlung wie automatisches Rückgängigmachen (*undo*) sowie Unterstützung für Animationen und Erkennung von Gestik.

- Garnet [MGD+90] ist eine Entwicklungsumgebung für Benutzerschnittstellen auf der Basis der funktionalen Programmiersprache Common Lisp für das X Window System und den Macintosh. Es enthält ein Toolkit, mehrere Komponentenbibliotheken und interaktive Designwerkzeuge, mit denen Teile der Benutzerschnittstellen ohne Programmierung erstellt werden können.

- Das Simple User Interface Toolkit (SUIT, [PCD92]) wurde speziell für die Ausbildung entwickelt und ist aufgrund seines Designs besonders einfach erlernbar. Es ist für das X Window System, den Macintosh und Microsoft Windows verfügbar.

- GENIUS [JWZ93] generiert dynamische Benutzerschnittstellen für Datenbankanwendungen aus einem erweiterten Datenmodell. Dazu benutzt es ein Expertensystem, das sich an aus existierenden Richtlinien abgeleiteten, gespeicherten Regeln orientiert, um aus Sichtdefinitionen die statischen Komponenten der Benutzerschnittstellen zu erzeugen. Das dynamische Verhalten wird mit Dialognetzen, die auf Petri-Netzen basieren, spezifiziert. Die Ausgabe von GENIUS wird für ein bestehendes UIMS generiert.

Ein Trend in der Forschung zur Programmierung graphischer Benutzerschnittstellen ist die Integration von sprachbasierten und graphischen Ansätzen. Erstere geben dem Designer oft nicht genug Freiheiten, letztere meist zu viele. Die Hoffnung ist, durch deren Kombination die Vorteile aus beiden Ansätzen zu vereinen [Mye94, S. 34 f.].

Eine weitere Forderung für Systeme zur Programmierung graphischer Benutzerschnittstellen ist die Einbeziehung oder Ermöglichung der Benutzerkonfigurierbarkeit bzw. -erweiterbarkeit. Diese Aussage gilt insbesondere auch für datenbankorientierte Umgebungen [Row92, S. 9] und soll quasi durch eine Programmierbarkeit der fertigen Benutzerschnittstelle erreicht werden [Mye94, S. 35 f.]. Im Gegensatz zu mit Systemprogrammiersprachen erstellten Benutzerschnittstellen werden die mit Tcl/Tk erstellten dieser Forderung in besonderer Weise gerecht („[...] GUI toolkits based on languages such as C or C++ have proven hard to learn, clumsy to use, and inflexible in the results they produce.“ [Ous98, S. 28]). Die interpretierende Skriptsprache erlaubt nämlich die Konfiguration und Erweiterung von Anwendungen selbst zur Laufzeit. Das in dieser Arbeit vorgestellte Visualisierungsverfahren ist ebenfalls unter den Gesichtspunkten Konfigurierbarkeit und Erweiter-

barkeit entwickelt worden. Die prototypische Implementierung wurde zudem in Tcl/Tk ausgeführt, so daß die obige Forderung sicher erfüllt wurde.

Goyal et al. stellen die Frage, ob die Programmierung graphischer Benutzerschnittstellen ein für die Datenbankforschung relevantes Problem ist („Is GUI Programming a Database Research Problem?“ [GHK+96]). Sie argumentieren für die Ausnutzung von Datenbanktechniken zur Implementierung komplexer Benutzerschnittstellen, u. a. Pufferverwaltung für zu verändernde Schnittstellenobjekte, konkurrierende und verteilte Strategien für Benutzerschnittstellen, Serialisierbarkeit für Mensch-Maschine-Interaktionen, Indextechniken für Bildschirmausgaben und Materialisierung von Sichten zur Effizienzsteigerung. Die enge Kopplung von Visualisierung und Datenbank, die in dieser Arbeit beschrieben wird, ist eine gute Voraussetzung für weitere Untersuchungen im Sinne von Goyal et al. Das Konzept der Selbstreferenzierung, das in der Datenbankwelt bei der Behandlung von Meta-Daten verwendet wird, erweist sich auch für das vorgestellte Visualisierungsverfahren als sehr hilfreich.

Das in dieser Arbeit vorgestellte Visualisierungsverfahren ist z. B. mit der Erzeugung der statischen Komponenten in GENIUS vergleichbar, da diese auch dort aus einer strukturellen Beschreibung von Datenobjekten generiert werden. Im Gegensatz zu GENIUS, das nur einfache Datenobjekte in seinen statischen Masken anzeigt, soll hier jedoch auch die Visualisierung von und Interaktion mit komplexen Datenobjekten unterstützt werden.

2.2.7 Entwicklung von Benutzerschnittstellen für Datenbanken

Die Betrachtung der GUI-Techniken wird in diesem Abschnitt mit einem kurzen Überblick der historischen Entwicklung von Benutzerschnittstellen für Datenbanken in Form einer teilweisen Zusammenfassung des Tutorials „The Evolution of User Interface Tools for Database Applications“ von Krishnamurthy und Zloof auf der VLDB 1995 [KZ95b] abgeschlossen. Ausgehend von programmgenerierten Reportausdrucken und „schlüsselfertigen“, formularbasierten Anwendungen ohne verteilter Ausführung (60er und 70er Jahre), über Ad-hoc- und formularbasierte Anfragen mit Sichtenkonzept und parametrisierbare Reports (70er und 80er Jahre), entwickelten sich die Benutzerschnittstellen für Datenbanken zu flexiblen, interaktiven, konsistenten und intuitiven Benutzerschnittstellen (seit den 80er Jahren). Die Entwicklungen waren in ihrer Zeit jeweils durch neu entstandene Konzepte und Software- und Hardware-Techniken motiviert, z. B. durch das relationale Datenmodell, die Objekt-Orientierung, durch Fensterumgebungen mit ereignisgesteuerten Merkmalen, leistungsstarke Rechner und Netzwerke, De-facto-Standards, usw.

Die Entwicklung wird von Krishnamurthy und Zloof auch aus den folgenden Perspektiven betrachtet:

Entwickler: Von Anwendungen, die in traditionellen programmiersprachenbasierten Umgebungen implementiert wurden über mit einfachen Werkzeugen wie Embedded SQL und sog. „Sprachen der vierten Generation“ (*4th generation languages*, 4GLs, z. B.

Windows/4GL von Ingres [Ing91]) erstellten Anwendungen und mit komplexeren Werkzeugen wie ereignisgesteuerten Sprachen, z. B. Visual Basic, erstellten Anwendungen ging die Entwicklung zu komplexen Client-Server-Anwendungen über, die in Umgebungen wie PowerBuilder oder NextStep erstellt werden.

Produkte: Von Aufsätzen zu bestehenden Produkten wie Formular- und Reportgeneratoren, über aus Datenbankprodukten hervorgegangenen Werkzeugen (z. B. Oracle Forms) und Programmierumgebungen, die Datenbankfunktionalität integrieren, wie z. B. Delphi, sind die Entwicklung zu Produkten hin, die als Entwicklungsumgebungen für Datenbankanwendungen erstellt werden, wie PowerBuilder oder SQL Windows.

Schnittstellenmerkmale: Von kommandozeilenorientierten Anwendungen ausgehend wurden Formulare mit einfachen Menüs und Funktionstasten entwickelt. Weitere Schritte waren die syntaxgesteuerte Anfragekonstruktion, kontextsensitive Menüs und graphische Schnittstellen mit Pop-Up-Menüs, *cut* & *paste* und überlappenden Fenstern (z. B. PICASSO [RKS+91]). Dabei wurden zunächst rudimentäre Stileigenschaften (*look* & *feel*: Menünamen, *short-cuts*, Rollbalken, usw.) und dann Widget-Typen (z. B. Combo-Box) standardisiert. Auch die Manipulation zusammengesetzter Objekte (z. B. in Tabellenkalkulationen und Reports) sowie das Abbilden von Daten in Widgets (*wizards*) wurden integriert.

Datenbankmerkmale: Die Administration und der Entwurf von Datenbanken wurden zunächst vernachlässigt und sind erst seit den 90er Jahren verbreitet. Die Dateneingabe erfolgt oft auch heute noch über Tabellen oder ist formularbasiert und hat inzwischen Merkmale wie z. B. Werteauswahl, Plausibilitätsprüfungen und Default-Werte. Ad-hoc-Anfragen kamen ab den 70er Jahren zunächst mit eingeschränkten Möglichkeiten (Fremdschlüssel, Equi-Join), auf und entwickelten sich z. B. in Form von QBE [Zlo82]. Die Datenausgabe basierte anfangs auf Reports, Formularen und tabellarischen Darstellungen und umfaßt inzwischen Graphiken, Visualisierungen und auch das interaktive Browsen.

2.3 Einige Konzepte von ESCHER

In Abschnitt 1.3 ist die historische Entwicklung des Forschungsprojekts ESCHER dargestellt worden. Aus dieser ergibt sich die Erklärung der Beschreibung von ESCHER als „graphischer Datenbank-Editor für komplexe Objekte“:

- graphisch — Die komplexen Objekte werden mit einer graphischen Benutzerschnittstelle visualisiert.
- Datenbank — Die komplexen Objekte sind in einer Datenbank persistent gespeichert.

- Editor — Die Interaktion orientiert sich an den Konzepten eines Editors und schließt auch das Ändern (Editieren) der komplexen Objekte ein.
- komplexe Objekte — Die Gegenstände der Interaktion sind komplex strukturierte Objekte.

Es handelt sich also um die Visualisierung von und die Interaktion mit komplex strukturierten Objekten, die in einer Datenbank gespeichert sind.

Ein Editor ist ein Software-Werkzeug zum Ändern von Daten. Im Allgemeinen können mit einem Editor beliebige Daten bearbeitet werden (Text-Editor), es gibt aber auch spezialisierte Editoren, die nur speziell strukturierte Daten bearbeiten können (z. B. der Editor eines Textverarbeitungsprogramms wie FrameMaker, oder ein HTML-Editor). Beide Editor-Typen zeichnen sich dadurch aus, daß sie einen navigierenden Zugriff auf Daten ermöglichen, wobei eine aktuelle Position durch den sog. „Cursor“ markiert wird.

Mit ESCHER ist ein anwendungsunabhängiges, generisches und intuitives Editieren von komplex strukturierten Objekten möglich. Das Editieren ist generisch, da mit *einem* Interaktionskonzept *alle* komplex strukturierten Objekte editiert werden können. Es umfaßt das Suchen und Ersetzen, Entfernen und Einfügen, Ändern, Vertauschen, und Sortieren von Datenobjekten und ihren Sub-Objekten.

Der navigierende Zugriff auf Daten, der durch den Begriff Editor bereits angedeutet wird, widerspricht dabei nicht dem deskriptiven Zugriff auf Daten, der v. a. mit dem relationalen Datenmodell verbunden ist. Vielmehr wird die Navigation auf komplexen Objekten als angemessene Art und Weise betrachtet, mit den Ergebnissen eines deskriptiven Datenbankzugriffs umzugehen. Eine der zukünftigen Richtungen bei der Entwicklung von Benutzerschnittstellenwerkzeugen für Datenbankanwendungen heißt „Query and Navigation“ [KZ95b, Folie 40]. In [KDD95, S. 241] heißt es: „The combination of associative access (find objects with certain properties in the database) and navigational access (to further investigate the objects found) is indispensable for future applications.“

Angemessen bedeutet auf jeden Fall, daß die komplexen Objekte mit den Mitteln einer graphischen Benutzerschnittstelle dargestellt (visualisiert) werden. Das Hauptproblem, das beim deskriptiven Datenbankzugriff auftritt, gilt es auch für die Visualisierung zu lösen: Die Performance muß akzeptabel bleiben, denn von den Reaktionszeiten ist der Grad der Interaktivität bei direkter Manipulation von Daten abhängig [KZ95b]. Die Navigation muß durch ein Gefühl für den Kontext im komplexen Objekt unterstützt werden, um dem Benutzer seine Position und Bewegungsmöglichkeiten zu verdeutlichen („[...] it is also neccesary to have a strong sense of structure and navigation support [...] so the users know where they are, where they have been, and where they can go.“ [Nie99, S. 67]).

Um es noch einmal deutlich zu machen: Wenn hier von einem navigierenden Zugriff gesprochen wird, ist damit das Browsen in einem graphisch visualisierten Datenobjekt gemeint.

Im Gegensatz dazu wird in der (Datenbank-) Literatur unter einem navigierenden Zugriff fast immer die Iteration in Mengen über Konstrukte einer Programmiersprachenschnittstelle verstanden, also das iterative „Sammeln“ von Daten im Kontrast zum deskriptiven Formulieren von Anfragen.

Im Folgenden werden einige hier relevante Konzepte von ESCHER besprochen. Das realisierte Visualisierungskonzept, das Ausgangspunkt der vorliegenden Arbeit ist, wird in Abschnitt 3.1 besprochen.

2.3.1 Das Datenmodell

Für die Definition von komplex strukturierten Datenobjekten wird von ESCHER das eNF^2-Datenmodell zugrunde gelegt. Explizite, formale Beschreibungen eines Datenmodells wurden in zwei Arbeiten angefertigt, die sich mit „Typerweiterungen im eNF^2-Datenmodell“ [Pau94] und dem Thema „Semantische Anreicherung eines Datenmodells für komplexe Objekte“ [The96] beschäftigen. Beide Arbeiten behandeln auch die Definition einer operationalen Basis für ESCHER.

Die vorliegende Arbeit bezieht sich weitestgehend auf das in [The96] entwickelte Datenmodell ESCHER$^+$. Neben der formalen Definition sowohl der strukturellen als auch der operationalen Aspekte des Datenmodells wird dort eine Erweiterung des eNF^2-Datenmodells, der strukturellen Basis von ESCHER$^+$, um die folgenden Konzepte vorgenommen:

- Objekttypen,
- Aufzählungstypen,
- variante Typen und
- Felder (Arrays), bzw. deren Spezialformen Matrizen und Vektoren.

Die varianten Typen werden hier nicht weiter betrachtet. Sie werfen eine Fülle von interessanten Fragestellungen bzgl. ihrer Visualisierung auf, die sicherlich einer Untersuchung Wert sind. Trotzdem bleibt die Betrachtung zunächst auf die anderen Typen beschränkt, um die Beschreibung des Visualisierungsverfahrens nicht zu überladen. Die Anwendung auf variante Typen kann dann als Überprüfung der Erweiterbarkeit des Visualisierungsverfahrens betrachtet werden (siehe Abschnitt 6.3).

Objekttypen, Aufzählungstypen und Vektoren sind inzwischen auch Gegenstand der aktuellen Standardisierungsbemühungen im kommerziellen Umfeld. Sie werden in SQL3 unter den Namen *user defined types*, *collection types* und *varying arrays* behandelt (siehe Abschnitt 2.1.4).

Abweichend von [The96] wird davon ausgegangen, daß nicht nur Tupelkomponenten mit Namen versehen sind, insbesondere sind auch Elementtypen von Kollektionen benannt. Diese Erweiterung wirkt sich auf die dortigen Definitionen 3.9 von komplexen Typen [The96,

S. 83 f.] und 3.10 von Typausdrücken [The96, S. 84] aus. Die angepaßten Definitionen werden hier wiedergegeben, da sie Grundlage für die strukturellen Definitionen in Abschnitt 3.4 sind. Ihre Form ist weitgehend unverändert aus [The96] übernommen worden.

In Definitionen 3.9 wird in [The96] die rekursive Konstruktion von komplexen Typen, die dort auch Strukturen genannt werden, beschrieben, wobei die Konstruktoren orthogonal angewendet werden können.

$\mathcal{A}$ ist die Menge der Attributnamen [The96, S. 78], TYPES, BASETYPES $\subset$ TYPES, ENUMTYPES $\subset$ TYPES und DEFTYPES $\subset$ TYPES sind die Mengen aller Typen bzw. der Basis-, Aufzählungs- und Objekttypen [The96, S. 72 ff.]. Für Strukturen gilt dann:

Definition 3.9′ (Strukturen)

Es sei CONSTR $= \{c_{\text{set}}, c_{\text{bag}}, c_{\text{list}}, c_{\text{tuple}}, c_{\text{array}}\} \subseteq$ TYPES die gegebene Menge von parametrisierbaren Typen (Konstruktoren).

Es sei $S_0 :=$ BASETYPES $\cup$ ENUMTYPES $\cup$ DEFTYPES.

Ferner seien $s, s_1, \ldots, s_k \in \bigcup_{0 \le j \le i} S_j$. Dann ergibt sich S_{i+1} wie folgt:

$$
\begin{aligned}
& c_{\text{set}}(a, s) \in S_{i+1}, \text{ mit } a \in \mathcal{A} && \text{(i)}\\
& c_{\text{bag}}(a, s) \in S_{i+1}, \text{ mit } a \in \mathcal{A} && \text{(ii)}\\
& c_{\text{list}}(a, s) \in S_{i+1}, \text{ mit } a \in \mathcal{A} && \text{(iii)}\\
& c_{\text{tuple}}((a_1, s_1), \ldots, (a_k, s_k)) \in S_{i+1}, && \text{(iv)}\\
& \quad \text{mit } a_i \in \mathcal{A} \text{ paarweise verschieden für } 0 \le i \le k\\
& c_{\text{array}}(a, s, (n_1, m_1), \ldots, (n_k, m_k)) \in S_{i+1}, && \text{(v)}\\
& \quad \text{mit } a \in \mathcal{A} \text{ und } n_i, m_i \in \mathbb{Z},\ n_i \le m_i \text{ für } 1 \le i \le k,\ k \in \mathbb{N}_0\\
& \text{nichts anderes liegt in } S_{i+1} && \text{(vi)}
\end{aligned}
$$

Es ist dann $\mathcal{S}(\text{TYPES}) := \bigcup_{i \ge 0} S_i$ die Menge aller **Strukturen** über TYPES. Die Strukturen aus CPLXTYPES $= \mathcal{S}(\text{TYPES}) - S_0$ heißen **komplexe Strukturen**, die Strukturen aus SIMPLETYPES $= S_0$ heißen **einfache Strukturen**. □

In Definition 3.10 wird die Funktion *name*, die vorher nur auf TYPES definiert war, auf $\mathcal{S}$(TYPES) fortgesetzt. *name* ordnet damit jedem durch Definition 3.9 konstruierbaren Typ eine textuelle Beschreibung zu, die in [The96] Typausdruck heißt. Die Definition ändert sich wie folgt:

Definition 3.10′ (Typausdrücke)

Die Funktion $name : \mathcal{S}(\text{TYPES}) \to \Sigma^*$ ist definiert durch

$$name(s) := name(t), \text{ falls } s = t \in \text{SIMPLETYPES} \quad \text{(i)}$$
$$name(c_{\text{set}}(a, s)) := \texttt{\{}\ a : name(s)\ \texttt{\}} \quad \text{(ii)}$$
$$name(c_{\text{bag}}(a, s)) := \texttt{\{*}\ a : name(s)\ \texttt{*\}} \quad \text{(iii)}$$
$$name(c_{\text{list}}(a, s)) := \texttt{<}\ a : name(s)\ \texttt{>} \quad \text{(iv)}$$
$$name(c_{\text{tuple}}((a_1, s_1), \ldots, (a_k, s_k))) := \texttt{[}\ a_1 : name(s_1), \ldots, a_k : name(s_k)\ \texttt{]}, \quad \text{(v)}$$
mit $a_i \in \mathcal{A}$ paarweise verschieden für $0 \leq i \leq k$
$$name(c_{\text{array}}(a, s, (n_1, m_1), \ldots, (n_k, m_k))) := \texttt{<}\ a : name(s) | n_1 : m_1, \ldots, n_k : m_k\ \texttt{>}, \quad \text{(vi)}$$
mit $a \in \mathcal{A}$ und $n_i, m_i \in \mathbb{Z}$, $n_i \leq m_i$ für $1 \leq i \leq k$, $k \in \mathbb{N}_0$

Für $s \in \mathcal{S}(\text{TYPES})$ nennen wir $name(s)$ den zu s gehörenden **Typausdruck**. □

In [The96] sind im Wesentlichen die Objekttypen die Basis für die strukturelle Definition von Datenbankinstanzen, übernehmen also die Funktion der Schemata im relationalen und eNF^2-Datenmodell bzw. Klassen in objekt-orientierten Datenmodellen. Hier wird der Begriff Schema verwendet, um die Definition einer Struktur zu bezeichnen. Die Instanz eines komplexen Objektes wird auch als **Objektbaum** oder **Datenbaum** bezeichnet.

Ein Schema kann als Baum, dessen Knoten die Attribute des Schemas repräsentieren, aufgefaßt werden. Die durch Konstruktoren definierten Attribute werden dann durch die inneren Knoten des Schemabaumes repräsentiert, atomare Attribute durch dessen Blätter.

Für Schemata, bzw. Schemabäume, wird hier eine graphische Notation verwendet, die an die Darstellung des IFO-Datenmodells [AH87] angelehnt ist. Konstruktoren werden durch Kreise, atomare Attribute durch Quadrate als Knoten dargestellt, die jeweils durch weitere graphische Elemente unterschieden werden. Kollektionen und Felder werden durch das Symbol ⊛, Tupel durch das Symbol ⊗ dargestellt. Objekttypen werden symbolisch durch einen Schraubenschlüssel dargestellt („konstruierter Typ“), der in einem der Konstruktion des Objekttyps entsprechenden Typsymbol enthalten ist. Ein Objekttyp, der auf oberster Ebene auf einem Tupel-Konstruktor basiert, hat z. B. das Symbol ⊛.

Abb. 2.2 auf der gegenüberliegenden Seite zeigt alle Symbole, die in der graphischen Notation von Schemabäumen verwendet werden, in Abb. 2.3 auf Seite 46 und Abb. 2.4 auf Seite 47 werden Schemabäume mit Symbolen dargestellt. Zwei Besonderheiten sind die Symbole 🔢 und ⊡ für Bytestrings und Links.

Um weitere, insbesondere multimediale, Daten [MW91] repräsentieren zu können, wird zu den Basistypen der Bytestring ($\beta_{\text{binary}} \in \text{BASETYPES}$) hinzugenommen, der die Speicherung beliebiger Daten erlauben soll. Mit „beliebig“ ist gemeint, daß die Daten, wie bei multimedialen Daten üblich, sehr großvolumig sein können, und daß ihre innere Struktur,

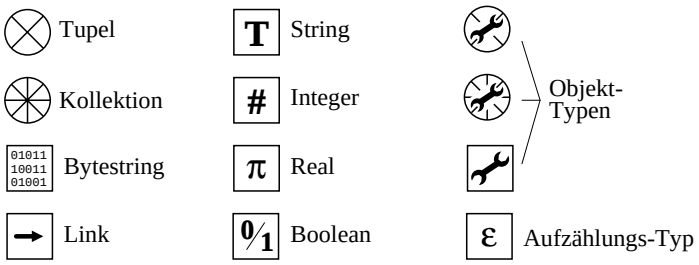


Abbildung 2.2: Symbole in Schemabäumen

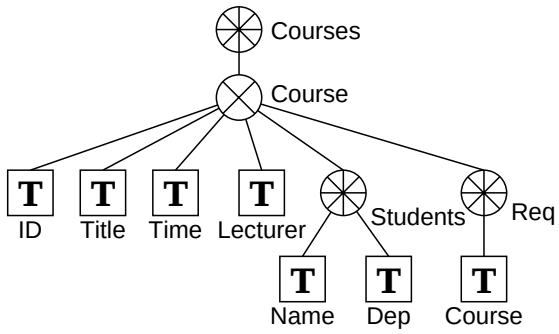
falls überhaupt eine vorhanden ist, keine Rolle spielt. Für diese Bytestrings hat sich die Bezeichnung BLOB, als Abkürzung für *binary large object*, eingebürgert (siehe z. B. [KB96])[12]. Hier wird jedoch der Begriff „Bytestring“ verwendet, weil einerseits im verwendeten Datenmodell auch Zeichenketten in ihrer Länge nicht beschränkt sind, andererseits aber auch „kurze“ Binärdaten gespeichert werden können. In Abschnitt 2.3.4 werden Bytestrings für die Definition eines Objekttyps Image verwendet.

Der in [The96, Definition 4.4, Seite 139 f.] als „intern“ vorgesehene Typ Link wird auf die externe Ebene gehoben. Dadurch wird die Basis für die Integration dieses in X-ESCHER bereits realisierten Konzeptes gelegt. Hier wird nicht auf Problematiken eingegangen, die mit solchen referenzierenden Datentypen verbunden sind, wie z. B. *dangling pointers*, sondern es werden nur Visualisierungsaspekte betrachtet.

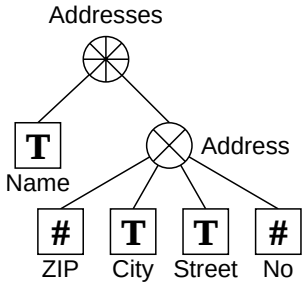
Als Besonderheit des in X-ESCHER implementierten Datenmodells sollen hier die sog. „Pseudo-Tupel“ erwähnt werden. Haben Kollektions-Knoten im Schemabaum mehr als einen Nachfolger, wird der Elementtyp dieser Kollektionen implizit als Tupel mit den Nachfolgerknoten als Komponenten angesehen. Um diese impliziten von den anderen, expliziten, Tupeln zu unterscheiden, werden jene „genuine Tupel“ genannt (siehe die folgenden Beispiele 2.1 und 2.2). Die Pseudo-Tupel sind insbesondere in X-ESCHERs Schemavisualisierung von Vorteil (siehe Abschnitt 3.1.2). Für den oben erwähnten Typ Link sind genuine Tupel als Elementtypen wichtig, da mit X-ESCHER Pseudo-Tupel für die Definition eines Link-Ziels im Schema nicht verwendet werden können. Pseudo-Tupel fallen aus den obigen Definitionen 3.9′ und 3.10′ heraus und werden hier in Typausdrücken informell durch Weglassen des Attributnamens vor den entsprechenden Tupeln ausgedrückt.

Schemata, ihre Darstellung als Typausdruck und Schemabaum, sowie der Einsatz von Objekttypen werden in Abb. 2.3 auf der nächsten Seite, Abb. 2.4 auf Seite 47 und Abb. 2.5 auf Seite 48 an Beispielen erläutert. Sie dienen der strukturellen Beschreibung komplexer Datenobjekte, welche die eigentlichen Gegenstände der Betrachtung sind. In ESCHER werden diese Datenobjekte immer Tabellen genannt, obwohl die Struktur der Schemata so allgemein ist, daß eine Tabelle aus einem atomaren Wert bestehen kann. Bei der Entwicklung des Visualisierungsverfahrens wird hier von „komplexen Datenobjekten“ gesprochen, aber in Verbindung mit ESCHER von „Tabellen“.

[12] Auch in SQL3 gibt es diesen Datentyp unter dem Namen BLOB.



(a) Schemabaum für Courses



(b) Schemabaum für Addresses

Abbildung 2.3: Beispiele für Schemabäume

Beispiel 2.1 Schemas Courses und Addresses

Die Abbildung zeigt zwei Schemabäume für Schemata namens Courses bzw. Addresses. Die entsprechenden Typausdrücke gemäß [The96] haben die folgende Form:

```
Courses:{ Course:[
      ID:string
      Title:string
      Time:string
      Lecturer:string
      Students:{ [
            Name:string
            Dep:string ] }
      Req:{ Course:string } ] }
```

```
Addresses:{ [
      Name:string
      Address:[
            ZIP:integer
            City:string
            Street:string
            No:integer ] ] }
```

Courses modelliert Vorlesungen als Menge von Tupeln, die Titel, Nummer, Termin, Dozent, Teilnehmer und Voraussetzungen einer Vorlesung als Komponenten enthalten. Die teilnehmenden Studenten werden durch Name und Fachbereich angegeben, die Voraussetzungen durch die Vorlesungsnummern. Das Tupel Course ist ein genuines Tupel, die Elemente der Menge Students sind Pseudo-Tupel.

Addresses modelliert eine Menge von Adressen, die jeweils einem Namen zugeordnet sind. Eine Adresse besteht aus Postleitzahl, Wohnort, Straße und Hausnummer. Die Elemente der Menge Addresses sind Pseudo-Tupel, Address ist ein genuines Tupel. □

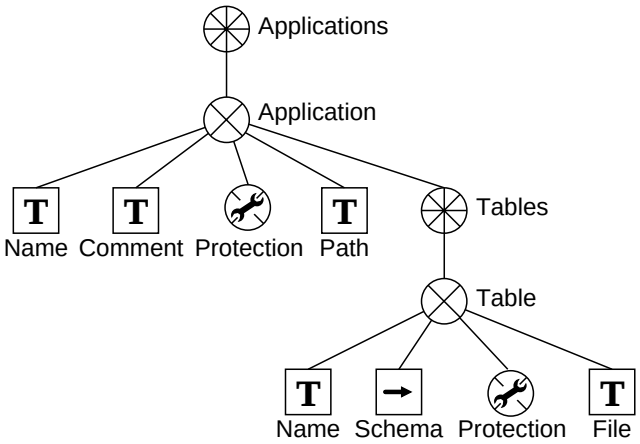


Abbildung 2.4: Schema der Meta-Tabelle Applications

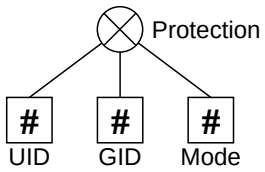
Beispiel 2.2 Schema Applications

Die Abbildung zeigt das Schema der Meta-Tabelle Applications. In dieser Tabelle werden Informationen über ESCHERs Anwendungen, und die darin enthaltenen Tabellen, gespeichert. Für eine Menge von Anwendungen werden jeweils deren Name, ein Kommentar, Zugriffsrechte und -pfad, sowie eine Menge von Tabellen gespeichert. Für jede Tabelle werden deren Name, ein Verweis auf ihr Schema, sowie Zugriffsrechte und -name[13] gespeichert. Ein entsprechender Typausdruck gemäß [The96] hat die folgende Form:

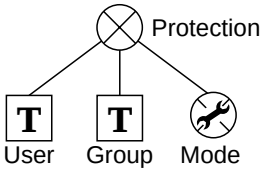
```
Applications:{ Application:[
      Name:string
      Comment:string
      Protection:Protection
      Path:string
      Tables:{ Table:[
            Name:string
            Schema:link
            Protection:Protection
            File:string ]}]}
```

Namen, Kommentar und Pfad sind atomare Attribute vom Typ String. Der Schema-Link verweist auf ein Table-Tupel, also in die Meta-Tabelle Applications selbst (in ESCHER werden auch Schemata als Tabellen gespeichert, siehe Abschnitt 2.3.3). Die Zugriffsrechte sollen an die Zugriffsrechte des UNIX-Dateisystems angelehnt sein. Sie sind hier als Objekttyp modelliert. Da die beiden Mengen Applications und Tables als Elementtypen ebenfalls genuine Tupel haben, enthält das Schema Applications keine Pseudo-Tupel. □

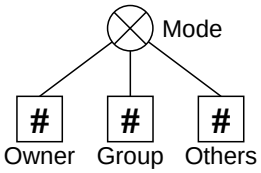
[13] Der Name einer Tabelle ist ihre Identifikation auf der externen Ebene, ihr Zugriffsname die Identifikation auf der internen Ebene. Letztere kann systemgeneriert sein und muß nicht, wie z. Zt. in X-ESCHER, ein Dateiname sein.



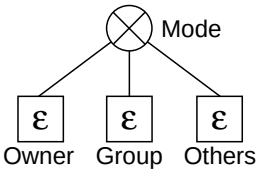
(a) Der Objekttyp Protection als Tupel mit Komponenten vom Typ Integer



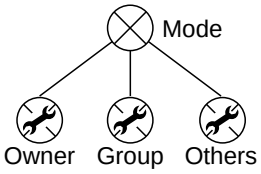
(b) Alternative Definition des Objekttyps Protection



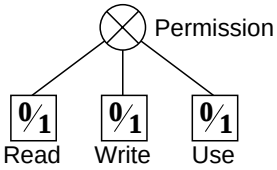
(c) Objekttyp Mode mit Komponenten vom Typ Integer



(d) Objekttyp Mode mit Komponenten vom Aufzählungstyp Permission



(e) Objekttyp Mode mit Komponenten vom Objekttyp Permission



(f) Objekttyp Permission

Abbildung 2.5: Definitionsalternativen des Objekttyps Protection

Beispiel 2.3 Objekttyp Protection

In der Abbildung sind verschiedene Alternativen für die Definition des Objekttyps Protection dargestellt. Die einfachste Struktur wäre die Modellierung durch die numerischen Benutzer- und Gruppen-Identifikation des Eigentümers einer Tabelle sowie des Zugriffsmodus als ein Bit-Muster (Abb. 2.5(a)). Alternativ könnten z. B. die Identifikationen textuell und der Modus wiederum als Objekttyp Mode modelliert werden (Abb. 2.5(b)). Letzterer trennt den Modus nach Rechten für den Eigentümer, die Gruppe und andere. Diese Rechte können dann als Bit-Muster (Abb. 2.5(c)), als Aufzählungstyp[14] (Abb. 2.5(d)) oder wiederum als Objekttyp definiert werden (Abb. 2.5(e)). Ein Objekttyp Permission würde die Aufteilung des Zugriffsmodus in seine einzelnen Booleschen Werte vervollständigen, indem für Lese-, Schreib- und Benutzungsrecht jeweils ein atomares, boolesches Attribut definiert wird (Abb. 2.5(f)). □

[14] Dieser Aufzählungstyp könnte die Zugriffsrechte z. B. durch die Label `read`, `write`, `use`, `read-write`, und `all` repräsentieren

Wie andere Datenmodelle integriert auch ESCHER$^+$ Nullwerte. Die üblichen Probleme, die beim Umgang mit Nullwerten entstehen, treten im operationalen Teil von Datenmodellen auf und sind daher in dieser Arbeit nicht von Bedeutung. In ESCHER gibt es keine nullwertigen Tupel, sondern höchstens Tupel, deren Komponenten alle nullwertig sind. Auch leere Tupel, also Tupel ohne Komponenten, sind per Definition ausgeschlossen. Kollektionen können nullwertig sein und haben dann eine unbekannte, d. h. nullwertige Kardinalität. Sie sind wohlunterschieden von leeren Kollektionen, deren Kardinalität 0 ist und semantisch eine andere Bedeutung haben. Nullwerte werden in dieser Arbeit durch das Symbol $\perp$, leere Kollektionen durch das Symbol ϵ dargestellt.

2.3.2 Das Fingerkonzept

Die Interaktion mit komplexen Datenobjekten basiert in ESCHER auf dem Konzept der **Finger** [Weg91a] und umfaßt die Navigation in und das Ändern von diesen Datenobjekten. Der Name beruht auf der Vorstellung, daß mit einem Finger auf ein Objekt gezeigt wird. Relativ zu seiner Position können mit einem Finger verschiedene Operationen ausgeführt werden. Diese **Fingeroperationen** verändern die Position des Fingers im Datenobjekt, bewegen ihn also, oder verändern das Datenobjekt selber, editieren es also.

Finger verhalten sich dabei ganz analog zu Cursorn in Text-Editoren: Sie können auf das nächste oder vorherige Datenobjekt, respektive das nächste oder vorherige Zeichen, bewegt werden; das Objekt, auf das gezeigt wird, respektive das Zeichen unter dem Cursor, kann gelöscht werden, usw.

Finger werden vom Object Manager umgesetzt, indem er die invarianten Identifikationen des Objektspeichers Record Manager ausnutzt (siehe Abschnitt 1.3)[15]. Jedes Sub-Objekt eines komplexen Datenobjektes ist im Objektbaum über einen eindeutigen Pfad, der von der Wurzel ausgeht, erreichbar. Der Object Manager repräsentiert einen Finger durch einen Stapel der Identifikationen, die zu den Sub-Objekten entlang des eindeutigen Pfades gehören [Weg91a, S. 341]. Finger können damit nicht auf Teilmengen einer Kollektion zeigen, da diese nicht durch eine Identifikation repräsentierbar sind. Dieses Konzept wird bei der Visualisierung von Fingern wichtig (siehe Abschnitt 4.3.3), da diese Stapel auch als Listen repräsentiert werden können. Mit diesen Listen können effizient Überschneidungen von Fingern und Datenobjekten festgestellt werden.

Finger können in zwei orthogonalen Richtungen bewegt werden:

1. Zwischen Objekt und Sub-Objekt bzw. umgebendem Objekt und
2. zwischen Objekten der gleichen Ebene.

[15] Alle Datenbanken versehen ihre Datenobjekte, zumindest intern, mit Identifikationen, um die Dateiorganisation durchzuführen (siehe z. B. [HR99, S. 145 ff.]).

Die grundlegenden Navigationsoperationen, die mit einem Finger ausgeführt werden können, sind *push*, *pop*, *next* und *back*. Die Benennung der Fingeroperationen *push* und *pop* ist durch die Vorstellung des Fingers als Stapel von Objekt- bzw. Sub-Objekt-Identifikationen motiviert. Die Bedeutung der Operationen ist wie folgt:

- *push* bewegt den Finger in ein komplexes Datenobjekt hinein, d. h. läßt ihn auf ein Sub-Objekt zeigen. Diese Operation ist gleichbedeutend mit dem Ablegen der Identifikation des Sub-Objektes auf dem Stack (*push on stack*). Für die *push*-Operation werden auch die Namen *in* und *enter* verwendet.
- *pop* bewegt den Finger aus einem Datenobjekt heraus, d. h. läßt ihn auf das umgebende Datenobjekt zeigen. Diese Operation ist gleichbedeutend mit dem Entfernen der obersten Identifikation vom Stack (*pop from stack*). Für die *pop*-Operation werden auch die Namen *out* und *escape* verwendet.
- *next* und *back* bewegen den Finger auf ein anderes Datenobjekt der gleichen Hierarchieebene, d. h. auf die nächste oder vorherige Komponente eines Tupels oder das nächste oder vorherige Element einer Kollektion[16]. Diese Operationen sind gleichbedeutend mit dem Ersetzen der obersten Identifikation des Stacks durch eine andere Identifikation. Für die *next*- und *back*-Operation werden auch die Namen *successor* und *predecessor* bzw. deren Abkürzungen *succ* und *pred* verwendet. Zusammenfassend wird auch von *move*-Operationen gesprochen.

Die Finger entsprechen damit nicht nur dem Cursor eines Text-Editors, indem sie eine Position markieren. Sie definieren gleichzeitig auch einen Bereich, nämlich ein Sub-Objekt eines komplexen Datenobjektes, und entsprechen damit auch dem Konzept der Blöcke. Die grundlegenden Änderungsoperationen, die mit einem Finger ausgeführt werden können, sind

- *insert*, *insert_before*, *delete*, *delete_before* (oder *backspace*), wenn der Finger auf ein Element einer Kollektion zeigt,
- *to_null* und *to_singleton*, wenn der Finger auf eine leere Kollektion zeigt, und
- *to_empty*, wenn der Finger auf eine nullwertige Kollektion zeigt.

Außerdem können atomare Werte gelesen und geschrieben werden. Wird ein Element in eine Kollektion eingefügt, ist es zunächst nullwertig (atomare Elemente und Kollektionen als Elemente) oder seine Komponenten sind alle nullwertig (tupelwertige Elemente)[17]. Der Finger zeigt nach dem Einfügen auf das neu eingefügte Element (für *to_singleton* wird implizit eine *push*-Operation ausgeführt). Nach dem Löschen zeigt der Finger auf das

[16] In ESCHER liegt immer eine, wenn auch implementationsabhängige, Ordnung auf Kollektionen vor.
[17] Tupel können nicht nullwertig sein.

nachfolgende Element, falls dieses existiert, ansonsten auf das vorhergehende; wird das letzte Element einer Kollektion gelöscht, zeigt der Finger nachher auf die leere Kollektion (es wird implizit eine *pop*-Operation ausgeführt).

Über die Funktionalität der meisten Text-Editoren hinaus bietet ESCHER die Möglichkeit, mit mehreren Fingern gleichzeitig in einem komplexen Datenobjekt zu navigieren. Die Positionen zweier Finger F_1 und F_2 können in drei verschiedenen Verhältnissen zueinander stehen:

1. $F_1 \cong F_2$ — Die Finger F_1 und F_2 zeigen auf dasselbe Datenobjekt (sie sind kongruent).
2. $F_1 \Subset F_2$ — Finger F_1 zeigt auf ein Sub-Objekt des Datenobjektes, auf das Finger F_2 zeigt.
3. $F_1 \Cap F_2$ — Die Finger überschneiden sich gegenseitig in keiner Weise.

Insbesondere können sich zwei Finger aufgrund des hierarchisch strukturierten Datenmodells nicht teilweise überschneiden. Die verwendeten Sprechweisen sind durch die visuelle Darstellung von Fingern motiviert (siehe Abschnitt 3.1.3). Wenn der Dereferenzierungsoperator $*$ auf einen Finger angewendet die Menge aller Objekte und Sub-Objekte, auf die der Finger zeigt, liefert, sind die folgenden Aussagen jeweils gleichwertig zu den obigen:

1. $*F_1 = *F_2$ — F_1 und F_2 sind identisch.
2. $*F_1 \subsetneq *F_2$ — F_1 ist in F_2 enthalten.
3. $*F_1 \cap *F_2 = \varnothing$ — F_1 und F_2 sind disjunkt.

Ein Finger wird dann quasi als Zeiger auf ein komplexes Datenobjekt im Sinne einer Programmiersprache betrachtet.

Finger werden nicht nur zur Interaktion, sondern auch intern für eine Fülle von Aufgaben verwendet, z. B. für die rekursive Traversierung der komplex strukturierten Datenobjekte, für das Sortieren und für die Generierung von Visualisierungen. Sie können über die graphische Benutzerschnittstelle (siehe Abschnitt 3.1.3), durch die Programmierschnittstelle des Object Managers, oder mit Tcl/DB (siehe Abschnitt 2.3.4) benutzt werden.

2.3.3 Meta-Daten

Relationale Datenbanksysteme speichern die Struktur ihrer Tabellen und weitere Systeminformationen in der Regel in Tabellen, die dann das sog. „Datenverzeichnis“ (*data dictionary*) bilden. Diese Tabellen enthalten für z. B. jedes Attribut einen Eintrag (d. h. ein Tupel), der den Attributnamen, -typ, usw. beschreibt.

In ESCHER bildet jede Strukturdefinition eine Tabelle, die als Schema interpretiert wird. Eine Besonderheit dieser Tabellen ist, daß sie entsprechend dem eNF^2-Datenmodell rekursiv definiert sind: Attribute sind durch Tupel repräsentiert, die als eine Komponente eine Liste von Sub-Attributen haben, deren Elemente wie das Attribut selbst strukturiert sind. Atomare Attribute sind durch leere Sub-Attribut-Listen gekennzeichnet und definieren so das Rekursionsende.

Die hier wichtigen Komponenten eines Attribut-Tupels sind Name, der Name des Attributes, Type, der Typ des Attributes, und Sub-Attributes, die Liste der Sub-Attribute[18] (siehe Abb. 2.6 auf der gegenüberliegenden Seite). In der Meta-Tabelle Applications, deren Schema in Beispiel 2.2 auf Seite 47 vorgestellt wurde, werden weitere Systeminformationen von ESCHER gespeichert.

Auch die Schemata repräsentierenden Tabellen haben ein Schema, das ihre Struktur beschreibt. Dieses Schema beschreibt die Struktur aller Schemata, also auch ihre eigene. Hier wurde das in Abschnitt 1.3 erwähnte Konzept der Selbstreferenzierung exemplarisch umgesetzt. Da dieses Schema quasi der Ursprung aller Daten ist, das als erstes für die Interpretation anderer Daten gelesen werden muß, heißt es **Boot-Schema**[19].

Das Boot-Schema ist in Abb. 2.6 auf der gegenüberliegenden Seite zusammen mit zwei Definitionsalternativen dargestellt, wobei in Abb. 2.6(a) die derzeit in X-ESCHER realisierte Variante zeigt. Durch einen rekursiv definierten Objekttyp Attribute könnte das Boot-Schema wie in Abb. 2.6(b) gezeigt definiert werden. In Abb. 2.6(c) ist eine Definitionsalternative dargestellt, die auf die Sub-Attribute eines komplexen Attributs durch Links verweist. Hierbei muß auf äußerster Ebene dann ein Mengenkonstruktor eingefügt werden.

Ein Schema kann also als Tabelle, deren Struktur durch das Boot-Schema definiert ist, angesehen werden; aus dieser Sicht ist es „nur“ eine speziell interpretierte Tabelle.

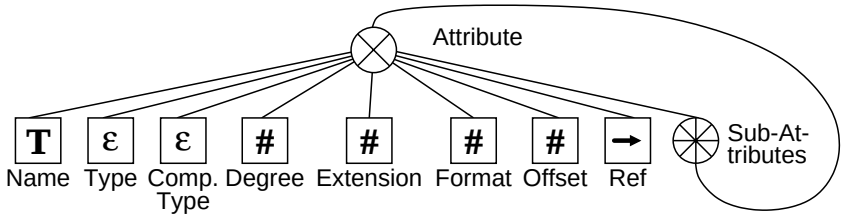
Die in Abschnitt 2.3.2 besprochenen Finger treten intern immer paarweise auf: Jeder Finger in einer Tabelle hat einen assoziierten Finger in der Tabelle, die das Schema repräsentiert. Die beiden Finger werden synchron bewegt, so daß für den Tabellenfinger immer die aktuellen Strukturinformationen verfügbar sind.

2.3.4 Erweiterungen

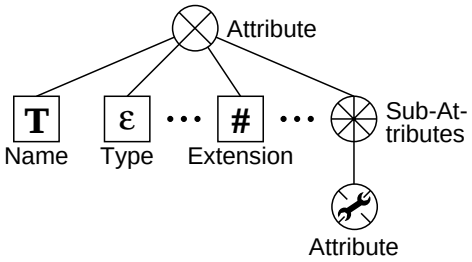
Die Arbeiten, die sich mit einer formalen Beschreibung des ESCHER zugrunde liegenden Datenmodells und dessen Erweiterung beschäftigen [The96, Pau94], wurden bereits erwähnt. Nur sehr wenige der Ergebnisse beider Arbeiten sowohl hinsichtlich des strukturellen als

[18] Exakt heißen die Attribute in X-ESCHER ANAME, BT und SUBATTR, siehe Abb. 3.7(b) auf Seite 73. Dort sind Schemata als Tabellen dargestellt, in denen der Attributtyp aber als Integer codiert ist.

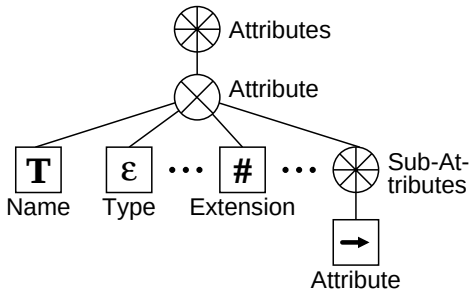
[19] Der Anglizismus *booten* (von *to bootstrap* und *pull o. s. up by one's own bootstraps*, dt. sich aus eigener Kraft hocharbeiten) hat sich im Deutschen für den Startvorgang eines Computers eingebürgert. Boot-Schema könnte im Deutschen mit „Urladeschema“ übersetzt werden, hier soll jedoch der englische Begriff verwendet werden.



(a) Boot-Schema



(b) Objekttyp Attribute rekursiv



(c) Objekttyp Attribute mit Link

Abbildung 2.6: Boot-Schema mit Definitionsalternativen

auch hinsichtlich des operationalen Teils des Datenmodells haben jedoch bisher Eingang in den implementierten Protyp X-ESCHER gefunden. Ein Grund dafür ist sicherlich, daß durch die System-Architektur eine relativ starre Implementierung entstanden ist. Ein Ziel dieser Arbeit ist, durch die Grundlagen für ein flexibles und modulares Visualisierungskonzept die Realisierung der oben genannten Erweiterungen zu ermöglichen und so deren Validierung und Weiterentwicklung voranzutreiben.

Tcl/DB

Ein erster Ansatz zur Integration operationaler Aspekte in ESCHER wurde durch die Verbindung mit der Skriptsprache Tcl [Ous94] unter dem Namen Tcl/DB realisiert [Ahm98]. Dieser weicht zwar von den Sprachvorschlägen aus [The96, Pau94] ab, die eher Entwicklungen im Sinne einer objekt-orientierten Datenbanksprache à la OQL sind. Tcl/DB stellt aber eine pragmatische Alternative dar, mit der die Fingeroperationen programmiersprachlich formuliert werden können[20]. Ihre Berechtigung wurde bereits in weiteren Forschungsaktivitäten belegt [WWT97, TW98, Tha98].

[20] Tcl/DB wurde auch schon als „eine Art NF²-Assembler“ bezeichnet.

Der Name der Skriptsprache (*scripting language*) Tcl steht für *Tool command language*. Tcl wurde von Ousterhout 1988 als erweiterbare, wiederverwendbare Kommandosprache konzipiert und entwickelt.

Die erste Erweiterung für Tcl wurde mit Tk von Ousterhout selbst realisiert. Tk ist ein auf Tcl basierendes Toolkit zur Programmierung graphischer Benutzerschnittstellen, das anfangs für X Windows konzipiert war. Inzwischen ist es aber auch in Microsoft Windows und Apple Macintosh Umgebungen verwendbar. Tcl und seine Erweiterung Tk werden meist zusammen unter dem Namen Tcl/Tk verwendet und haben inzwischen weite Anerkennung und Verbreitung gefunden. Auch heute noch betreibt Ousterhout die Weiterentwicklung und verbreitete Anwendung von Tcl/Tk, z. B. als aktive Komponente von XML [Ous99].

Skriptsprachen zeichnen sich durch Kommandointerpretation und ein schwach ausgeprägtes Typkonzept aus. Früher war auch Tcl eine rein interpretierende Sprache, seit Version 8.0 integriert sie aber einen *on-the-fly bytecode compiler*, um die Ausführungsgeschwindigkeit der Skripten zu erhöhen. Das Typkonzept von Tcl umfaßt prinzipiell nur Strings, Listen und assoziative Arrays. Erstere werden von einigen Kommandos speziell interpretiert, z. B. als Zahlen oder Tcl-Skripten. Listenelemente können selbst wieder Listen sein, womit in Tcl beliebige geschachtelte Strukturen darstellbar sind.

Tcl bietet die üblichen Konstrukte einer Programmiersprache wie Variablen, Schleifen, bedingte Anweisungen, Unterprogramme, Rekursion, usw. Durch ihre Art der Behandlung von Strings als Tcl-Skripten erlaubt sie aber selbst die Definition neuer Kontrollstrukturen, die wie auch Unterprogramme als *„first class“* Kommandos, d. h. wie die bereits vorhandenen *built-in* Kommandos, behandelt werden.

Die Erweiterbarkeit von Tcl manifestiert sich besonders in seiner Programmierschnittstelle („One of the reasons for the popularity of Tcl is the ability to extend the language by writing C code that implements new commands.“ [Ous98, S. 28]). Mit dieser können neue Tcl-Kommandos in der Programmiersprache C implementiert und somit Verbindungen zu allen Anwendungen, die ebenfalls eine auf C basierende Programmierschnittstelle haben, hergestellt werden.

Grundlegende Idee von Tcl/DB ist die Integration von Datenbankoperationen in Tcl, d. h. die Erweiterung um ESCHER-spezifische Kommandos anhand der Schnittstelle des Object Managers. Dazu werden in Tcl/DB drei neue Kommandos statisch definiert, die dem Systemmanagement (z. B. Hochfahren der Datenbank, Verwaltung von Anwendungen), dem Umgang mit Tabellen und Schemata (z. B. Anlegen neuer oder Öffnen existierender Tabellen), sowie der Verwaltung von Fingerobjekten (Generieren und Freigeben) dienen. Eine Übersicht dieser drei Kommandos sowie ihrer Optionen und Parameter ist in der folgenden Tabelle dargestellt (die Syntax von Tcl/DB orientiert sich in der Tradition von Tcl/Tk an den Shell-Sprachen von UNIX).

Kommando		Aufgabenbereich
Option	Parameter	Bemerkung
`escher`		System/Datenbank
`boot`	[*path*]	Datenbank hochfahren
`shutdown`		Datenbank herunterfahren
`list`	`applications`	Namen aller Anwendungen
	`schemas` | `tables`	Namen aller Schemata/Tabellen in der aktiven Anwendung
`application`	[`show`]	Name der aktiven Anwendung
	`select` *name*	Anwendung *name* auswählen
	`new` | `delete` *name*	Anwendung *name* anlegen/löschen
`import`	*file*	Tabelle/Schema importieren
`null`	[*value*]	Nullwert generieren oder testen
`table`		Tabellen
`open`	*name* [`-id` *tid*]	Tabelle *name* öffnen (mit TID *tid*)
`new`	*name scm*	Tabelle *name* zu Schema *scm* anlegen
`close` | `delete`	*tid*	Tabelle *tid* schließen/löschen
`name` | `schema`	*tid*	Namen zu TID *tid*
`finger`		Finger
`fork`	*fid* [`-id` *new_fid*]	Finger (mit FID *new_fid*) anlegen
`free`	*fid*	Finger *fid* löschen
`list`	*fid* [`-table` *tid*]	FIDs aller Finger (in der Tabelle *tid*)

Tabelle 2.1: System-, Tabellen- und Fingerkommandos von Tcl/DB

Jedes neue Fingerobjekt ist mit einer Fingeridentität (FID) assoziiert; mittels dieser kann über ein dynamisch generiertes Fingerobjektkommando mit dem Finger kommuniziert werden. Die Fingerobjektkommandos werden hier auch kurz *fid*-Kommandos genannt. Ihre Optionen und Parameter sind in der folgenden Tabelle dargestellt.

Option	Aufgabenbereich
Parameter	Bemerkung
`push`	*in*-Operation
[`-first`]	erstes Sub-Objekt (Default)
`-last`	letztes Sub-Objekt
`-name` *attribute*	Komponente namens *attribute*
`-index` *i*	*i*-tes Sub-Objekt
[`-path`] *path*	gemäß *path*
`-rid` *rid*	Sub-Objekt mit Identifikation *rid*
`pop`	*out*-Operation (keine Parameter)

Fortsetzung auf der nächsten Seite

Option	Aufgabenbereich
Parameter	Bemerkung
`go`	*move*-Operationen
`-first` | `-last`	erstes/letztes Objekt
`-back` | `-next`	vorheriges/nachfolgendes Objekt
`-name` *attribute*	Tupelkomponente namens *attribute*
`-index` *i*	*i*-tes Objekt
`get`	Retrieve (atomar)
[*path*]	
`set`	Update
[`-value`] *value* [*path*]	Wert (atomar)
`-channel` *cid* [*path*]	Bytestring (atomar)
`-tonull` [*path*]	zu Null machen (atomar/Kollektionen)
`-toempty` [*path*]	nullwertig → leer (Kollektionen)
`-tosingleton` [*path*]	leer → einelementig (Kollektionen)
`delete`	Löschen (Element)
[*path*]	
`insert`	Einfügen (Element)
[`-after`] [*path*]	nach *fid* (Default)
`-before` [*path*]	vor *fid*
`info`	Information über *fid*
[`-all`] [*path*]	alle verfügbaren Informationen (Default)
`-card` [*path*]	Kardinalität (komplex)
`-first` | `-last` [*path*]	auf erstem/letztem Sub-Objekt?
`-index` [*path*]	Index als Sub-Objekt
`-type` [*path*]	Typbezeichner
`-name` [*path*]	Attributname
`is`	Prädikate
`-null` [*path*]	nullwertig?
`-empty` [*path*]	leer? (Kollektionen)
`-atomic` [*path*]	atomarer Typ?
`-complex` [*path*]	komplexer Typ?
`-collection` [*path*]	Kollektions-Typ?
`-tuple` [*path*]	Tupel-Typ?
`link`	Operationen mit Links
[`-link`] [*path*]	Fingerposition als Link-Ziel (Default)
`-atomic` [*path*]	Wert (atomares Link-Ziel)
`-newfinger` [`-fid` *new_fid*] [*path*]	neuen Finger auf Link-Ziel
`-followfinger` *other_fid* [*path*]	Finger *other_fid* auf Link-Ziel positionieren

Tabelle 2.2: Das Fingerobjektkommando von Tcl/DB

Nullwerte werden in Tcl/DB durch das Literal „\?“ dargestellt, leere Kollektionen wie leere Tcl-Listen durch „{}“. Detailliertere Informationen über die Kommandos von Tcl/DB sind in [Ahm98] und in den Dokumentationen der aktuellen Implementierungen des ESCHER-Projektes zu finden.

Ein Tcl/DB-Skript ist im Allgemeinen so aufgebaut, daß zunächst eine Tabelle geöffnet wird (`table open`); das Resultat ist ein String, der die Tabellenidentifikation (TID) darstellt. Die TID kann mit dem Suffix „`.root`“ versehen werden und dient dann als erster (Wurzel-) Finger in der Tabelle dem Generieren neuer (Arbeits-) Finger mit dem Kommando `finger`[21]. Resultat ist wiederum ein String, der jetzt die FID des generierten Fingerobjektes darstellt. Mit den Arbeitsfingern kann dann in den Datenobjekten navigiert und operiert werden. Abschließend werden generierte Finger wieder freigegeben und geöffnete Tabellen wieder geschlossen.

```
 1  set t [table open courses.tbl]
 2  set f [finger fork $t.root]
 3  $f push {?ID=="CS409"?.Students}
 4  if [$f push -first] {
 5    while 1 {
 6      set dep [$f get Dep]
 7      if [info exists deps($dep)] {
 8        incr deps($dep)
 9      } else {
10        set deps($dep) 1
11      }
12      if [$f info -last] break
13      $f go -next
14    }
15  }
16  finger free $f
17  table close $t
```

Beispiel 2.4 Tcl/DB-Skript Hörerstatistik

Das Skript erstellt für die Vorlesung mit der Nummer „CS409“ eine Statistik über die Anzahl der Hörer pro Fachbereich. Das `push`-Kommando in Zeile 3 hat als Argument einen Pfad, der eine deklarative Spezifikation enthält. □

Skript 2.1: Tcl/DB-Skript Hörerstatistik

[21] In Anlehnung an die Prozeßgenerierung in UNIX-Systemen heißt die Option des Kommandos `finger` zum Generieren eines Fingerobjektes `fork`.

Die Skripten der Beispiele 2.4 und 2.5 beziehen sich auf das Vorlesungsschema aus Beispiel 2.1 auf Seite 46 (siehe auch Abb. 3.1 auf Seite 63 zu Beispiel 3.1) und berechnen eine Hörerstatistik (Skript 2.1 auf der vorherigen Seite) bzw. Vorlesungsvoraussetzungen (Skript 2.2).

```
 1  proc requirements {t id {req {}}} {
 2    set f [finger fork $t.root]
 3    set res {}
 4    $f push "?ID==\"$id\"?.Req"
 5    foreach reqby [val_get $f] {
 6      if {[lsearch -exact [concat $req $res] $reqby] < 0} {
 7        lappend req $reqby
 8        set res [concat $res [requirements $t $reqby $req]]
 9      }
10    }
11    finger free $f
12    return $res
13  }
14
15  set t [table open courses.tbl]
16  set reqs [requirements $t CS409]
17  table close $t
```

Beispiel 2.5 Tcl/DB-Skript Voraussetzungen

Das Skript definiert eine rekursive Funktion `requires`, die eine Liste aller für eine Vorlesung vorausgesetzten Vorlesungen berechnet und ruft diese für die Vorlesung mit der Nummer „CS411“ auf. Das Kommando `val_get`, das in Zeile 5 aufgerufen wird, liefert einen komplex strukturierten Wert (in diesem Fall die Menge der vorausgesetzten Vorlesungen) als Tcl-Liste. □

Skript 2.2: Tcl/DB-Skript Voraussetzungen

Die Kombination mit anderen Erweiterungen, insbesondere mit Tk, hat sich als geeignete Basis für weitere Forschungsaktivitäten erwiesen. Eine der ersten Anwendungen von Tcl/DB für ESCHER ist ein Web-Interface [TWW99], das die Ausführung von Tcl/Tk-Skripten durch ein Tcl-Plug-in als sog. „Tclets“ erlaubt. Auch eine Re-Implementierung der graphischen Benutzerschnittstelle von X-ESCHER auf der Basis von Tcl/DB, d.h. insbesondere ohne OSF Motif, ist bereits in der Testphase.

Durch die Verwendung von Tcl/DB können für ESCHER auch Methoden, der Datentyp Skript und berechnete Werte realisiert werden. Insbesondere der zweite Punkt, die Speicherung von ausführbaren Anweisungen durch einen Datentyp Skript, ist interessant. in

Verbindung mit Tk können dadurch sich selbst modifizierende Benutzerschnittstellen realisiert werden.

Multimedia

Eine andere Erweiterung von X-ESCHER ist die Integration von multimedialen Datentypen, insbesondere Rasterbildern [Lie95]. Diese wurde zeitlich vor der Entwicklung von Tcl/DB realisiert, so daß dazu in die internen Schichten von X-ESCHER eingegriffen werden mußte. Es hat sich gezeigt, daß dazu auf fast allen Ebenen weitreichende Änderungen nötig waren; nur die unterste Schicht, der Objektspeicher, konnte unverändert bleiben.

Intern und extern werden die Bilder als atomare Typen behandelt: Sie werden als unstrukturierte Byte-Folge im Objektspeicher abgelegt und ähnlich wie die anderen atomaren Typen an der graphischen Benutzerschnittstelle repräsentiert. Ähnlich heißt, daß Darstellungsbreite und -höhe der Bilder festgelegt sind und die Visualisierung durch eine entsprechende Skalierung erzeugt wird. Programmtechnisch sind Bilder aber als strukturierte Objekte implementiert, die z. B. Angaben über den Namen der Quelldatei und die Größe des gespeicherten Bildes enthalten.

Mit der Erweiterung des Datenmodells aus [The96] und Bytestrings hätten Bilder durch einen Objekttyp Image realisiert werden können, der in Abb. 2.7 dargestellt ist. Dadurch wären zumindest die Änderungen der internen Ebenen vermieden worden[22].

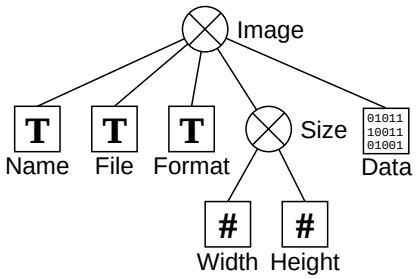


Abbildung 2.7: Definition des Objekttyps Image

[22] Auch die Integration der Bytestrings erfordert Änderungen auf der internen Ebene. Die Bytestrings können dann aber für alle unstrukturierten Binärdaten verwendet werden. Im Gegensatz dazu wurden im Rahmen von [Lie95] z. B. für jedes Grafikformat (GIF, Bitmap, usw.) eigene Typen implementiert.

Kapitel 3

Visualisierungen — statischer Teil

Der Begriff Visualisierung wird im Folgenden synonym sowohl für

- das Ergebnis der visuellen Repräsentation (Darstellung) eines Datenobjektes als auch
- den Vorgang der visuellen Repräsentation (Generierung) und
- die Festlegungen (Algorithmen und Datenstrukturen), die den Vorgang der visuellen Repräsentation eins Datenobjektes steuern, verwendet.

Eine Visualisierung im ersten Sinn ist das Bild, das der Benutzer des Editors am Bildschirm sieht. Sie wird hier mit dem Begriff „Repräsentation“ bezeichnet. Ihre Generierung wird durch eine Visualisierung im zweiten und dritten Sinn definiert. Abschnitt 3.1 befaßt sich mit der von X-ESCHER realisierten Repräsentation von Datenobjekten und der Interaktion mit ihnen. Der Begriff der Visualisierung als Festlegung, wie ein Datenobjekt dargestellt wird, ist Gegenstand der Betrachtungen in den darauf folgenden Abschnitten.

3.1 Das Visualisierungskonzept von X-ESCHER

Geschachtelte Relationen werden von X-ESCHER durch eine graphische Benutzerschnittstelle visualisiert. Die Relationen werden in einer tabellarischen Form dargestellt, wobei die geschachtelt relationale Struktur des eNF^2-Datenmodells in einer Schachtelung von Tabellen in der Darstellung ihre Entsprechung findet. Zusätzlich werden von X-ESCHER Strukturinformationen visualisiert. Auch das Konzept der Finger wird durch ihre visuelle Repräsentation in die Benutzerschnittstelle integriert.

3.1.1 Repräsentation von Tabellen

Die tabellarische Darstellung ist eine Anordnung von graphischen Objekten. Diese Objekte sind rechteckige Bereiche, die jeweils einen einfachen oder strukturierten Wert visualisieren;

sie Übernehmen die Funktion der elementaren Dialogbausteine bzw. Behälter-Dialogbausteine, wie sie in [Kli96, S. 112 f.] genannt werden.

Bei der Untersuchung des in X-ESCHER realisierten Visualisierungskonzeptes kann dieses anhand einer Klassifizierung der Typen (bzw. Typkonstruktoren) in drei Fälle zerlegt werden:

1. Atome sind „einfache“, nicht weiter strukturierte Objekte. Sie werden in einem rechteckigen Bereich dargestellt, dessen Breite fest und durch das Schema bestimmt ist.

2. Kollektionen sind Sammlungen von gleich strukturierten (isomorphen) Objekten (Elementen), deren Anzahl variabel und veränderbar ist. Sie treten in der Form von Mengen, Listen und Multimengen auf. Die Reihenfolge der Elemente ist nur für Listen wichtig, aber prinzipiell für alle Kollektionen variabel und veränderbar. Kollektionen werden durch **vertikale Konkatenation** dargestellt, d. h. die Elemente, bzw. die Repräsentationen der Elemente, die gleich breit sind, werden übereinander, evtl.[1] getrennt durch horizontale Linien, dargestellt. Eine Veränderung der Elementanzahl wirkt sich in einer Veränderung der Visualisierungshöhe der Kollektion aus.

3. Tupel sind Sammlungen von beliebig strukturierten Objekten (Komponenten), deren Anzahl und Reihenfolge fest und durch das Schema bestimmt sind. Sie werden durch **horizontale Konkatenation** dargestellt, d. h. die Komponenten bzw. die Repräsentationen der Komponenten werden nebeneinander, getrennt durch vertikale Linien, dargestellt. Die Komponenten können unterschiedliche Visualisierungshöhen haben; die Visualisierungshöhe eines Tupels bestimmt sich aus der maximalen Höhe einer Komponente. Atomare Komponenten werden in der Höhe angepaßt.

Diese Form der Anordnung graphischer Objekte zu einer tabellarischen Darstellung geschachtelter Relationen wird im ESCHER-Projekt auch als „Kasseler Normalform“ bezeichnet. Sie ist im Vergleich zu graphen- und formularbasierten sowie anderen tabellarischen Repräsentationen durch ihre Schnörkellosigkeit klar gegliedert und intuitiv erfaßbar (siehe auch die Bemerkung zu arbiträren Vereinbarungen in Abschnitt 1.1 auf Seite 10). Als Beispiel ist in Abbildung 3.1 auf der gegenüberliegenden Seite die Repräsentation einer Vorlesungstabelle dargestellt.

Doppellinien, die durch eine Ineinanderschachtelung von Tabellen ohne Überdeckung der begrenzenden Linien entstehen, können die Darstellung geschachtelter Relationen besonders entstellen (siehe z. B. [PVG92]). Das gilt auch für Wannenförmige Darstellungen (siehe z. B. [MNE96b]), die höchstens für kleine Tabellen geeignet sind.

Aus den drei Fällen und dem rekursiven Aufbau des Visualisierungskonzeptes kann abgeleitet werden, daß die Visualisierungsbreiten aller Komponenten fest sind. Die Visualisierungshöhen von Kollektionen hängen von ihren Kardinalitäten und den Visualisierungshöhen

[1] Die Elemente werden nicht durch horizontale Linien getrennt, wenn sie atomar oder Tupel mit ausschließlich atomaren Komponenten sind. Ihre Struktur ist dann so einfach, daß sie „in einer Zeile nebeneinander“ dargestellt werden.

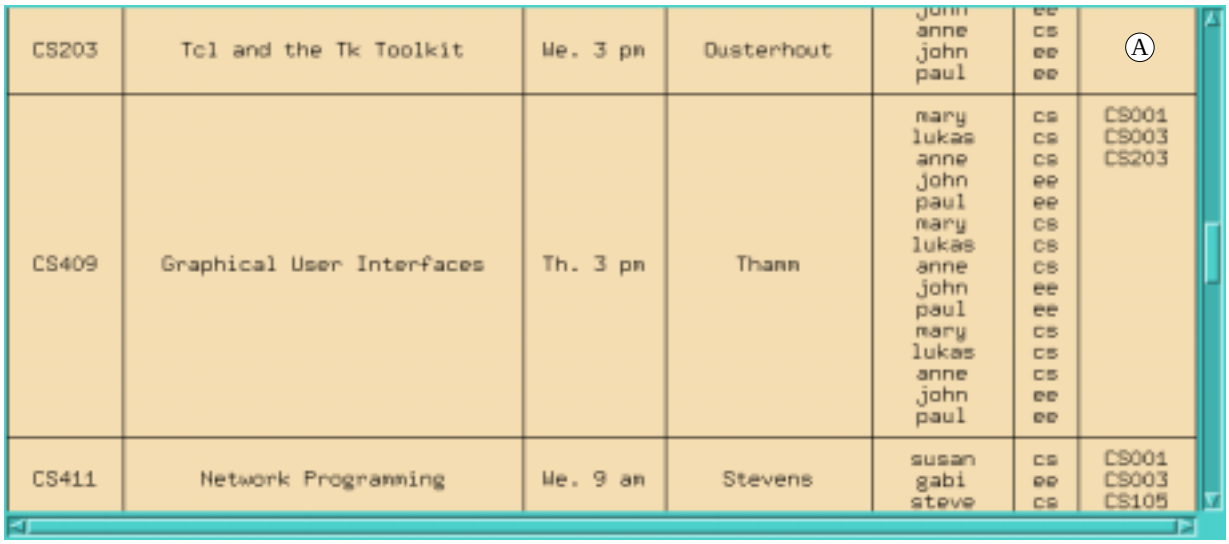


Abbildung 3.1: X-ESCHER-Repräsentation der Vorlesungstabelle

Beispiel 3.1 X-ESCHER-Repräsentation der Vorlesungstabelle

Die Abbildung zeigt die X-ESCHER-Repräsentation einer Vorlesungstabelle, die durch das Schema Courses (siehe Beispiel 2.1 auf Seite 46) strukturiert ist. Drei Vorlesungs-Tupel (IDs „CS203“, „CS409“ und „CS411“) sind als Mengenelemente vertikal konkateniert. Sie haben (hinreichend) komplexe Sub-Attribute, so daß sie durch Linien getrennt sind. Die Komponenten der einzelnen Tupel sind vertikal konkateniert und durch Linien getrennt.

Die Course-Tupel haben nur sechs Komponenten, obwohl in Abb. 3.1 sieben Spalten zu sehen sind. Die Erklärung sind die einzeiligen Tupel der Students-Menge mit jeweils zwei Komponenten. In Abschnitt 3.1.2 wird gezeigt, wie dieser Fehlinterpretation durch Darstellung zusätzlicher Strukturinformation vorgebeugt werden kann.

Auffallend ist, daß die Werte der atomaren Attribute der oben und unten nur teilweise sichtbaren Vorlesungen im sichtbaren Bereich der jeweiligen Tupel vertikal zentriert dargestellt sind (ID: „CS203“ und „CS411“, Title: „Tcl ... “ und „Network ... “, usw.). Wären die Werte im ganzen von den Tupeln beanspruchten Bereich zentriert, würden sie in Abb. 3.1 nicht sichtbar sein, da sie oberhalb bzw. unterhalb des sichtbaren Ausschnitts lägen.

Ferner ist anzumerken, daß in der letzten Spalte die drei Werte „CS001“, „CS003“ und „CS203“ als Elemente der Menge „Req“ am oberen Rand dargestellt werden. Deshalb darf man aus der leeren Fläche Ⓐ nicht schließen, daß für die Vorlesung „CS203“ keine anderen Vorlesungen vorausgesetzt werden. □

ihrer Elemente ab, die Visualisierungshöhen von Tupeln hängen von den Visualisierungshöhen ihrer Komponenten ab.

In X-ESCHER ist die Breite der Attribute in der Schemadefinition festgelegt. Diese Visualisierungsinformationen werden zusammen mit der Strukturinformation als Meta-Daten gespeichert. Die Breite muß explizit nur für atomare Attribute angegeben werden, für komplexe Attribute kann sie aus deren struktureller Definition abgeleitet werden. Diese Vorgehensweise orientiert sich an den Vorgaben für Tabellen in der Textverarbeitung (*desk top publishing*), bei der Spaltenbreiten vom Anwender eingesetzt werden.

Die explizite Speicherung der Attributbreiten ist auch aus Performance-Gründen wichtig, da bei deren Berechnung aus den Visualisierungsbreiten der Daten insbesondere für über mehrere Ebenen geschachtelte Kollektionen die Breitenberechnung nicht mehr durch lokale Breitenvergleiche vorgenommen werden kann. Diese Problematik wird in Abb. 3.2 auf der gegenüberliegenden Seite näher erläutert.

Aufgrund der festen, durch das Schema bestimmten Breiten, wird die Zusammengehörigkeit von Sub-Objekten eines komplexen Datenobjektes durch **visuelle Korrelation** verdeutlicht, indem die zugehörigen graphischen Objekte in der Repräsentation ähnlich angeordnet werden:

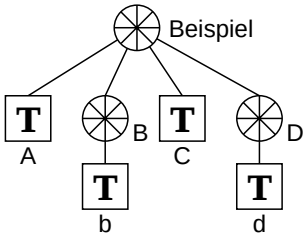
- Alle Elemente einer Menge belegen die gleiche horizontale Position.
- Alle Komponenten eines Tupels belegen die gleiche vertikale Position.
- Aus den ersten beiden Punkten ergibt sich, daß alle Objekte, die durch den gleichen Knoten im Schemabaum strukturiert werden, die gleiche horizontale Position belegen.

3.1.2 Repräsentation von Schemata

Von X-ESCHER wird nicht nur die Tabelle visualisiert, sondern auch ihre Struktur, also das Schema. Dadurch erhält der Benutzer zusätzliche Strukturinformation über die Tabelle. Diese Information ist jedoch nur hilfreich, wenn die Repräsentation von Schema und Tabelle zueinander passen; sie kann sogar störend sein, wenn sie falsche Zusammenhänge suggeriert. Die Schemata werden daher ebenfalls in einer geschachtelten Form dargestellt. Um die visuelle Korrelation zur Tabellenrepräsentation zu erhalten, werden atomare Attribute durch ihren Namen repräsentiert und in der im Schema festgelegten Breite dargestellt. Die im Schema festgelegten Breiten der einzelnen Attribute werden also sowohl in der Schema- als auch in der Tabellenrepräsentation benutzt.

In der Schemarepräsentation wird zwischen komplexen und atomaren Attributen unterschieden:

- Komplexe Attribute werden dargestellt, indem ihr Attributname, getrennt durch eine horizontale Linie, oberhalb der Elementstruktur bzw. Komponentenstrukturen



(a) Schemabaum

bisdieBox	1	vollist		w
	2			x
	3			y
	4			z
a vollist	5	auchdiese		
breiteBox	6			
	7	6	{}	
	8			

(b) unkorreliert

bisdieBox	1	vollist	w
	2		x
	3		y
	4		z
a	vollist	5	auchdiese
breiteBox	6		
	7	6	{}
	8		

(c) korreliert

Abbildung 3.2: Berechnung von Attributbreiten

Beispiel 3.2 Berechnung von Attributbreiten

In Abb. 3.2(a) ist ein Schema dargestellt, dem ein Typausdruck gemäß [The96] der folgenden Form entspricht:

```
Beispiel:{[A:string B:{b:string} C:string D:{d:string}]}
```

Ein Datenobjekt wird hier in der üblichen Klammer-Notation angeben[2]:

```
{ ["bisdieBox" {"1" "2" "3" "4"} "vollist" {"w" "x" "y" "z"}]
  ["a" {"vollist"} "5" {"auchdiese"}]
  ["breiteBox" {"6" "7" "8"} "6" {}] }
```

Die Beispiel-Menge besteht aus drei Tupeln; das erste Tupel hat als Komponenten die beiden atomaren Werte „bisdieBox“ (Attribut A) und „vollist“ (Attribut C), sowie zwei Mengen, die Ziffern bzw. Buchstaben enthalten, usw.

Für die Berechnung der Darstellungsbreite der b-Werte reicht es nicht aus, das Maximum der Darstellungsbreiten von b-Werten eines Beispiel-Tupels zu bilden. Das würde zu einer tabellarischen Darstellung mit unkorrelierten Breiten wie in Abb. 3.2(b) führen[3], die eine falsche visuelle Korrelation suggeriert (man betrachte z. B. den C-Wert „5“).

Das Maximum muß aus den Darstellungsbreiten *aller* b-Werte bestimmt werden; insbesondere hieße das, daß auch die Visualisierungsbreiten von nicht sichtbaren Sub-Objekten berechnet werden müßten. Die daraus resultierende tabellarische Darstellung mit korrelierten Breiten ist in Abb. 3.2 dargestellt. □

[2]Diese Klammer-Notation ist strukturell vergleichbar mit Typausdrücken ohne Attributnamen und wurde in einer ähnlichen Form bereits im AIM-P Projekt benutzt [LPS91].

[3]Das Beispiel wirkt etwas künstlich, erfüllt aber seinen Zweck. Diese naive Darstellung würde z. B. mit Tcl/Tk beim einfachen Schachteln von `frame` und `label` Widgets durch den `pack` Geometriemanager [Ous94, S. 187] entstehen, dem ein sog. „Höhlenmodell“ (*cavity-based model* [HM98, S. 16]) zugrunde liegt.

zentriert angeordnet wird. Der Attributname hat dabei die gleiche Breite wie die Sub-Struktur(en), so daß ein rechteckiger Bereich entsteht[4]. Komponentenstrukturen werden horizontal konkateniert, getrennt durch vertikale Linien. Für genuine Tupel wird also der Attributname zentriert über den Komponentenstrukturen dargestellt. Mengen werden durch ihren Attributnamen, angeordnet über der Elementstruktur, dargestellt.

- Atomare Attribute werden durch ihren Attributnamen, der zentriert in einem rechteckigen Bereich angeordnet wird, dargestellt.

Alle Attributnamen sind zusätzlich mit einem Präfix versehen, an dem der Typ des Attributes abgelesen werden kann. Z.B. haben Namen von Mengen das Präfix „`{}`“, Integer-Attribute sind mit dem Zeichen „`#`“ gekennzeichnet. Dem Typ String ist das leere Präfix zugeordnet.

Die Höhen der Rechtecke, die die Attributnamen von komplexen Attributen enthalten, sind alle gleich. Die Höhen der Rechtecke, die die Sub-Attribute enthalten, sind jeweils durch die maximale Höhe eines Sub-Attributes festgelegt, so daß alle Sub-Attribute eines komplexen Attributes die gleiche Höhe haben. Insbesondere werden die Höhen der Rechtecke, die die Attributnamen der atomaren Attribute enthalten, dazu so angepaßt, daß alle Rechtecke der atomaren Attribute auf einer Linie enden, wodurch die Repräsentation des Schemas insgesamt einen rechteckigen Bereich darstellt.

Die Repräsentationen von Tabelle und Schema sind, jeweils in einem eigenen Fenster, übereinander angeordnet. Übersteigt die Größe der dargestellten Objekte die Größe der Fenster, ist die Position des sichtbaren Ausschnitts (*viewport*) durch Rollbalken veränderbar. Dazu sind nur drei Rollbalken notwendig, da die horizontale Viewport-Veränderung[5] für Schema und Tabelle synchron über einen einzigen Rollbalken erfolgt.

Abb. 3.3 auf der gegenüberliegenden Seite führt Beispiel 3.2 fort und zeigt eine X-ESCHER-Repräsentation von Beispielschema und -tabelle. In Abb. 3.4 auf Seite 68 wird Beispiel 3.1 fortgeführt und die Vorlesungstabelle mit Schema dargestellt.

3.1.3 Fingervisualisierung und Interaktion

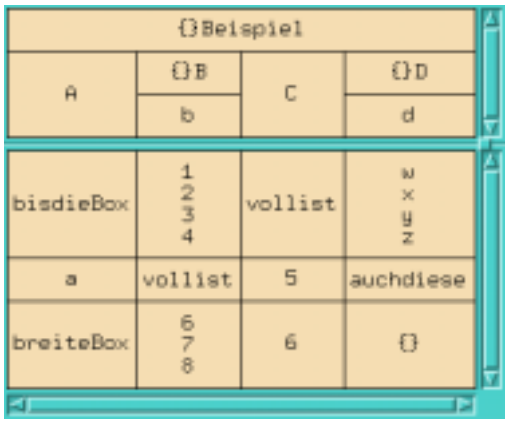
Finger wurden in Abschnitt 2.3.2 eingeführt. Dem Benutzer von X-ESCHER werden zur Interaktion mit komplexen Datenobjekten Finger durch ihre visuelle Repräsentation zugänglich gemacht. Ein Finger wird dargestellt, indem das Objekt, auf das er zeigt, durch Färbung hervorgehoben wird. Durch die rekursiv strukturierte, ineinander geschachtelte Tabellenrepräsentation bedeckt ein Finger in ihr immer einen zusammenhängenden, rechteckigen Bereich.

[4] Pseudo-Tupel haben keinen Attributnamen, so daß das obere Rechteck für diesen Spezialfall fehlt.

[5] Mit dem Begriff „Viewport-Veränderung“ ist hier immer die Veränderung der Position des Viewports gemeint, nicht die Veränderung seiner Größe.

{}Beispiel			
A	{}B	C	{}D
	b		d
bisdieBox	1	vollist	w
	2		x
	3		y
	4		z
a	vollist	5	auchdiese
breiteBox	6	6	{}
	7		
	8		

(a) Schema und Tabelle korreliert



(b) X-ESCHER-Repräsentation

Abbildung 3.3: Visuelle Korrelation mit Schema

Beispiel 3.3 Visuelle Korrelation mit Schema

Das Datenobjekt aus Beispiel 3.2 ist in Abb. 3.3(a) mit zugehörigem Schema dargestellt. Da die Struktur des Schemas und die Daten in der Tabelle korreliert dargestellt sind, können aus dem Schema Attributnamen und -typen abgelesen und den Daten zugeordnet werden.

Abb. 3.3(b) zeigt die X-ESCHER-Repräsentation der Tabelle mit zugehörigem Schema. Sie unterscheidet sich von der Darstellung in Abb. 3.3 im Wesentlichen nur durch die fehlenden horizontalen Linien zwischen den atomaren Mengenelementen[6] — die Gesamterscheinung wirkt wesentlich „ruhiger“. Die Darstellung erlaubt eine intuitive Erfassung von Struktur und Inhalt (Wert) des Datenobjektes. □

[6]Siehe Fußnote 1 auf Seite 62.

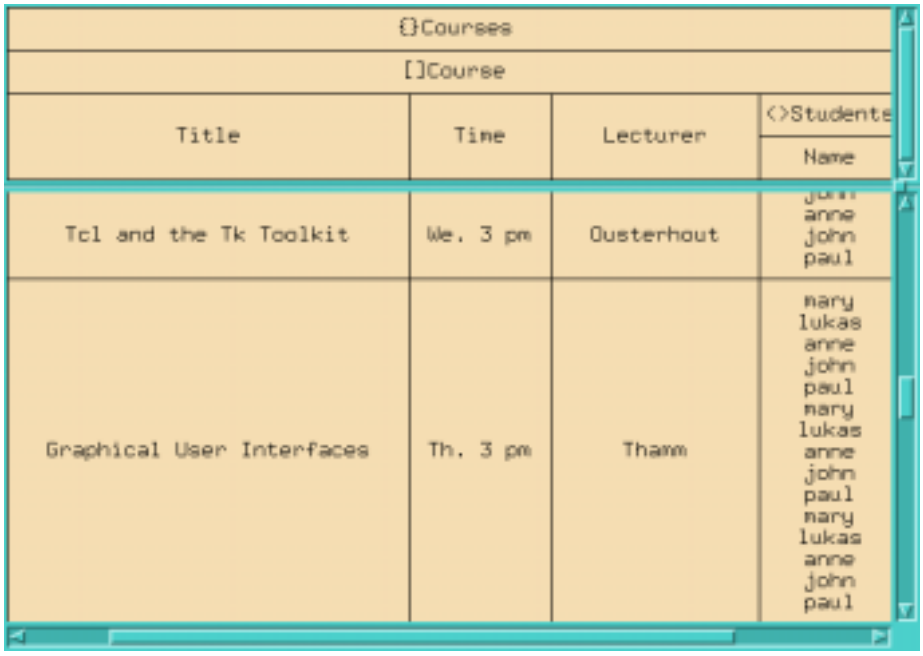


Abbildung 3.4: X-ESCHER-Repräsentation der Vorlesungstabelle mit Schema

Beispiel 3.4 X-ESCHER-Repräsentation der Vorlesungstabelle mit Schema

In der Abbildung ist die Breite des Viewports kleiner als die der Repräsentation. Der sichtbare Ausschnitt wurde mit dem Rollbalken so eingestellt, daß die Attribute ID, Dep und Req nicht sichtbar sind. Ähnlich wie einige Werte in Abb. 3.1 auf Seite 63 vertikal zentriert dargestellt wurden, sind hier die Attributnamen Courses und Course im sichtbaren Bereich horizontal zentriert dargestellt. So bleiben sie auch angezeigt, wenn der sichtbare Bereich horizontal weit von der Mitte entfernt ist (siehe z. B. Abb. 3.6(b) auf Seite 71).

Im Schema ist der Attributname der Menge Courses ist mit dem Präfix „{}“ versehen und erstreckt sich über die ganze Breite. Das genuine Tupel Course mit dem Präfix „[]“ ist ähnlich dargestellt. Die Darstellungshöhe der atomaren Komponenten Title, Time und Lecturer sind an die der komplex strukturierten Komponente Students angepaßt. Die Liste Students ist mit dem Präfix <> versehen und enthält ein Pseudo-Tupel als Elementtyp, dessen Komponente Name daher direkt unter dem Listenattributnamen dargestellt ist. Hier wird der bereits in Abschnitt 2.3.1 auf Seite 45 erwähnte Vorteil der Pseudo-Tupel in der Schemarepräsentation deutlich: Die Darstellung des genuinen Course-Tupels ist als Strukturinformation nicht notwendig und daher eigentlich unerwünscht, da sie die Schemarepräsentation komplizierter und damit weniger intuitiv macht.

Alle atomaren Attribute sind hier vom Typ String. Atomare Attribute, die nicht vom Typ String sind und kein leeres Präfix haben, zeigt Abb. 3.7 auf Seite 73. □

Verschiedene Finger werden durch unterschiedliche Färbungen dargestellt. Damit ein Finger F_1, der in einem anderen Finger F_2 enthalten ist ($F_1 \Subset F_2$), nicht vollständig von diesem verdeckt wird, definiert $\Subset$ eine Stapelungsreihenfolge für ineinander enthaltene Finger. Kongruente Finger werden durch eine besondere farbliche Markierung dargestellt. In Abb 3.5 auf der nächsten Seite sind verschiedene Finger in der bereits bekannten Vorlesungstabelle dargestellt.

Durch die visualisierten Finger wird beim Navigieren in einer Tabelle stets ein Kontext gewahrt. Üblicherweise wird die Fingerinteraktion über die Tastatur gesteuert, wobei es immer einen aktiven Finger gibt, auf den sich die Fingeroperationen beziehen. Die Namen *enter*, *escape*, *next*, *back*, *insert*, *delete* und *backspace* der Fingeroperationen entsprechen der Benennung der Tasten, mit denen die Fingeroperationen ausgelöst werden können. So werden Finger z. B. durch die Pfeiltasten von einem Objekt zum nächsten bewegt, d. h. von einem Element zum nächsten in einer Kollektion oder von einer Komponente zur nächsten in einem Tupel.

Änderungsoperationen werden relativ zum Finger in Analogie zum Cursor eines Text-Editors ausgeführt, z. B. Einfügen und Löschen vor oder nach dem Finger bzw. Cursor. Einige Fingeroperationen werden nicht mit „eigenen“ Tastenbelegungen, sondern kontextabhängig ausgelöst. Zeigt ein Finger z. B. auf eine nullwertige Kollektion, bewirkt das „Hineingehen“ zunächst eine Umwandlung in eine leere Kollektion (Operation *to_ empty*). Da der Finger nicht in eine leere Kollektion hineinzeigen kann, wird seine Position nicht verändert. Beim nächste Hineingehen wird die leere Kollektion dann in eine einelementige Kollektion umgewandelt (Operation *to_ singleton*) und der Finger auf dem neuen Element positioniert.

Neben der Fingervisualisierung in der Tabellenrepräsentation wird von X-ESCHER auch die einem Finger entsprechende Position in der Schemarepräsentation visualisiert. Dabei wird auch die Schemakomponente, die die Struktur des Objektes definiert, auf das der Finger zeigt, durch Färbung hervorgehoben. Beide Fingervisualisierungen verändern ihre Position bei Fingerbewegungen synchron.

Die Viewport-Veränderung ist an die Fingerbewegung gekoppelt: Würde der aktive Finger durch eine Navigationsoperation den Viewport „verlassen“, d. h. auf ein nicht sichtbares Objekt zeigen, wird die Position des Viewports automatisch so verändert, daß der Finger im Viewport sichtbar bleibt.

Durch die übereinstimmenden Visualisierungsbreiten der Datenobjekte und der entsprechenden Attribute im Schema, die übereinander angeordneten Fenster und die synchrone Veränderung des Viewports entsteht eine visuelle Korrelation zwischen den in Tabellen- und Schemarepräsentation zusammengehörigen Fingern. Erst durch diese zusätzliche Strukturinformation wird die Position und der Kontext eines Fingers in einem komplexen Datenobjekt intuitiv erfaßbar.

Abb. 3.6 auf Seite 71 zeigt eine X-ESCHER-Repräsentation der Vorlesungstabelle mit Fingervisualisierungen.

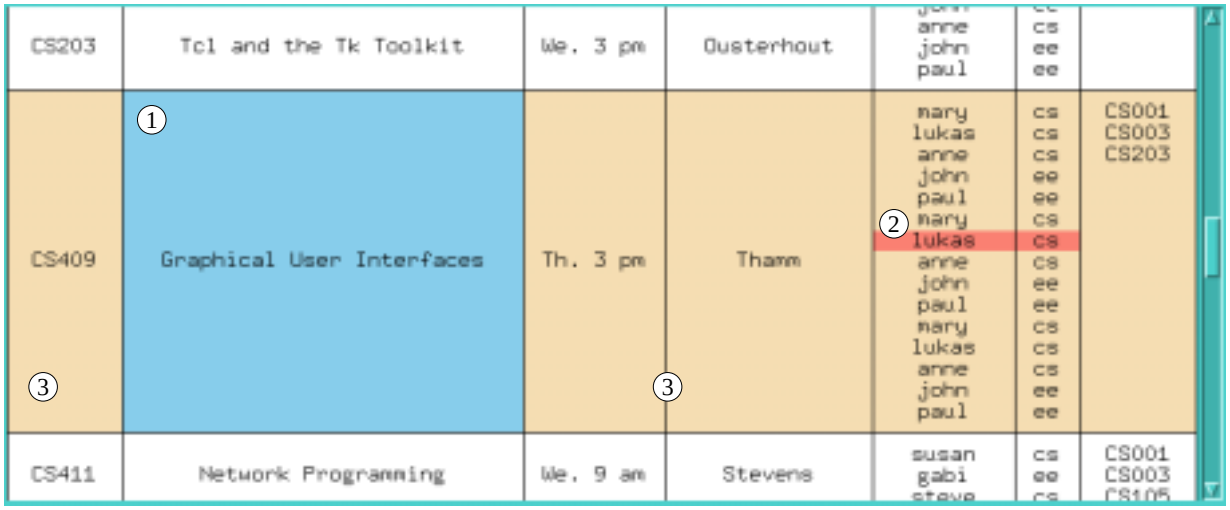


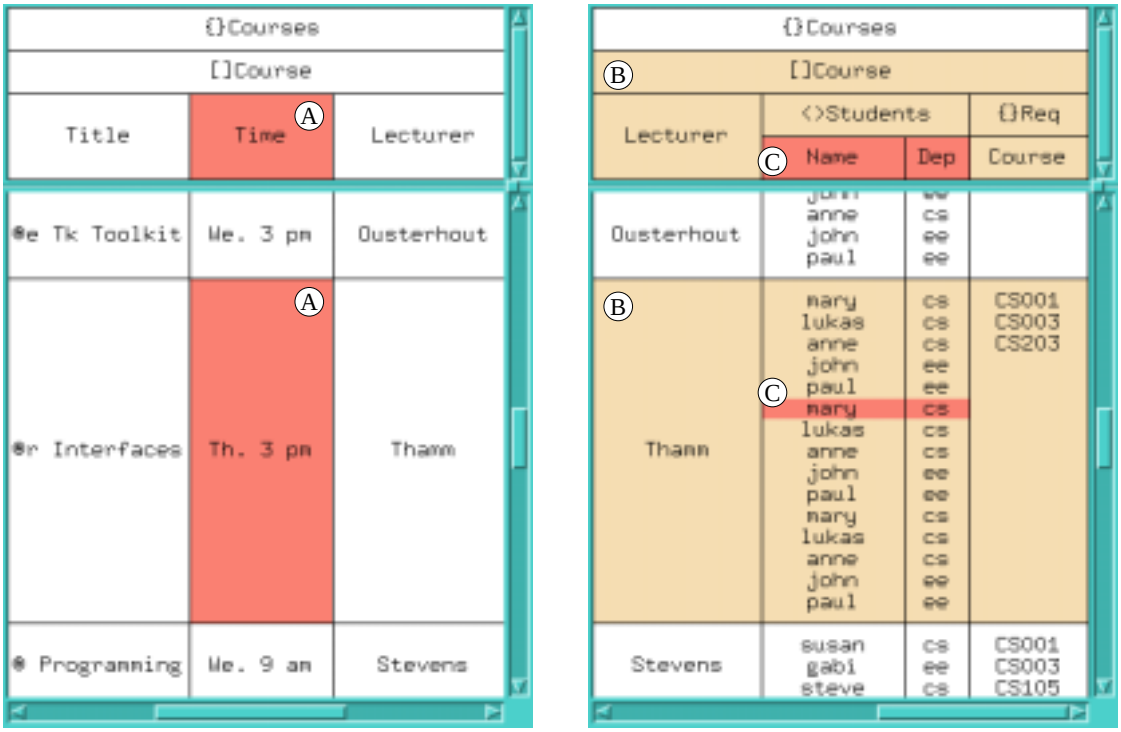
Abbildung 3.5: X-ESCHER-Repräsentation der Vorlesungstabelle mit Fingern

Beispiel 3.5 Fingern in verschiedenen Verhältnissen zueinander

In der Abbildung ist die X-ESCHER-Repräsentation der Vorlesungstabelle mit drei Fingern F_1, F_2 und F_3 dargestellt, die mit ①, ② und ③ markiert sind. F_3 liegt zuunterst, da $F_3 \Subset F_1$ und $F_3 \Subset F_2$ gilt. Die von F_3 bedeckte Fläche erscheint nur deswegen unzusammenhängend, weil F_3 teilweise durch F_1 verdeckt ist. F_1 und F_2 sind disjunkt ($F_1 \Cap F_2$) und zeigen jeweils auf ein Sub-Objekt (den Titel „Graphical User Interfaces“ der Vorlesung) bzw. Sub-Sub-Objekt (den Teilnehmer „lukas“ aus dem Fachbereich „cs“) des Datenobjektes, auf das Finger F_3 zeigt (die ganze Vorlesung). Die Darstellung kongruenter Finger wäre in diesem Bild nicht erkennbar, da sich deren farbliche Markierung in Graustufen zu wenig abhebt. □

Beispiel 3.6 Bewegung von Fingern

Für Finger F_2 würde die ↑-Taste die *back*-Operation auslösen und ihn auf dem vorhergehenden Element positionieren, hier der Teilnehmerin „mary“ aus der Informatik. Die ↵-Taste (Enter, Return) würde die *in*-Operation auslösen und F_2 auf die erste Komponente des Tupels zeigen lassen, hier den Namen „lukas“. □



(a) Ein Finger (b) Zwei Finger

Abbildung 3.6: X-ESCHER-Repräsentation der Vorlesungstabelle mit Schema und Fingern

Beispiel 3.7 X-ESCHER-Repräsentation der Vorlesungstabelle mit Schema und Fingern

In Abb. 3.6(a) zeigt ein Finger Ⓐ auf eine Vorlesungszeit (Attribut Time). In Abb. 3.6(b) sind zwei Finger dargestellt: Einer Ⓑ zeigt auf ein Vorlesungs-Tupel, der andere Ⓒ auf eine Teilnehmerin der Vorlesung.

In beiden Abbildungen sind die Breiten der Viewports stark reduziert, also schmaler als die Darstellungsbreiten der Tabellen bzw. Schemata. Die Positionen der Viewports sind jeweils so angepaßt, daß die Finger sichtbar sind. Wenn Finger Ⓐ nach links bewegt wird, dann wird dementsprechend im Schema das Attribut Title markiert. Außerdem wird der Viewport so verändert, daß die Titel komplett sichtbar sind — in Abb. 3.6(a) sind sie links abgeschnitten, symbolisiert durch das Zeichen „@“ am linken Rand[7]. □

[7] Dieses „Abschneiden mit Markierung“ wäre mit Clipping durch die Hardware oder das Fenstersystem nicht realisierbar.

3.1.4 Repräsentation von Schemata als Tabellen

Wie in Abschnitt 2.3.3 erläutert wurde, sind auch Schemata Tabellen, deren Struktur durch das Boot-Schema definiert wird. Dementsprechend können Schemata auch als Tabellen dargestellt werden, und in dieser Darstellung prinzipiell editiert werden. Jedoch wird man dem Anwender das Editieren nicht gestatten, weil Inkonsistenzen aufgrund mangelnder Kenntnis der internen Strukturen leicht möglich sind.

Beispiel 3.8 X-ESCHER-Repräsentationen von Schemata als Tabellen

In Abb. 3.7 auf der gegenüberliegenden Seite sind die X-ESCHER-Repräsentationen von zwei Schemata als Tabellen dargestellt. Abb. 3.7(a) zeigt das Schema Courses aus Beispiel 2.1 auf Seite 46 als Datenobjekt. Ein Finger zeigt auf die Sub-Attribut-Menge, die die Komponenten eines Course-Tupels definiert. Die zweifache Darstellung der Menge SUBATTR im Schema direkt untereinander resultiert aus der in Beispiel 3.4 auf Seite 68 angesprochenen Zentrierung der Attributnamen bei horizontal eingeschränktem Viewport.

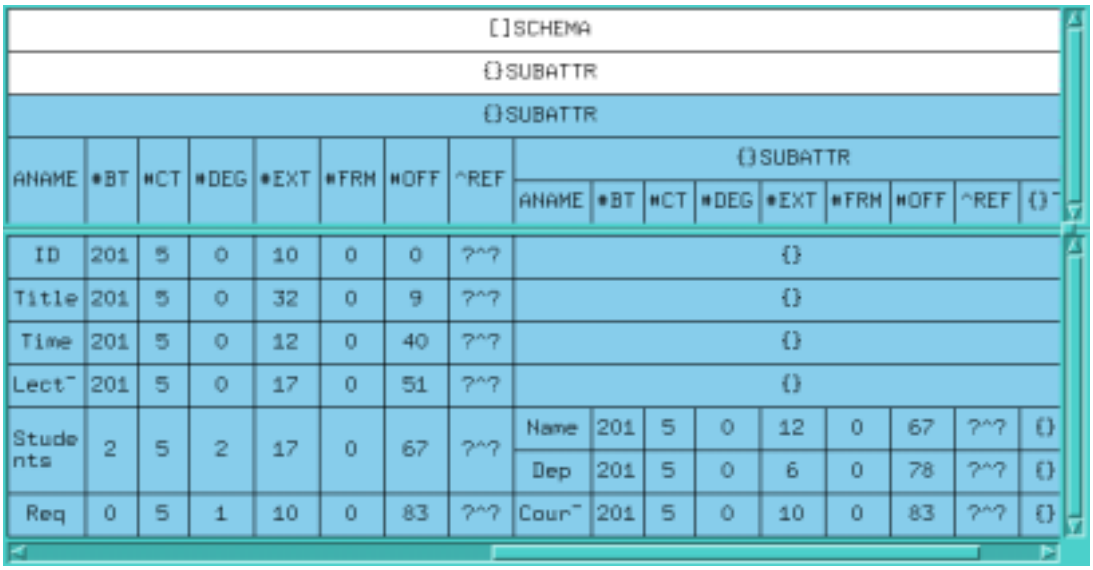
Abb. 3.7(b) zeigt das Boot-Schema als Tabelle, d. h. es ist einmal als Datenobjekt (im unteren Bereich), und einmal als Schema (im oberen Bereich) dargestellt. Außerdem sind drei Finger dargestellt: Finger Ⓐ zeigt auf die Definition des äußersten Attributes (genuines Tupel SCHEMA), Finger Ⓑ zeigt auf die Definition des Attributnamens (String ANAME und Finger Ⓒ auf die Definition der Tupelkomponenten (Menge SUBATTR)[8]. Am rechten Rand läßt sich die Rekursivität des Boot-Schemas und die damit verbundene Selbstähnlichkeit erkennen. □

Durch die Benutzung eines rekursiven Boot-Schemas entstehen für die Repräsentation eines Schemas als Tabelle zwei grundlegende Probleme. Erstens ist die Breite eines Attributes nicht mehr a priori festgelegt, sondern von den Daten abhängig. Je größer die Schachtelungstiefe eines Schemas ist, desto breiter ist die Repräsentation des Schemas als Tabelle.

Zweitens sind die Repräsentation des Boot-Schemas als Tabelle und auch als Schema theoretisch unendlich. Rekursiv definierte Schemata sind eigentlich nichts ungewöhnliches. Z. B. kann zur Modellierung von Baugruppen eine Bauteilstruktur durch eine Menge von Unterbauteilen definiert sein, wobei die Unterbauteile selbst wieder Bauteile sind. Ungewöhnlich ist aber die Interpretation des Boot-Schemas als rekursives Datenobjekt. Ein Baugruppenobjekt ist dagegen immer endlich, da die „kleinsten“ Teile einer Baugruppe keine Unterbauteile haben.

Das erste Problem, die datenabhängige Attributbreite, wurde bei der Implementierung als Spezialfall für die Tabellenrepräsentation von Schemata berücksichtigt. Dabei wurde die Tatsache, daß nur das letzte Sub-Attribut eines Tupels rekursiv definiert ist, das Boot-Schema also quasi endrekursiv ist, ausgenutzt. Das zweite Problem, die Unendlichkeit des

[8] Das Boot-Schema ist aus internen Gründen von dem genuinen Tupel SCHEMA umgeben, das exakt wie die Pseudo-Tupel der Sub-Attribut-Mengen strukturiert ist.



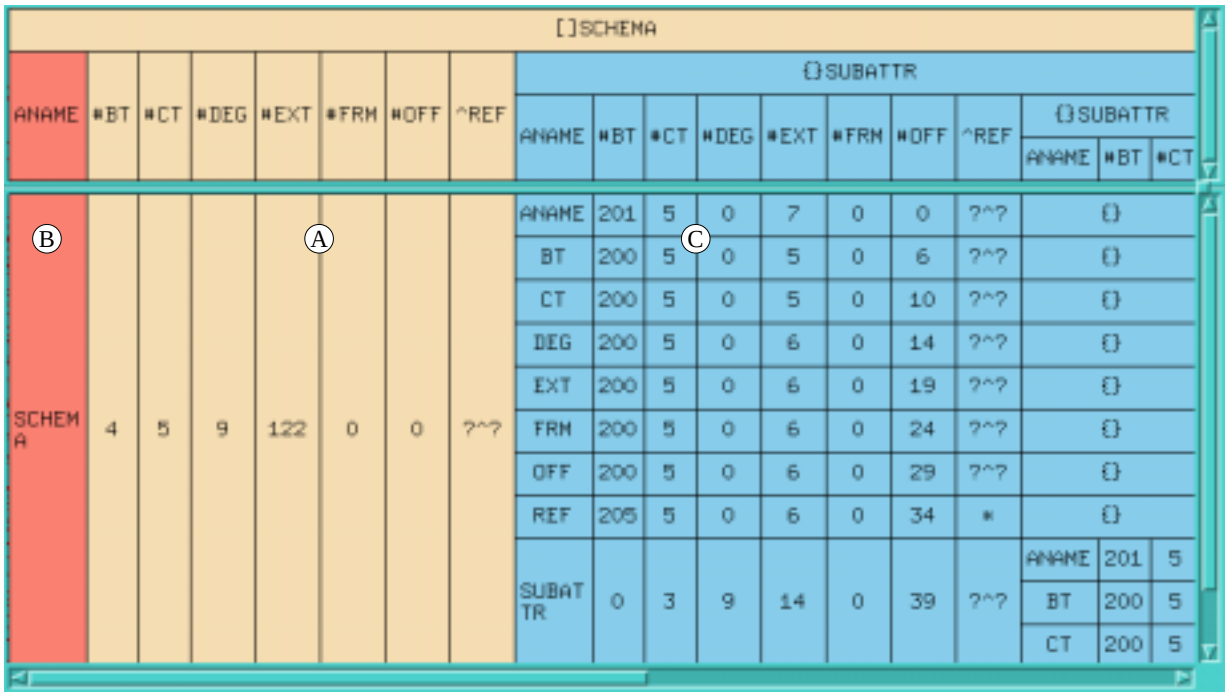
[]SCHEMA

{}SUBATTR

{}SUBATTR

ANAME	#BT	#CT	#DEG	#EXT	#FRM	#OFF	^REF	{}SUBATTR							
								ANAME	#BT	#CT	#DEG	#EXT	#FRM	#OFF	^REF
ID	201	5	0	10	0	0	?^?	{}							
Title	201	5	0	32	0	9	?^?	{}							
Time	201	5	0	12	0	40	?^?	{}							
Lect~	201	5	0	17	0	51	?^?	{}							
Students	2	5	2	17	0	67	?^?	Name	201	5	0	12	0	67	?^?
								Dep	201	5	0	6	0	78	?^?
Req	0	5	1	10	0	83	?^?	Cour~	201	5	0	10	0	83	?^?

(a) Schema Courses



[]SCHEMA

ANAME	#BT	#CT	#DEG	#EXT	#FRM	#OFF	^REF	{}SUBATTR									
								ANAME	#BT	#CT	#DEG	#EXT	#FRM	#OFF	^REF	{}SUBATTR		
																ANAME	#BT	#CT
SCHEMA	4	5	9	122	0	0	?^?	ANAME	201	5	0	7	0	0	?^?	{}		
								BT	200	5	0	5	0	6	?^?	{}		
								CT	200	5	0	5	0	10	?^?	{}		
								DEG	200	5	0	6	0	14	?^?	{}		
								EXT	200	5	0	6	0	19	?^?	{}		
								FRM	200	5	0	6	0	24	?^?	{}		
								OFF	200	5	0	6	0	29	?^?	{}		
								REF	205	5	0	6	0	34	*	{}		
								SUBATTR	0	3	9	14	0	39	?^?	ANAME	201	5
																BT	200	5
																CT	200	5

(b) Boot-Schema

Abbildung 3.7: X-ESCHER-Repräsentationen von Schemata als Tabellen

Boot-Schemas, wird für dessen Repräsentation durch einen beschränkten sichtbaren Bereich umgangen, d. h. nur der endliche, sichtbare Teil der theoretisch unendlichen Repräsentation wird auch tatsächlich generiert (*lazy evaluation*). Die Länge der Rollbalken wird durch die Größe der tatsächlich generierten Repräsentation bestimmt.

3.1.5 Manipulation von Datenobjekten

Die Änderung eines Datenobjektes kann in zwei Fälle zerlegt werden:

1. Kollektionen können durch Einfügen und Löschen von Elementen sowie das Löschen der ganzen Kollektion verändert werden (siehe die in Abschnitt 2.3.2 beschriebenen Operationen und Abschnitt 3.1.3 zu Fingern).

2. Atomare Werte können durch die Eingabe einer textuellen Repräsentation verändert werden. In X-Escher ist dies in einem dedizierten Eingabebereich realisiert; ausgelöst wird das Ändern eines atomaren Wertes, indem die *in*-Operation auf einen Finger angewendet wird, der auf ein atomares Objekt zeigt, also „in den atomaren Wert hinein“ gegangen wird.

Sowohl Kollektionen als auch atomare Objekte können auf Null gesetzt, d. h. durch einen Nullwert ersetzt werden.

3.2 Eine Zwischenbilanz

Im Folgenden werden die wesentlichen Nachteile des gerade beschriebenen Visualisierungskonzeptes von X-Escher beschrieben und daraus Anforderungen an das zu entwickelnde Visualisierungsverfahren abgeleitet. Außerdem werden beispielhaft einige Überlegungen angestellt, welche Visualisierungsalternativen bei der Entwicklung berücksichtigt werden sollen.

3.2.1 Nachteile des bisherigen Visualisierungskonzeptes

Die Repräsentation einer Tabelle in X-Escher orientiert sich am Schema dieser Tabelle. Es ist nicht möglich, diese Repräsentation zu verändern, etwa um wichtige Teile der Tabelle hervorzuheben, oder unwichtige Teile der Tabelle auszublenden. Wird eine andere Repräsentation gewünscht, muß bisher also ein entsprechendes Schema erzeugt und eine, ggf. modifizierte Kopie der Tabelle angelegt werden.

Einzig die Breiten der atomaren Attribute eines Schemas sind in X-ESCHER parametrisiert und können für ein Schema beliebig eingestellt werden. Damit ist aber auch schon die Repräsentation jeder Tabelle, die mit diesem Schema strukturiert wurde, festgelegt. Tabellen- oder benutzerspezifische Repräsentationen sind nicht möglich.

Analog zum Konzept der Sichten in relationalen Datenbanken, die abgeleitete Tabellen als andere Sicht auf Basistabellen ohne Redundanz realisieren, könnte eine Repräsentation als Sicht auf visueller Ebene, also als abgeleitete Repräsentation eines Schemas, aufgefaßt werden.

Das Visualisierungskonzept ist en bloc implementiert. Für jeden Typ bzw. Typkonstruktor ist genau eine Darstellungsart realisiert, die auf alle Instanzen angewendet wird. Änderungen der Darstellung können nur durch Re-Implementierung, d. h. den üblichen Programmieren-Compilieren-Ausprobieren-Zyklus realisiert werden. Dadurch ist die Repräsentation auch nicht zur Laufzeit, etwa durch in den Daten gespeicherte Anweisungen, veränderbar.

Die in Abschnitt 2.3.4 angesprochene Erweiterung von X-ESCHER um multimediale Datentypen zeigte, wie schwierig X-ESCHERs Visualisierungskonzept erweiterbar ist. Bereits die Tatsache, daß Bilder höher sind als eine Zeile, wie alle anderen Darstellungen atomarer Typen, hatte ein aufwendiges Re-Design des Visualisierungsalgorithmus zur Folge. Auch die Eingabe von Bildern war nicht mehr über die textorientierte Schnittstelle möglich. Hier mußte eine spezielle Lösung implementiert und in X-ESCHER integriert werden, was den Aufwand dieser Erweiterung nochmal erhöhte.

Wäre X-ESCHERs Visualisierungsverfahren erweiterbar, hätte — zusammen mit der Erweiterung des Datenmodells aus [The96] und der damit möglichen Strukturierung der Bilder als Objekttypen (siehe Abb. 2.7 auf Seite 59) — die Darstellung als Erweiterung des Visualisierungsverfahrens umgesetzt werden können.

Die Erfahrung im ESCHER-Projekt hat gezeigt, daß bereits die Strukturierungsmöglichkeiten des eNF^2-Datenmodells so komplex sind, daß die generische Repräsentation nicht immer ausreichend ist, um beliebige Datenobjekte intuitiv verständlich darstellen zu können. Beispiele für Datenobjekte, deren generische Repräsentation zu Problemen führt, sind direkt ineinander geschachtelte Kollektionen, hierarchisch strukturierte Objekte, deren Granularität auf verschiedenen Hierarchieebenen stark variiert, und ganz allgemein „große“ Objekte.

Auch die Interaktion ist durch die fest vorgegebenen Zuordnungen zwischen Tastatur und Fingeroperationen sehr starr, und insbesondere weder konfigurierbar noch erweiterbar. Strukturelle Umwandlungen [Rod92] durch

- *nest* und *unnest*-Operationen [TF86],
- Löschen und Einfügen von Komponenten in Tupeln,
- Veränderung von atomaren Datentypen und
- Austausch von Kollektions-Konstruktoren

sind im Interaktions- und Visualisierungskonzept nicht berücksichtigt.

Diese Nachteile werden nun als Anforderungen für das Visualisierungsverfahren, das in den folgenden Abschnitten und Kapiteln entwickelt werden soll, formuliert:

- Für ein Schema sollen beliebig viele Visualisierungen unabhängig voneinander definiert werden können.
- Diese sollen jede Tabelle, die durch das Schema strukturiert wird, visualisieren können.
- Die Visualisierungen sollen, wie Tabellen und Schemata, persistente Datenbankobjekte sein.
- Das Visualisierungsverfahren soll erweiterbar sein in dem Sinn, daß neue Visualisierungskonzepte, z. B. für einzelne Konstruktoren oder Objekttypen, definiert und integriert werden können.
- Das Visualisierungsverfahren soll flexibel sein in dem Sinn, daß für die einzelnen Attribute eines Schemas entsprechend ihres Typs aus einem Vorrat von Visualisierungskonzepten gewählt werden kann.
- Das Visualisierungsverfahren soll eine interpretierende Komponente enthalten, die eine Modifikation zur Laufzeit ermöglicht, insbesondere auch durch in den Daten gespeicherte Anweisungen.
- Die Erweiterbar und Konfigurierbarkeit des Visualisierungsverfahrens soll sich auch auf die Interaktion beziehen.

Es wird sich zeigen, daß alle Anforderungen erfüllt werden können und die oben genannten Nachteile mit dem vorgestellten Visualisierungsverfahren nicht mehr bestehen.

Auch einige Stärken von X-ESCHERs Visualisierungskonzept sollen als Anforderungen für das zu entwickelnde Visualisierungsverfahren formuliert werden:

- Das Visualisierungsverfahren soll beliebig komplexe Objekte ohne Benutzerkonfiguration in einer Grunddarstellung (*default representation*, Default-Repräsentation) anzeigen können.
- Das Visualisierungsverfahren soll das Konzept der visuellen Korrelation, auch zwischen Repräsentationen von Tabelle und Schema bzw. Datenobjekt und Struktur, unterstützen.
- Das Visualisierungsverfahren soll ESCHERs Fingerkonzept integrieren.

3.2.2 Visualisierungsalternativen

Für jeden der drei in Abschnitt 3.1.1 aufgeführten Fälle sollen hier einige Visualisierungsalternativen betrachtet werden. Die Beobachtungen werden dann verallgemeinert und Auswirkungen bzw. Anforderungen an das zu entwickelnde Visualisierungsverfahren formuliert.

Darstellungen atomarer Objekte müssen keine festen Breiten und/oder Höhen haben. Insbesondere Zeichenketten (Strings) haben oft unterschiedliche Längen, so daß eine einzeilige Darstellung mit fester Breite für einige Strings zu kurz sein wird oder von anderen Strings nur zu einem Bruchteil ausgefüllt wird. Werden sie als Elemente einer Menge untereinander dargestellt, sollten sie trotzdem alle die gleiche Breite haben, damit ein homogenes Bild entsteht. Lange Strings könnten in diesem Fall durch Umbruch in mehreren Zeilen dargestellt werden, wodurch implizit eine variable Höhe der Darstellungen entsteht. Eine Anforderung an das Visualisierungsverfahren ist, daß auch für atomare Objekte variable Längen erlaubt sein sollen.

Kollektionen können auch dargestellt werden, indem ihre Elemente horizontal konkateniert werden. Dadurch ensteht sofort eine variable Breite der Darstellung. Eine weitere Möglichkeit ist die Kombinationen von horizontaler und vertikaler Konkatenation, wodurch eine matrixähnliche Darstellung von Kollektionen entsteht. Werden Kollektionen elementweise dargestellt (*one-tuple-at-a-time*, ein Element zu einer Zeit), entsteht eine formularartige Präsentation. Die Auswahl der Elemente kann über Vor/Zurück-Knöpfe, Rollbalken, oder ähnliches erfolgen. Besitzen die Elemente einen Schlüssel[9], kann die Auswahl auch über Indexe, z. B. in Form einer Combo-Box, erfolgen. Zur Handhabung „großer“ Kollektionen können z. B. Darstellungen mit Rollbalken, Fischaugen und Faltungen oder Kombinationen davon herangezogen werden.

Ähnliches gilt für Darstellungen von Tupeln. Neben der horizontalen Konkatenation kann auch die vertikale Konkatenation, und die Kombination beider, eingesetzt werden. Durch Komponenten mit variabel Breiten Darstellungen können auch Darstellungen von Tupeln variable Breiten erhalten, wiederum können Rollbalken usw. eingesetzt werden.

Eine um 90° gedrehte Darstellung beinhaltet die Vertauschung der Darstellung von Kollektionen und Tupeln, d. h. Kollektionen werden horizontal konkateniert, Tupel vertikal konkateniert; Tupelkomponenten werden dann in ihrer Breite statt in ihrer Höhe angepaßt.

3.2.3 Schlußfolgerungen

Ein generisches, aber starres Visualisierungskonzept ist nicht ausreichend, um beliebige, komplex strukturierte Datenobjekte angemessen zu visualisieren. Daher soll ein Visualisierungsverfahren entwickelt werden, das flexibel genug ist, um für spezielle Anforderungen

[9] Eine Forderung der *partitioned normal form* (PNF) [RKS88, Hul90].

oder individuelle Visualisierungsaspekte modifiziert werden zu können. Die Modifizierbarkeit umfaßt dabei zwei Aspekte: Einerseits soll es konfigurierbar sein, d. h. aus vorhandenen Alternativen sollen Teile von Repräsentationen adäquat gestaltet werden können. Die Erweiterbarkeit ist eine logische Fortsetzung der Konfigurierbarkeit, indem die zur Verfügung stehenden Darstellungsmöglichkeiten an neue Anforderungen angepaßt werden können. Das Verfahren soll dabei seine generische Natur, d. h. die Anwendbarkeit auf beliebige, komplex strukturierte Datenobjekte *ohne* Konfiguration nicht verlieren. Außerdem sollen die generierten Repräsentationen als Komponenten in graphische Benutzerschnittstellen integriert werden können.

Ziel ist es also, ein Visualisierungsverfahren für komplex strukturierte Datenobjekte zu entwickeln, das

- generisch,
- konfigurierbar und
- erweiterbar ist, und
- Repräsentationen als Komponenten generiert.

Das bedeutet für das Visualisierungsverfahren:

Individuelle Darstellung von Teilobjekten bzw. einzelnen Schemakomponenten:
: Für jede Komponente eines Schemas soll individuell entschieden werden können, wie die durch ihn strukturierten Objekte dargestellt werden. Dabei soll die Visualisierung einer Schemakomponente prinzipiell unabhängig von der(den Visualisierung(en) der Sub-Komponente(n) des Schemas sein.

Erweiterbarkeit durch Spezialvisualisierungen für Objekttypen:
: Für geeignete Typen sollen spezielle Visualisierungen verwendet werden können, die nicht generisch, sondern auf diese Typen zugeschnitten sind. Dies ist insbesondere für Objekttypen sinnvoll.

Veränderung von Visualisierungsparametern:
: Die Darstellung einzelner Schemakomponenten soll, soweit möglich, individuell verändert werden können. Dies soll nicht nur durch den Einsatz von verschiedenen Darstellungen für die Schemakomponenten, sondern auch durch eine Parametrisierung dieser Darstellungen erreicht werden. Z. B. können für die textuelle Repräsentation eines atomaren Wertes die verschiedenen Eigenschaften der Schriftart parametrisiert werden.

Durch die Eigenschaften Flexibilität und Parametrisierbarkeit entsteht eine Aufteilung des Visualisierungsverfahrens in einen statischen und einen dynamischen Teil. Der statische Teil besteht im Wesentlichen aus der Definition von Datenstrukturen; der dynamische Teil umfaßt die Algorithmen, die durch diese Datenstrukturen gesteuert werden und auf ihnen operieren. Diese Aufteilung drückt sich auch in der Gliederung der vorliegenden Arbeit aus: Der Rest dieses Kapitels beschreibt den statischen Teil des Visualisierungsverfahrens; daraus wird dann in Kapitel 4 der dynamische Teil entwickelt.

3.3 Visualisierungsfunktionen

Die Individualisierung und Erweiterbarkeit führt dazu, für jeden der drei in Abschnitt 3.1.1 identifizierten Fälle eine Klasse von **Visualisierungsfunktionen** zu betrachten:

1. Atomare Visualisierungsfunktionen,
2. Kollektions-Visualisierungsfunktionen und
3. Tupel-Visualisierungsfunktionen.

Jede Klasse soll durch ihre Schnittstelle, die durch Methoden bzw. deren Signaturen definiert wird, einen Rahmen für die Definition der Visualisierungsfunktionen bilden. Letztere, die Visualisierungsfunktionen, sollen dann quasi frei wählbar und miteinander kombinierbar sein und so die Basis der verallgemeinerten Repräsentation von komplexen Objekten bilden.

Die Visualisierungsfunktionen dieser drei Klassen haben gemeinsam, daß sie **datenbezogen** sind, d. h. sie visualisieren Datenobjekte. Es gibt aber auch Visualisierungsfunktionen, die nicht datenbezogen sind. Z. B. können durch den Einsatz von Viewports mit Rollbalken Höhe (vertikaler Rollbalken) und/oder Breite (horizontaler Rollbalken) der Repräsentation eines Datenobjektes reduziert bzw. eine variable Höhe und/oder Länge fixiert werden, was unabhängig von dem zugrunde liegenden Datenobjekt ist. Solche **Konstruktionen** müssen als vierter Fall für die Klassifizierung aus Abschnitt 3.1.1 angesehen werden, so daß eine weitere Klasse von Visualisierungsfunktionen betrachtet wird:

4. Konstruktions-Visualisierungsfunktionen.

Bisher wurden nur **generische Visualisierungsfunktionen** betrachtet. Generische Visualisierungsfunktionen sind einem Konstruktor zugeordnet (bzw. atomar) und müssen die Fähigkeit besitzen, alle Datenobjekte, deren Typ mit diesem Konstruktor gebildet wurde (bzw. alle atomaren Datenobjekte), zu visualisieren.

Im Gegensatz dazu sollen **spezielle Visualisierungsfunktionen** konkreten Typen zugeordnet sein, insbesondere Objekttypen. Sie können (und im Allgemeinen werden sie) sich auf die Struktur dieser Typen beziehen. Ist z. B. ein Polygon durch ein Tupel modelliert, das als Komponenten die Rand- und Füllfarbe des Polygons sowie eine Liste der Eckpunktkoordinaten enthält, kann die spezielle Visualisierungsfunktion Instanzen dieses Typs durch eine graphische Darstellung des Polygons visualisieren.

Generische Repräsentationen komplexer Datenobjekte sind immer rekursiv durch die Repräsentationen von Sub-Objekten definiert. Kollektionen haben genau eine Sub-Struktur, die den Elementtyp beschreibt, d. h. ihre Elemente haben homogene Repräsentationen. Tupel haben i. d. R. mehr als eine Sub-Struktur, die die Komponententypen beschreiben, d. h. ihre Komponenten haben inhomogene Repräsentationen. Im Gegensatz dazu sind spezielle Repräsentationen im Allgemeinen nicht durch Repräsentationen von Sub-Objekten definiert. Obwohl sie für komplexe Strukturen definiert sein können, sollen Spezielle

Visualisierungsfunktionen daher der Klasse der atomaren Visualisierungsfunktionen zugeordnet werden. Später wird deutlich, daß spezielle Visualisierungsfunktionen, unabhängig von dieser Klassifizierung, trotzdem Repräsentationen von Sub-Strukturen in ihre Repräsentationen integrieren können (siehe Abschnitt 4.2.4).

Der Bezug generischer Visualisierungsfunktionen für Konstruktoren auf Sub-Strukturen läßt sich als eine erste Forderung an die Schnittstellen der Visualisierungsfunktionen für Kollektionen und Tupel formulieren: Sie sollen jeweils eine Methode namens Sub für die Integration der Element- bzw. Komponentenrepräsentationen umfassen.

3.3.1 Klassenhierarchie

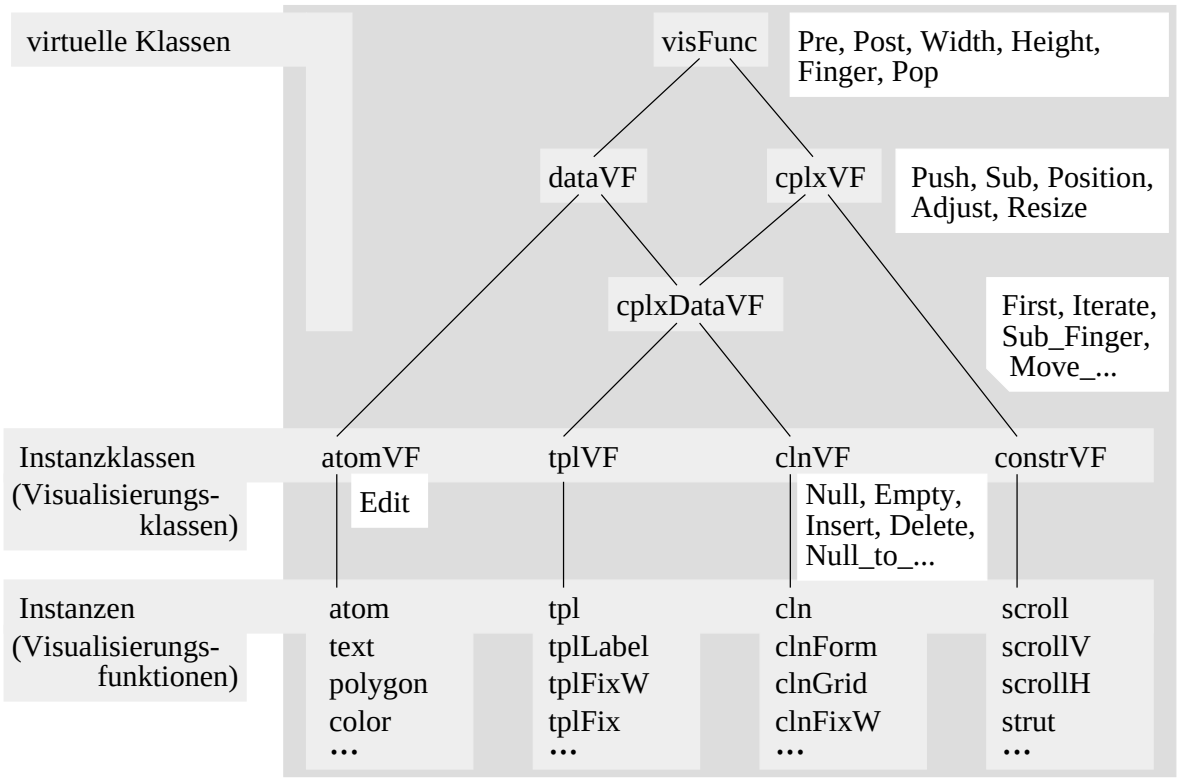


Abbildung 3.8: Klassenhierarchie für Visualisierungsfunktionen

Eine Klassenhierarchie im objekt-orientierten Sinn (siehe Abb. 3.8) hat als Wurzel eine virtuelle Klasse visFunc, die Struktur und Verhalten an alle Klassen vererbt. Sie definiert außerdem die Methoden, die alle Visualisierungsfunktionen implementieren müssen. Die Klasse visFunc hat zwei direkte Unterklassen, dataVF und cplxVF, die nicht disjunkt sind: In ihrem Schnitt liegt die Klasse cplxDataVF. dataVF vereint alle datenbezogenen, cplxVF alle komplexen, und cplxDataVF die datenbezogenen, komplexen Visualisierungsfunktionen. Die Klasse dataVF erweitert das Interface der Visualisierungsfunktionen nicht, wird aber

bei der Integration der Interaktion eine Rolle spielen. Die Klassen atomVF und cplxDataVF sind die disjunkte Aufteilung von dataVF. In atomVF sind alle atomaren Visualisierungsfunktionen enthalten. cplxVF erweitert die Schnittstellen von komplexen Visualisierungsfunktionen um weitere Methoden und wird in die zwei disjunkte Klassen cplxDataVF und constrVF aufgeteilt. Erstere definiert weitere, datenbezogene Methoden für komplexe Visualisierungsfunktionen, letztere enthält Konstruktions-Visualisierungsfunktionen. Die datenbezogenen, komplexen Visualisierungsfunktionen der Klasse cplxDataVF werden schließlich in die disjunkten Klassen tplVF und clnVF für Tupel- und Kollektions-Visualisierungsfunktionen aufgeteilt, so daß als Instanzklassen die vier **Visualisierungsklassen** atomVF, tplVF, clnVF und constrVF zur Verfügung stehen. Im mathematischen Sinn bilden diese vier Klassen vier endliche, disjunkte Mengen $\mathcal{F}_{\text{atom}}$, $\mathcal{F}_{\text{tpl}}$, $\mathcal{F}_{\text{cln}}$ und $\mathcal{F}_{\text{constr}}$, deren Elemente Visualisierungsfunktionen sind. Somit ist $\mathcal{F} = \mathcal{F}_{\text{atom}} \cup \mathcal{F}_{\text{tpl}} \cup \mathcal{F}_{\text{cln}} \cup \mathcal{F}_{\text{constr}}$ die Menge aller Visualisierungsfunktionen.

Die vier Visualisierungsklassen werden unter dem Begriff „Instanzklassen“ zusammengefaßt, da die einzelnen Visualisierungsfunktionen Instanzen dieser Visualisierungsklassen sind. Alle anderen Klassen (visFunc, dataVF, usw.) sind virtuelle Klassen, die nicht der Instantiierung dienen.

In Abb. 3.8 sind bereits einige Methoden-Namen enthalten: Pre, Post, First, Sub, Iterate, Push, Pop, usw. In Kapitel 4, bei der algorithmischen Formulierung des Visualisierungsverfahrens, wird die Notwendigkeit dieser Methoden erläutert. Außerdem werden weitere Methoden hinzukommen. In Kapitel 5 werden dann exemplarisch einige Visualisierungsfunktionen beschrieben. Dabei wird auch auf die konkrete Realisierung ihrer Methoden eingegangen.

3.3.2 Parametrisierung

Visualisierungsfunktionen, bzw. ihre Implementierungen, können mehr oder weniger allgemein gehalten werden. Je allgemeiner eine Visualisierungsfunktion ist, desto eher benötigt sie zur Konkretisierung eine Parametrisierung. Die Anwendung einer Visualisierungsfunktion muß dann durch Angabe von Parametern konkretisiert werden.

Beispiel 3.9 Parametrisierung von Visualisierungsfunktionen

- Für die Darstellung von Tupeln und Kollektionen durch horizontale bzw. vertikale Konkatenation kann die Dicke der senkrechten und vertikalen Trennlinien parametrisiert werden.

- Für die Darstellung von atomaren Werten durch ihre textuelle Repräsentation können z. B. Schriftart, Feldbreite und -höhe, ggf. relativ zur Schriftart, Ausrichtung im Feld, usw. durch Parameter gewählt werden.

- Wird eine Kollektion durch vertikale und horizontale Konkatenation dargestellt, muß durch die Parameter bestimmt werden, welche Dimension variabel ist, und wie viele Elemente in der anderen Dimension über- bzw. nebeneinander dargestellt werden.

- Wird eine Kollektion mit elementweiser Navigationsmöglichkeit dargestellt, kann durch Parameter bestimmt werden, welche Form diese Navigationsmöglichkeit annehmen soll, z. B. Vor- und Zurück-Knöpfe, ein Rollbalken oder Karteireiter. Außerdem kann festgelegt werden, ob die Navigation eine horizontale oder vertikale Richtung simuliert (d. h. z. B. für Knöpfe, ob sie als Hoch-/Runter-Paar oder als Links-/Rechts-Paar dargestellt werden). □

Für jede Visualisierungsfunktion muß definiert sein, durch welche Parameter sie konkretisiert werden kann, und für jede Anwendung einer Visualisierungsfunktion müssen die Parameter mit Werten besetzt werden. Außerdem soll jede Visualisierungsfunktion für ihre Parameter Default-Werte definieren (siehe Abschnitt 3.8.2). Damit müssen nur Parameter, die von den Default-Werten abweichen sollen, bei der Anwendung einer Visualisierungsfunktion spezifiziert werden. Wie das geschieht, wird Abschnitt 3.6.2 zeigen.

3.3.3 Bindungen für Eingabeereignisse

Die Interaktion mit Datenobjekten, bzw. deren Repräsentationen, kann als eine Umsetzung von Eingabeereignissen in Operationen auf den Datenobjekten angesehen werden. In Abschnitt 4.3 wird deutlich werden, daß es von einer Visualisierungsfunktion abhängt, welche Eingabeereignisse in welche Operationen umgesetzt werden. Ohne den konkreten Mechanismus dieser Umsetzung zu kennen, soll an dieser Stelle bereits gefordert werden, daß durch Visualisierungsfunktionen eine Bindung zwischen Eingabeereignissen und auszuführenden Operationen definiert wird. Ähnlich wie bei den Parametern soll jede Visualisierungsfunktion Default-Bindungen definieren.

Da sich die Interaktion immer auf Datenobjekte beziehen soll, werden Bindungen nur von datenbezogenen Visualisierungsfunktionen definiert, also nicht von Konstruktions-Visualisierungsfunktionen. Das heißt nicht, daß in einer generierten Repräsentation keine Bindungen außer den durch die Visualisierungsfunktionen definierten existieren. Vielmehr ist es auch notwendig, andere sensitive Elemente von generierten Repräsentationen mit entsprechenden Bindungen zu versehen, z. B. Rollbalken. Diese stehen aber nicht mit der Manipulation der Datenobjekte in Verbindung und werden daher nicht in Operationen auf den Datenobjekten umgesetzt.

Komplexe Visualisierungsfunktionen müssen auch Bindungen für Repräsentationen von Sub-Objekten definieren, da z. B. bei der Navigation mit den *move*-Operationen der Finger auf ein Sub-Objekt des komplexen Objektes zeigt, in dem sich bewegt wird. Die Repräsentationen des Sub-Objektes „weiß“ aber in diesem Fall nicht, ob durch die ⏎-Taste

oder die ⊡-Taste auf das nächste Sub-Objekt navigiert wird — das ist von der umgebenden Repräsentation, d. h. von der komplexen Visualisierungsfunktion und ggf. von deren Parametrisierung abhängig.

Um ein ergonomisches Arbeiten zu ermöglichen ist es notwendig, daß alle Operationen ohne Maus ausgelöst werden können, d. h. die Operationen müssen an Tastaturereignisse gebunden sein [UGB93]. Für Visualisierungsfunktionen wird hier sogar nur die Interaktion ohne Maus durch Bindungen zwischen Tastaturereignissen und Operationen definiert. Aus diesen Bindungsdefinitionen werden aber auch Pop-Up-Menüs generiert, mit denen eine mausbasierte Interaktion möglich ist.

3.4 Visualisierungsbaum

Für alle Komponenten eines Schemas soll die Möglichkeit einer individuellen Darstellung bestehen (siehe Abschnitt 3.1). Für ein Schema als Baum betrachtet bedeutet das, daß für jeden Knoten des Schemabaumes eine individuelle Darstellung ausgewählt werden kann. Man könnte also das Schema erweitern, indem die Knoten des Schemabaumes mit Anmerkungen versehen werden, die ihre Darstellung beschreiben. Das würde aber bedeuten, daß für jedes Schema nur genau eine Repräsentation existieren kann. Diese Einschränkung ist weder erwünscht noch notwendig. Vielmehr kann es zu jedem Schema mehrere Repräsentationen geben, die jeweils durch einen aus dem Schemabaum abgeleiteten, individuellen **Visualisierungsbaum** bestimmt sind.

Ein Visualisierungsbaum muß nicht genau die gleiche Struktur wie der zugehörige Schemabaum haben. Der Begriff „gleiche Struktur“ wird hier im Sinne von „Anzahl der Knoten“, „Anzahl der Nachfolger eines Knotens“ und „Reihenfolge der Nachfolger eines Knotens“ verwendet. In Abschnitt 3.6.2 werden als Operationen auf Visualisierungsbäumen das Löschen von Knoten bzw. Teilbäumen, das Ändern der Reihenfolge der Nachfolger eines Knotens und das Einfügen von Knoten bzw. Teilbäumen betrachtet.

Die Knoten eines Visualisierungsbaumes heißen **Visualisierungsknoten**. Jeder Visualisierungsknoten beschreibt die individuelle Repräsentation einer Schemakomponente. Dazu muß er, gemäß den obigen Anforderungen, festlegen, wie die Komponente dargestellt und mit welchen Parametern diese Darstellung realisiert werden soll.

Formal wird ein Visualisierungsbaum wie folgt definiert: Seien $\mathcal{V}_{\text{cln}}$, $\mathcal{V}_{\text{tpl}}$, $\mathcal{V}_{\text{constr}}$ und $\mathcal{V}_{\text{atom}}$ abzählbar unendliche, disjunkte Mengen von Kollektions-, Tupel-, Konstruktions- und atomaren Knoten. Dann ist $\mathcal{V} = \mathcal{V}_{\text{cln}} \cup \mathcal{V}_{\text{tpl}} \cup \mathcal{V}_{\text{constr}} \cup \mathcal{V}_{\text{atom}}$ ein abzählbar unendlicher Vorrat von Knoten. Knoten sollen n-Tupel sein, wobei hier zunächst weder n noch die Komponenten der Tupel spezifizieren werden sollen.

Ein Visualisierungsbaum $T = (V, E, r)$ ist ein gerichteter, azyklischer Graph (DAG), der durch eine endliche Knoten-Menge $V \subset \mathcal{V}$, die Kanten-Relation $E \subset V \times V$, und die

Wurzel $r \in V$ definiert ist. Für E gelten, zusätzlich zu den DAG-Bedingungen, folgende strukturellen Einschränkungen:

- $\forall v_2 \in V, v_2 \neq r : \exists v_1 \in V : (v_1, v_2) \in E \wedge \forall v_3 \in V : (v_3, v_2) \in E \Rightarrow v_1 = v_3$, d. h. jeder Knoten, bis auf die Wurzel, hat genau einen Vorgänger.
- $v \in \mathcal{V}_{\text{constr}} \cup \mathcal{V}_{\text{cln}} \wedge (v, v_1) \in E \wedge (v, v_2) \in E \Rightarrow v_1 = v_2$, d. h. Konstruktions- und Kollektions-Knoten haben höchstens einen Nachfolgerknoten.
- $v \in \mathcal{V}_{\text{constr}} \cup \mathcal{V}_{\text{cln}} \cup \mathcal{V}_{\text{tpl}} \Rightarrow \exists v' \in V : (v, v') \in E$, d. h. Konstruktions-, Kollektions- und Tupel-Knoten haben mindestens einen Nachfolgerknoten.
- $v \in \mathcal{V}_{\text{atom}} \Rightarrow \forall v' \in V : (v', v) \notin E$, d. h. atomare Knoten haben keinen Nachfolgerknoten.

Die erste Komponente eines Knotens ist eine Visualisierungsfunktion, d. h. $v = (f, \cdot)$[10] mit $f \in \mathcal{F}$. Die Abbildung visFunc bildet einen Visualisierungsknoten $v = (f, \cdot)$ auf seine Visualisierungsfunktion ab:

$$\begin{aligned} \text{visFunc} : \mathcal{V} &\rightarrow \mathcal{F} \\ v &\mapsto \text{visFunc}(v) := f \end{aligned}$$

Die Zuordnung zwischen Visualisierungsknoten v und Visualisierungsfunktionen f wird während der Konstruktion eines Visualisierungsbaumes definiert, wobei Kollektions-Knoten Kollektions-Visualisierungsfunktionen zugeordnet werden, usw., d. h.:

- $\forall v \in \mathcal{V}_{\text{cln}} : \text{visFunc}(v) \in \mathcal{F}_{\text{cln}}$
- $\forall v \in \mathcal{V}_{\text{tpl}} : \text{visFunc}(v) \in \mathcal{F}_{\text{tpl}}$
- $\forall v \in \mathcal{V}_{\text{constr}} : \text{visFunc}(v) \in \mathcal{F}_{\text{constr}}$
- $\forall v \in \mathcal{V}_{\text{atom}} : \text{visFunc}(v) \in \mathcal{F}_{\text{atom}}$

Die Visualisierungsfunktion eines Nachfolgerknotens im Visualisierungsbaum wird auch **Sub-Visualisierungsfunktion** genannt.

Die zweite und dritte Komponente eines Knotens sind eine Struktur und ein Attribut, d. h. $v = (f, s, a, \cdot)$ mit $s \in \mathcal{S}(\text{TYPES})$ und $a \in \mathcal{A}$ (siehe Abschnitt 2.3.1). Sie stellen also die Verbindung zwischen einem Visualisierungsbaum und einem Schema her. Die Abbildungen

[10] Da bisher weder Anzahl noch Art der Komponenten eines Knotens festgelegt sind, wird hier und im Folgenden ein Punkt $\cdot$ als Platzhalter benutzt, der durch ein beliebiges Tupel substituiert werden kann.

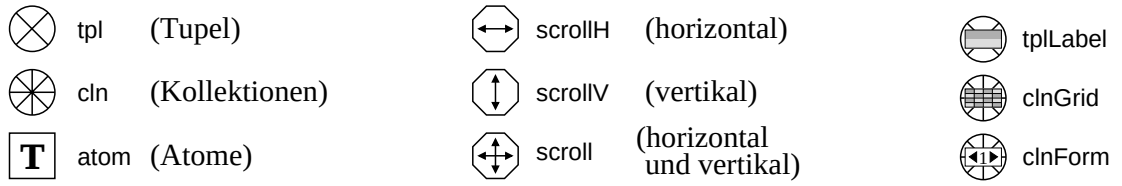
struk und attr bilden einen Visualisierungsknoten $v = (f, s, a, \cdot)$ auf seine Struktur bzw. sein Attribut ab:

$$\begin{aligned} \text{struk} : \mathcal{V} &\to \mathcal{S}(\text{TYPES}) \\ v &\mapsto \text{struk}(v) := s \\ \text{attr} : \mathcal{V} &\to \mathcal{A} \\ v &\mapsto \text{attr}(v) := a \end{aligned}$$

Die Zuordnung zwischen Visualisierungsknoten v, Strukturen s und Attributen a wird wieder während der Konstruktion eines Visualisierungsbaumes definiert, wobei folgende Einschränkungen gelten:

- $v \in \mathcal{V}_{\text{constr}}$, $(v, v') \in E \Rightarrow \text{struk}(v) = \text{struk}(v')$, $\text{attr}(v) = \text{attr}(v')$, d. h. Konstruktions-Knoten sind das Attribut und die Struktur ihres Nachfolgerknotens zugeordnet.
- $v \in \mathcal{V}_{\text{cln}}$, $(v, v') \in E \Rightarrow \text{struk}(v) = \text{c}_{\text{cln}}(a, s)$, $\text{struk}(v') = s$, $\text{attr}(v') = a$, d. h. jedem Kollektions-Knoten ist eine Kollektions-Struktur zugeordnet und dem Nachfolgerknoten eines Kollektions-Knotens ist die Elementstruktur zugeordnet.
- $v \in \mathcal{V}_{\text{atom}} \Rightarrow \text{struk}(v) \in \text{SIMPLETYPES}$, d h. jedem atomaren Knoten ist eine einfache Struktur zugeordnet (mit SIMPLETYPES wird die Menge S_0 aus [The96, Definition 3.9, S. 83 f.], ohne Varianten, bezeichnen).
- $v \in \mathcal{V}_{\text{tpl}}, (v, v') \in E \Rightarrow \text{struk}(v) = \text{c}_{\text{tuple}}(\ldots, (a', s'), \ldots), \text{struk}(v') = s', \text{attr}(v') = a'$, d. h. jedem Tupel-Knoten ist eine Tupel-Struktur zugeordnet und jedem Nachfolgerknoten des Tupel-Knotens sind Komponentenstrukturen zugeordnet.

Diese Zuordnung über die Strukturen aus [The96] ist konform mit den oben formulierten strukturellen Einschränkungen.



(a) Default (b) Konstruktionen (c) andere

Abbildung 3.9: Symbole für Visualisierungsfunktionen

Visualisierungsbäume werden graphisch analog zu Schemabäumen dargestellt, wobei die Visualisierungsknoten durch die Visualisierungsfunktionen repräsentierende Symbole markiert sind. Einige Symbole für Visualisierungsfunktionen sind in Abb. 3.9 auf der vorherigen Seite dargestellt. Die Symbole der Default-Visualisierungsfunktionen[11] in Abb. 3.9(a) für Tupel und Kollektionen stimmen mit den entsprechenden Symbolen der Konstruktoren überein. Dies mag zwar zunächst verwirrend sein, kann aber nicht zu Verwechslungen führen, da Schemabäume und Visualisierungsbäume immer getrennt dargestellt werden. Die Konstruktions-Visualisierungsfunktionen in Abb. 3.9(b) sind Viewports mit Rollbalken.

In den Abbildungen auf den folgenden Seiten sind Beispiele von Visualisierungen und durch sie definierte Repräsentationen dargestellt. Die Bedeutung der in den Beispielen verwendeten Begriffe „Default-Visualisierung“ und „verfeinerte Visualisierung“ wird in den Abschnitten 3.6.1 und 3.6.2 besprochen. Außerdem werden für einige Visualisierungsknoten Parameter angegeben, deren genaue Bedeutung in Kapitel 5 in den jeweiligen Abschnitten der Visualisierungsfunktionen erläutert werden.

Die Anwendung einer Visualisierungsfunktion, also deren Eintragung in einem Visualisierungsbaum, kann durch Angabe von Parametern und Bindungen konkretisiert werden (siehe Abschnitte 3.3.2 und 3.3.3). Realisiert wird dieses, indem die Visualisierungsknoten um zwei Komponenten erweitert werden, die Mengen sind.

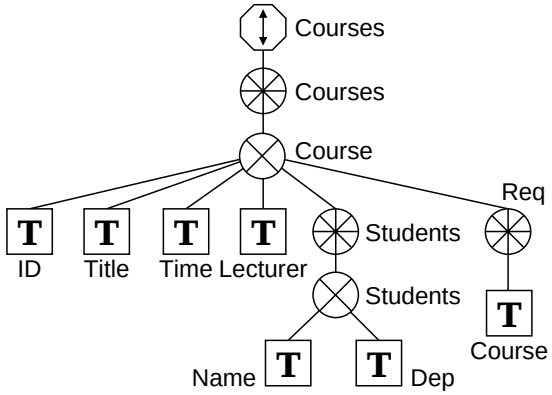
Die erste Menge enthält Name-Wert-Paare von Parametern, die von den Default-Werten abweichen, die zweite Menge enthält Tripel aus Spezifikationen von Tastaturereignissen, auszuführenden Operationen und einem Vererbungs-Flag. Die Bedeutung dieser drei Komponenten wird erst später konkretisiert (siehe Abschnitt 4.3). Ein Visualisierungsknoten ist also ein Tupel der Form $v = (f, s, a, P, B, \cdot)$ mit einer Menge P von Name-Wert-Paaren (*name*, *value*) und einer Menge B von Bindungs-Spezifikationen (*event*, *operation*, *inherit*).

Die Generierung einer Repräsentation wird sowohl vom Visualisierungsbaum als auch vom Datenobjekt gesteuert: Für Tupel definiert der Visualisierungsbaum, welche Komponenten in welcher Reihenfolge darzustellen sind, für Kollektionen definieren Datenobjekt und Visualisierungsfunktion die Anzahl und Reihenfolge der Elemente.

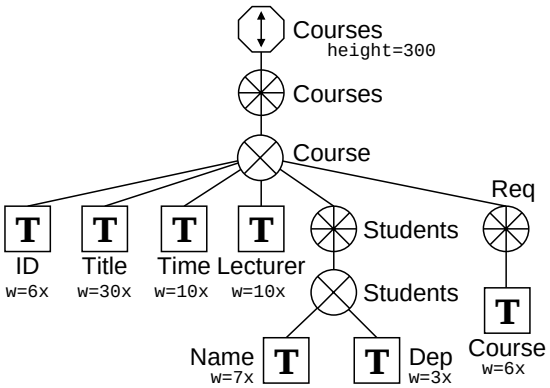
Ein Visualisierungsbaum ist die Definition einer Repräsentation, d. h. er entspricht dem Begriff der Visualisierung als „Festlegung, wie Datenobjekte visuell repräsentiert werden“. Wenn keine Verwechslungen möglich sind, sollen Visualisierungsbäume daher im Folgenden mit dem kürzeren Begriff „**Visualisierung**“ bezeichnet werden.

Visualisierungsknoten werden, wie auch die Visualisierungsfunktionen, objekt-orientiert beschrieben, d. h. die anwendbaren Operationen werden durch Methoden und deren Signaturen einer Klasse visNode formuliert. Bei der Beschreibung der Methoden der Klasse visNode wird detailliert vorgegangen, da sie u. a. die Generierung der Repräsentationen steuern. Die Beschreibung der Methoden von Visualisierungsfunktionen ist dagegen zunächst grob, da sie implementationsabhängig und für die verschiedenen Visualisierungsfunktionen unterschiedlich ist. Sie werden dann in Kapitel 5 konkretisiert.

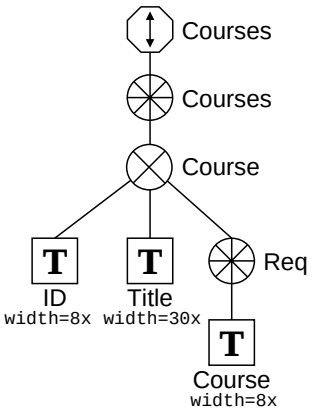
[11] Die Bedeutung von Default-Visualisierungsfunktionen wird in Abschnitt 3.6.1 erläutert.



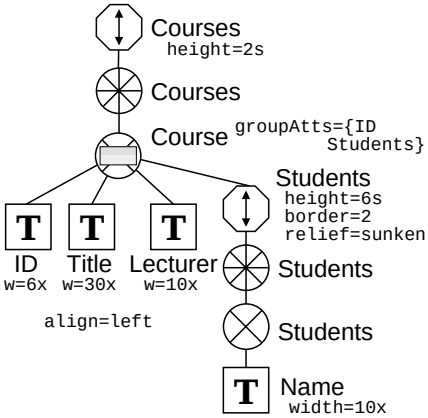
(a) Default-Visualisierung



(b) angepaßte Attributbreiten



(c) Vorlesungsvoraussetzungen

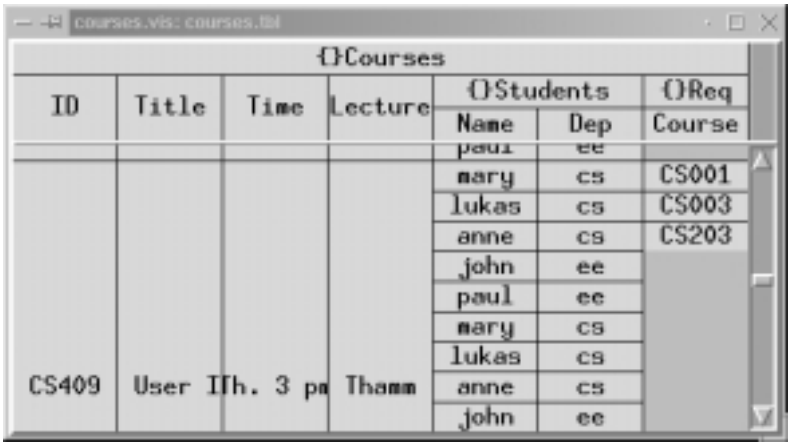


(d) teilnehmende Studenten

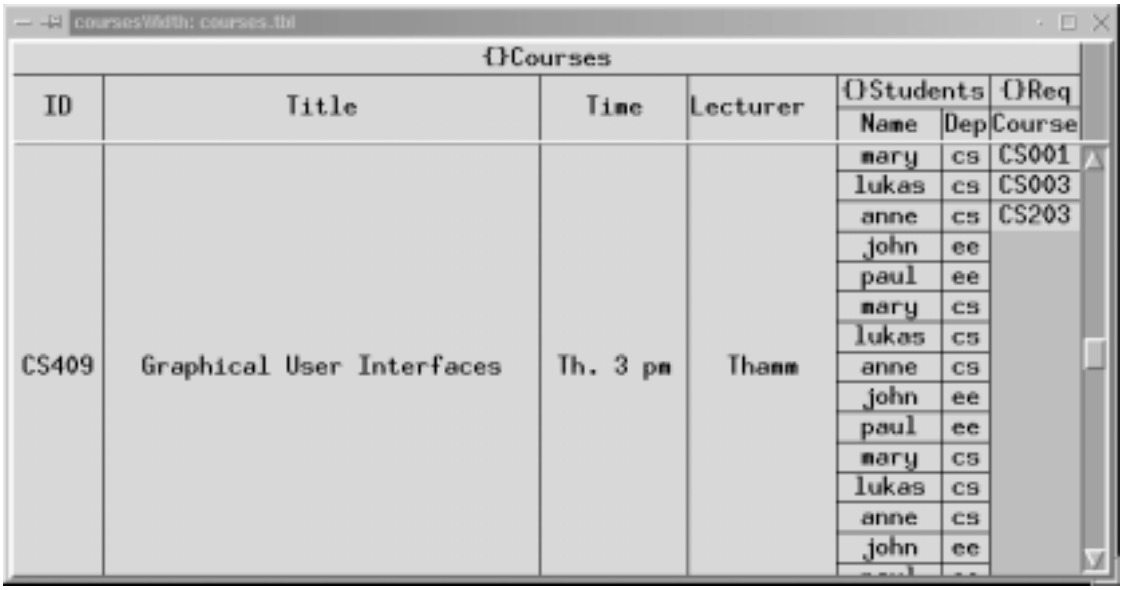
Abbildung 3.10: Visualisierungen für das Schema Courses

Beispiel 3.10 Visualisierungen zum Schema Courses

Die Abbildung zeigt aus dem Schema Courses (siehe Beispiel 2.1 auf Seite 46) abgeleitete Visualisierungen. Abb. 3.10(a) zeigt die Default-Visualisierung, Abb. 3.10(b) eine Visualisierung mit angepaßten Attributbreiten (alle atomaren Attribute werden per Default mit einer Breite von sieben Zeichen dargestellt; `w` ist eine Abkürzung für `width`) und einem etwas größeren Viewport. In Abb. 3.10(c) wurden im Tupel Course die drei Komponenten Time, Lecturer und Students ausgeblendet, so daß nur Vorlesungsnummer und -titel sowie die vorausgesetzten Vorlesungen dargestellt werden. In Abb. 3.10(d) wurde für Course die Tupel-Visualisierungsfunktion tplLabel eingetragen; die Komponenten werden links ausgerichtet. Die Studenten werden ohne Fachbereich in einem Viewport dargestellt; Time und Req wurden ausgeblendet. □



(a) Default-Repräsentation

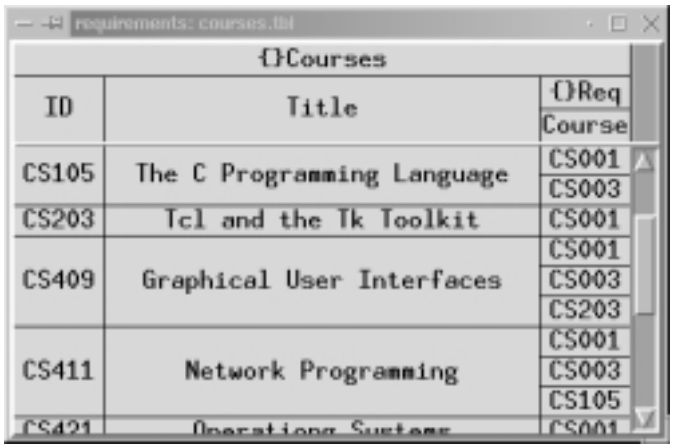


(b) Repräsentation mit angepaßten Attributbreiten

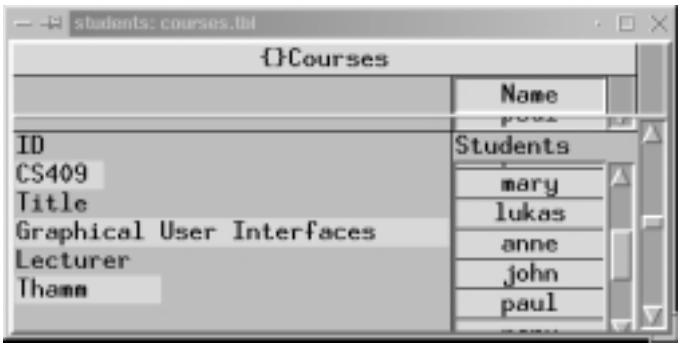
Abbildung 3.11: Repräsentationen der Vorlesungstabelle (1)

Beispiel 3.11 Repräsentationen der Vorlesungstabelle (1)

Die Abbildung zeigt zwei Repräsentationen der Vorlesungstabelle, die mit den Visualisierungen aus Abb 3.10(a) und 3.10(b) auf der vorherigen Seite erzeugt wurden. Deutlich sichtbar sind die angepaßten Attributbreiten in Abb. 3.11(b) und der dort gegenüber Abb. 3.11(a) höhere Viewport. □



(a) Vorlesungsvoraussetzungen

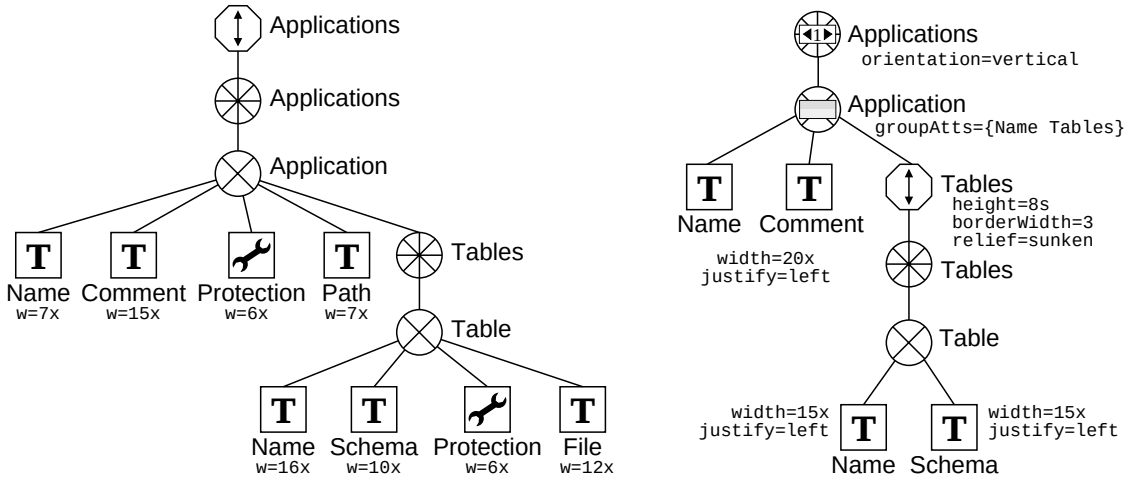


(b) teilnehmende Studenten

Abbildung 3.12: Repräsentationen der Vorlesungstabelle (2)

Beispiel 3.12 Repräsentationen der Vorlesungstabelle (2)

Die Abbildung zeigt zwei Repräsentationen der Vorlesungstabelle, die mit den Visualisierungen aus Abb. 3.10(c) und 3.10(d) auf Seite 87 erzeugt wurden. In Abb 3.12(a) sind die Tupel variabel hoch, aber durch das Entfernen der „großen“ Studentenmengen überschaubar geworden. In Abb 3.12(b) sind die Studentenmengen in einem Viewport dargestellt, so daß die Course-Tupel eine feste Höhe haben und ebenfalls überschaubar geworden sind. Die vertikale Konkatenation der Komponenten ID, Title und Lecturer durch die Visualisierungsfunktion tplLabel reduziert die Visualisierungsbreite der Tupel; durch die Darstellung der Attributnamen als Label über den Komponenten können diese im Schema fehlen. □



(a) mit angepaßten Attributbreiten

(b) MetaSelect

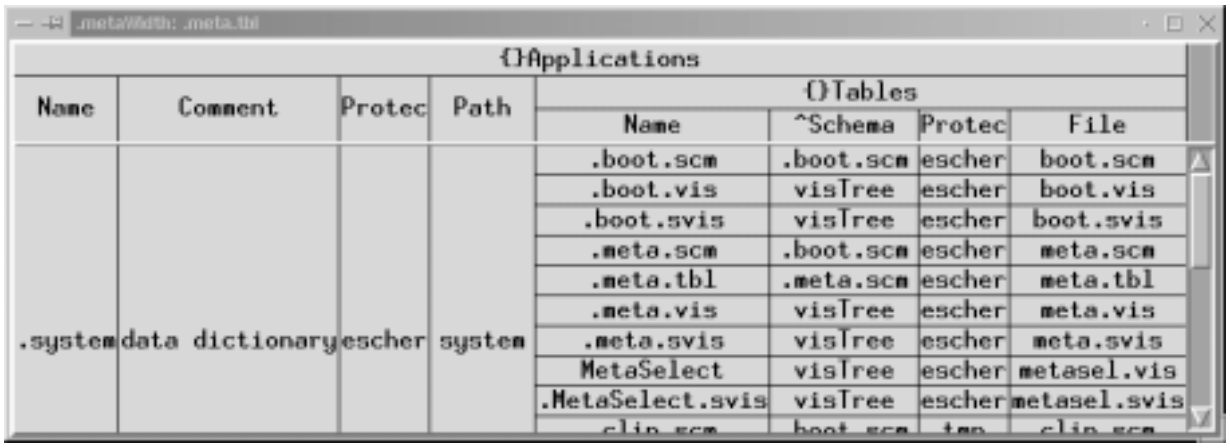
Abbildung 3.13: Visualisierungen für das Schema Applications

Beispiel 3.13 Visualisierungen zum Schema Applications

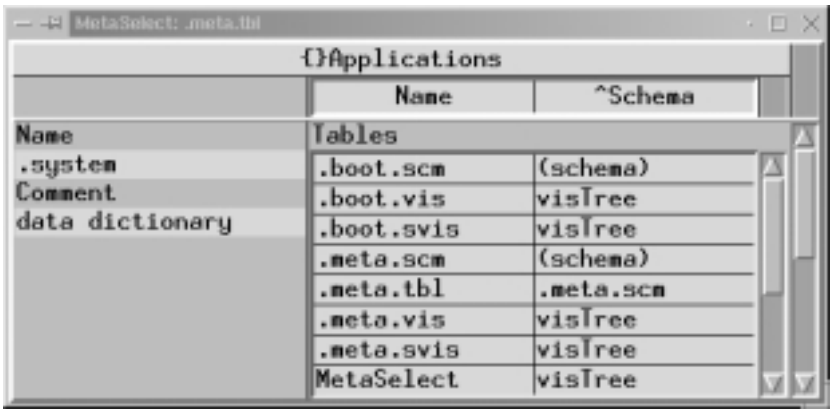
Die Abbildung zeigt zwei Visualisierungen, die aus dem Schema Applications (siehe Beispiel 2.2 auf Seite 47) abgeleitet wurden. Abb. 3.13(a) zeigt eine Visualisierung, die aus der Default-Visualisierung durch Anpassung der Attributbreiten abgeleitet wurde. Abb. 3.13(b) zeigt eine verfeinerte Visualisierung, die gegenüber der Default-Visualisierung wie folgt verändert wurde:

- Konstruktions-Visualisierungsfunktion scrollV eingefügt (vertikal rollbarer Viewport um die Menge Tables),
- Kollektions-Visualisierungsfunktion clnForm für die Menge Applications,
- Tupel-Visualisierungsfunktion tplLabel für das Tupel Application,
- in den Tupeln jeweils zwei Komponenten ausgeblendet (Protection und Path in Application, Protection und File in Table), und
- weitere Parameter an einigen Visualisierungsknoten spezifiziert.

Diese Visualisierung wird in Abschnitt 6.1.1 Grundlage für eine Anwendung des Visualisierungsverfahrens auf ESCHERs Data Dictionary sein und heißt MetaSelect. Mit diesen Visualisierungen generierte Repräsentationen sind in Abb. 3.14(b) auf der gegenüberliegenden Seite dargestellt. □



(a) Repräsentation mit angepaßten Attributbreiten



(b) Repräsentation MetaSelect

Abbildung 3.14: Repräsentationen der Meta-Tabelle Applications

Beispiel 3.14 Repräsentationen der Meta-Tabelle Applications

Die Abbildung zeigt zwei Repräsentationen der Meta-Tabelle Applications, die mit den Visualisierungen aus Abb. 3.13(a) und 3.13(b) auf der gegenüberliegenden Seite erzeugt wurden. □

Bei der *detaillierten* Beschreibung einer Methode wird ihre Berechnung durch einen Algorithmus angegeben, der prinzipiell auch als Anleitung zur Implementierung geeignet ist; eine *grobe* Beschreibung einer Methode umfaßt „nur“ eine Darstellung der Aufgabe, die die Methode erfüllt[12].

3.5 Breiten und Höhen

Ein grundlegendes Problem der visuellen Repräsentation komplexer Datenobjekte sind variable Längen[13], die sich z. B. durch Einfügen und Löschen von Sub-Objekten dynamisch, d. h. zur Laufzeit jederzeit ändern können. Da Repräsentationen komplexer Objekte rekursiv durch die Repräsentationen der Sub-Objekte definiert sind, kann nur aus deren Eigenschaften abgeleitet werden, wie sie sich bzgl. variabler Längen verhalten. Auf Visualisierungsfunktionen übertragen bedeutet das, daß komplexe Visualisierungsfunktionen nur aus den Eigenschaften der Sub-Visualisierungsfunktion(en) ihre Eigenschaften bestimmen können.

3.5.1 Längeneigenschaften und -anforderungen

Eine Visualisierungsfunktionen kann folgende **Längeneigenschaften** haben[14]:

- Die Länge ist fest (nicht veränderbar),
- die Länge ist (von außen) anpaßbar oder
- die Länge ist variabel (von innen veränderbar).

Feste, nicht veränderbare Längen bedürfen keiner weiteren Erläuterung. Diese Längeneigenschaft hat z. B. eine atomare Visualisierungsfunktion, die textuelle Repräsentationen von atomaren Werten in einem Rechteck fester Größe darstellt.

Eine Länge ist variabel, wenn sie von Repräsentationen von Sub-Objekten abhängt. Sie ist in dem Sinne „von innen“ veränderbar, daß sich Längenänderungen der „inneren“ Repräsentationen auf die Länge der „äußeren“ Repräsentation auswirken können. Ein einfaches Beispiel ist die vertikal konkatenierte Darstellung von Kollektionen, deren Höhe direkt von der Anzahl und den Visualisierungshöhen der Elemente abhängt.

Bleiben noch die von außen anpaßbar Längen. Diese Eigenschaft soll am folgenden Beispiel erläutert werden:

[12] Diese Unterscheidung drückt sich auch in den weiter unten verwendeten Marken „Berechnung:“ und „Aufgabe:“ bei der detaillierten bzw. groben Beschreibung von Methoden aus.

[13] Wenn im Folgenden von einer „Länge“ die gesprochen wird, ist damit eine Breite und/oder eine Höhe gemeint.

[14] Eigentlich sind hier die Längeneigenschaften der von einer Visualisierungsfunktion generierten Repräsentation gemeint.

Beispiel 3.15 Von außen anpaßbare Längen

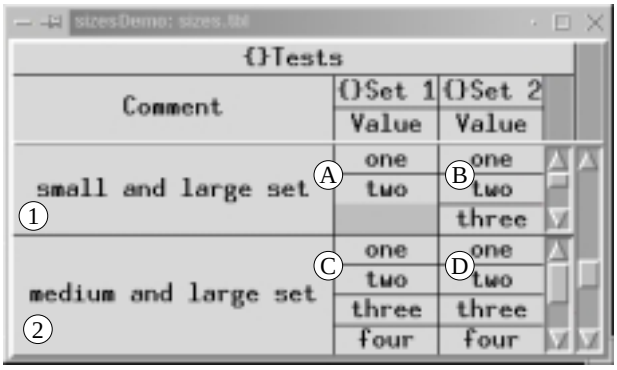


Abbildung 3.15: Von außen anpaßbare Längen

In Abb. 3.15 ist eine Kollektion dargestellt, die ihre Elemente vertikal Konkateniert. Die Elemente (Tupel ① und ②) haben neben einem String zwei weitere Kollektionen als Komponenten (Ⓐ und Ⓑ bzw. Ⓒ und Ⓓ), die ihre Elemente ebenfalls horizontal konkatenieren. Jeweils eine Menge (Ⓑ und Ⓓ) wird zusätzlich von einer Konstruktions-Visualisierungsfunktion in einem vertikal rollbaren Viewport dargestellt.

Die Höhe der Konstruktions-Visualisierungsfunktion ist von außen anpaßbar, d. h. sie paßt die Höhe der Viewports der äußeren Höhe, nämlich der Höhe des umgebenden Tupels, an. In Tupel ① ist die Höhe der Kollektion Ⓐ kleiner als die des Viewports. Letzterer bestimmt daher die Höhe des Tupels. In Tupel ② ist die Höhe der Kollektion Ⓒ dagegen größer als die des Viewports. Ⓒ bestimmt daher die Höhe des Tupels und die Höhe des Viewports, in dem die Kollektion Ⓓ dargestellt ist, wird größer. □

Um die Längeneigenschaften zu erfüllen, können Visualisierungsfunktionen **Längenanforderungen** an ihre Sub-Visualisierungsfunktionen stellen. Z. B. muß eine Tupel-Visualisierungsfunktion, die Komponenten horizontal konkateniert, und eine feste und nicht veränderbare Breite als Eigenschaft erfüllen will, von ihren Sub-Visualisierungsfunktionen fordern, daß ihre Breiten ebenfalls fest sind.

Im Folgenden sollen für einige Visualisierungsfunktionen beispielhaft die Längeneigenschaften und -anforderungen bestimmt werden. In Kapitel 5 wird bei der Beschreibung der Visualisierungsfunktionen ebenfalls auf deren Längeneigenschaften bzw. -anforderungen eingegangen.

cln (siehe Abschnitt 5.8.1) stellt Kollektionen durch vertikale (horizontale) Konkatenation der Elemente dar. Die Breite (Höhe) wird durch die Element-Visualisierungsfunktion bestimmt. Hat die Element-Visualisierungsfunktion eine feste Breite (Höhe), ist die Visualisierungsbreite (-höhe) von cln ebenfalls fest. Die Höhe (Breite) ist variabel und

hängt von der Kardinalität der Kollektion und der Höheneigenschaft (Breiteneigenschaft) der Element-Visualisierungsfunktion ab.

clnGrid stellt Kollektionen durch horizontale und vertikale Konkatenation der Elementen in einer matrixähnlichen Anordnung dar. Zwei Repräsentationen sind schematisch in Abb. 3.16 dargestellt. Die Element-Visualisierungsfunktion muß feste Breite und Höhe haben. Je nach Parametrisierung hängt entweder die Breite (Abb. 3.16(a)) oder die Höhe (Abb. 3.16(b)) von der Kardinalität der Kollektion ab. Die entsprechend andere Länge von clnGrid ist fest.

(a) horizontal variabel (b) vertikal variabel

Abbildung 3.16: Horizontal und vertikal konkatenierte Kollektionen

clnForm (siehe Abschnitt 5.8.2) stellt Kollektionen elementweise in dar. Die Element-Visualisierungsfunktion muß feste Breite und Höhe haben, sowohl Breite als auch Höhe von clnForm sind fest.

tpl (siehe Abschnitt 5.7.1) stellt Tupel durch horizontale oder vertikale Konkatenation der Komponenten dar. Höhe und Breite werden durch die Komponenten-Visualisierungsfunktionen bestimmt. Nur wenn alle Komponenten-Visualisierungsfunktionen feste Breite und/oder Höhe haben, ist die Breite bzw. Höhe von tpl fest.

scrollH, scrollV und scroll (siehe Abschnitte 5.5.1 und 5.5.2) stellt Repräsentationen von Datenobjekten in Viewports mit horizontalem und/oder vertikalem Rollbalken dar. Ist eine Dimension ohne Rollbalken (z.B. die Horizontale), muß die entsprechende Länge der Sub-Visualisierungsfunktion fest sein (im Beispiel die Breite). Dimensionen mit Rollbalken (im Beispiel die Vertikale), bzw. die entsprechenden Längen (im Beispiel die Höhe), sind von außen anpaßbar.

polygon (siehe Abschnitt 5.3.3) stellt Polygone graphisch dar. Sie ist eine spezielle Visualisierungsfunktion, gehört damit zu den atomaren Visualisierungsfunktion und hat

daher keine Sub-Visualisierungsfunktion, an die Längenanforderungen gestellt werden. Länge und Breite eines Polygons bzw. dessen umgebener *bounding box* sind abhängig von den Koordinaten der Eckpunkte. Sind die Längen von polygon durch die *bounding box* des Polygons bestimmt, sind Breite und Höhe variabel. Die Breite und/oder Höhe kann jedoch auch begrenzt werden, indem die Darstellung des Polygons skaliert, am Rand abgeschnitten, oder in einem rollbaren Viewport dargestellt wird.

Für alle Visualisierungsfunktionen müssen ihre Längeneigenschaften, für komplexe Visualisierungsfunktionen müssen zusätzlich die Längenanforderungen an Sub-Visualisierungsfunktionen definiert sein (siehe Abschnitt 3.8).

3.5.2 Längenberechnung

Die Längen einer Repräsentation hängen von mehreren Faktoren ab:

- Vom Datenobjekt,
- von den gewählten Visualisierungsfunktionen und
- von deren Parametrisierung.

Einige Visualisierungsfunktionen können die Breite und/oder Höhe der durch sie generierten Repräsentationen unabhängig vom Datenobjekt, also bereits vor der Generierung der Repräsentation, bestimmen. Dieses wird ausgenutzt, um die Visualisierung, soweit möglich, mit Längeninformationen anzureichern, die dann bei der Generierung der Repräsentation benutzt werden können. Dazu werden zwei weitere Komponenten für Visualisierungsknoten eingeführt, die somit Tupel der Form $v = (f, s, a, P, B, w, h)$ mit einer Breite w und einer Höhe h sind.

Die Schnittstelle der Visualisierungsklasse visFunc wird um die Methoden Width und Height erweitert. Aufgabe dieser Methoden ist, die Breite bzw. Höhe der durch einen Visualisierungsknoten (bzw. dessen Teilbaum) definierten Repräsentation, dem eine Visualisierungsfunktion zugeordnet ist, zu bestimmen. Sie tun dies unabhängig von einem Datenobjekt und können sich falls notwendig auf bereits berechneten Längen von Nachfolgerknoten beziehen.

Wie sich variable Längen von Nachfolgerknoten nach außen fortpflanzen, hängt von der Visualisierungsfunktion ab. Enthält z. B. ein Tupel (horizontal konkateniert) eine Menge (vertikal konkateniert) als Komponente, pflanzt sich die variable Höhe der Kollektion nach außen fort, d. h. auch das Tupel hat eine variable Höhe. Die Variabilität einer Länge wird durch den Rückgabewert Null der entsprechenden Methode signalisiert.

Methode Width

Klasse:
visFunc
Objekt:
Visualisierungsfunktion f
Parameter:
Visualisierungsknoten v
Aufgabe:
Berechne die (datenunabhängige) Breite der Repräsentation von v. Benutze dazu, falls notwendig, die bereits berechneten Breiten der Nachfolgerknoten von v.
Rückgabe:
Breite der Repräsentation von v, falls diese unabhängig vom Datenobjekt ist, Null sonst.

Methode Height

Klasse:
visFunc
Objekt:
Visualisierungsfunktion f
Parameter:
Visualisierungsknoten v
Aufgabe:
Berechne die (datenunabhängige) Höhe der Repräsentation von v. Benutze dazu, falls notwendig, die bereits berechneten Höhen der Nachfolgerknoten von v.
Rückgabe:
Höhe der Repräsentation von v, falls diese unabhängig vom Datenobjekt ist, Null sonst.

Die Anreicherung mit Größeninformationen wird in einem rekursiven Verfahren durchgeführt, das die Visualisierung *depth-first* durchläuft. So werden zunächst die Width- und Height-Methoden der atomaren Visualisierungsfunktionen an den Blättern aufgerufen und die berechneten Längen dort eingetragen, bevor die Methoden der inneren, komplexen Knoten sich darauf beziehen können.

Die Anforderungen der Visualisierungsfunktionen bzgl. der Längen ihrer Sub-Visualisierungsfunktionen müssen erfüllt sein, wodurch es bei dieser Längenberechnung zu Inkonsistenzen kommen kann (siehe Beispiel 3.16 auf Seite 101). Die Längenberechnung wird unabhängig von einem Datenobjekt durchgeführt; sie ist daher für alle Datenobjekte gültig und bleibt auch für sich ändernde Datenobjekte gültig. Da sie vor der Generierung einer Repräsentation ausgeführt wird, haben erkannte Inkonsistenzen keine Auswirkungen

auf die Repräsentation, sondern müssen zu einer Korrektur der Visualisierung führen. Die Längenberechnung für einen Visualisierungsknoten wird wie folgt formuliert:

Methode Sizes

Klasse:
visNode
Objekt:
Visualisierungsknoten v
Parameter:
Längenanforderung *req*
Berechnung:

1. Bestimme die Visualisierungsfunktion f von v.
2. Bestimme die Längeneigenschaften *cap* von f.
3. Wenn die Längenanforderung *req* „fest“ und die Längeneigenschaft *cap* „variabel“ ist, dann signalisiere einen Fehler.
4. Wenn f eine komplexe Visualisierungsfunktion ist, dann:
   4.1. Bestimme die Längenanforderung req' von f.
   4.2. Für jeden Nachfolgerknoten v' von v:
      4.2.1. Führe die Methode Sizes von v' mit dem Parameter req' aus.
5. Führe die Width- und Height-Methoden von f mit dem Parameter v aus und bestimme die Breite w und Höhe h von v.
6. Wenn die Längenanforderung *req* „fest“ und die Längeneigenschaft *cap* „abhängig“, aber die bestimmte Länge Null ist, dann signalisiere einen Fehler.
7. Trage die bestimmten Längen für v ein.

Rückgabe:
—

3.6 Konstruktion und Modifikation von Visualisierungen

Die konzeptuelle Definition einer Repräsentation durch eine Visualisierung wird jetzt umgesetzt, indem Verfahren zur Konstruktion und Modifikation von Visualisierungen entwickelt werden.

Eine Anforderung war, Repräsentationen beliebiger Datenobjekte *ohne* Konfiguration generieren zu können (siehe Abschnitt 3.1). Dieses Ziel wird erreicht, indem das Verfahren zur Konstruktion von Visualisierungen so entwickelt wird, daß es aus einem beliebigen

Schema eine korrekte Visualisierung generiert. Diese automatisch generierte Visualisierung wird **Default-Visualisierung** (oder **Default-Visualisierungsbaum**) genannt, eine mit ihm generierte Repräsentation ist eine **Default-Repräsentation**. Das Verfahren wird im folgenden Abschnitt entwickelt und beschrieben.

Ausgehend von dieser einen Visualisierung kann dann die Default-Repräsentation durch schrittweise Veränderung weiterentwickelt werden, um eine speziellen Vorstellungen oder Anforderungen entsprechende Repräsentation zu erhalten. Diese **Verfeinerung** genannte Methodik wird in Abschnitt 3.6.2 beschrieben.

3.6.1 Default-Visualisierung

Beim Generieren der Default-Visualisierung eines Schemas wird für jedes Attribut ein Visualisierungsknoten definiert, der die folgenden Komponenten hat:

- Die dem Attributtyp entsprechende Default-Visualisierungsfunktion,
- die Attributstruktur,
- der Attributname und
- Nullwerte für Parameter, Bindungen und Längen.

Dazu muß für jeden Konstruktor bzw. Typ eine Default-Visualisierungsfunktion definiert sein (siehe Abschnitt 3.8). Für Attribute, deren Typ ein Tupel-Konstruktor ist, werden für alle im Schema definierten Komponenten Nachfolgerknoten angelegt. Für Attribute, deren Typ ein Kollektions-Konstruktor ist, wird ein Nachfolgerknoten für die Visualisierung der Elemente angelegt. Für die ESCHER-spezifischen Pseudo-Tupel als Elementtyp von Kollektionen (siehe Abschnitt 2.3.1, Seite 45) wird ein Tupel-Visualisierungsknoten eingefügt, der je Komponente des Pseudo-Tupels einen Nachfolgerknoten erhält.

Für die Erstellung der Default-Visualisierung wird ein Algorithmus benutzt, der sich durch die Struktur des Schemas (siehe Abschnitt 2.3.1) leiten läßt. Die Struktur wird rekursiv durchlaufen und so die Visualisierung mit Visualisierungsknoten und zugehörigen Default-Visualisierungsfunktionen erstellt. Die Nachfolgerknoten werden jeweils aus den Sub-Attribut-Definitionen der komplexen Attribute, die Default-Visualisierungsfunktionen aus den Typen der Attribute im Schema bestimmt. Die Breiten und Höhen sowie die Mengen der Parameter und Bindungen bleiben zunächst undefiniert (Nullwert $\perp$) bzw. leer (leere Menge ϵ).

Ist die Struktur der Visualisierung erstellt, werden mit der Sizes-Methode des generierten Wurzelknotens die Höhen und Breiten berechnet. Dabei werden keine Anforderungen an die äußeren Längen des Wurzelknotens gestellt. Wenn mindestens eine der bestimmten Längen des Wurzelknotens Null ist, wird die Visualisierung um eine neue Wurzel erweitert, die ein Konstruktions-Visualisierungsknoten ist und eine die variable Länge kompensierende Konstruktions-Visualisierungsfunktion zugeordnet hat.

Die Längenberechnung könnte auch jeweils nach der Definition eines Visualisierungsknotens und seiner Nachfolgerknoten durch den Aufruf von dessen Methoden Width und Height durchgeführt werden. Um die Überprüfung der Längeneigenschaften auch für die generierte Default-Visualisierung auszuführen, wird jedoch die Methode Sizes für den Wurzelknoten benutzt.

Methode Create

Klasse:
visNode

Objekt:
—

Parameter:
Struktur s, Attribut a

Berechnung:

1. Wenn $s = c_{\text{cln}}(a', s')$ gilt[15], dann:
   1.1. Bestimme die Default-Kollektions-Visualisierungsfunktion f_{cln}.
   1.2. Generiere einen Kollektions-Visualisierungsknoten $v = (f_{\text{cln}}, s, a, \perp, \perp, \epsilon, \epsilon) \in \mathcal{V}_{\text{cln}}$.
   1.3. Führe die Create-Methode für Visualisierungsknoten, mit den Parametern s' und a' aus und bestimme den Element-Visualisierungsknoten v' von v.
2. Wenn $s = c_{\text{tuple}}((a_1, s_1), \ldots, (a_k, s_k))$ gilt, dann:
   2.1. Bestimme die Default-Tupel-Visualisierungsfunktion f_{tpl}.
   2.2. Generiere einen Tupel-Visualisierungsknoten $v = (f_{\text{tpl}}, s, a, \perp, \perp, \epsilon, \epsilon) \in \mathcal{V}_{\text{tpl}}$.
   2.3. Für alle i zwischen 1 und k:
      2.3.1. Führe die Create-Methode für Visualisierungsknoten, mit dem Parametern s_i und a_i aus und bestimme den i-ten Komponenten-Visualisierungsknoten v_i von v.
      2.3.2.
3. Wenn $s \in$ SIMPLETYPES gilt, dann:
   3.1. Bestimme die atomare Default-Visualisierungsfunktion f_{atom}.
   3.2. Generiere einen atomaren Visualisierungsknoten $v = (f_{\text{atom}}, s, a, \perp, \perp, \epsilon, \epsilon) \in \mathcal{V}_{\text{atom}}$.

Rückgabe:
Visualisierungsknoten v

[15] c_{cln} ist eine abkürzende Schreibweise für einen Kollektions-Konstruktor, steht also für c_{set}, c_{list} oder c_{bag}.

3.6.2 Verfeinerung von Visualisierungen

Die Art der Repräsentation eines Datenobjektes ist abhängig von dem Zweck, der mit dieser Repräsentation erreicht werden soll; für verschiedene Zwecke werden unterschiedliche Repräsentationen benötigt. Es genügt nicht, eine einzige Visualisierung abzuleiten, wie dies zunächst mit der Default-Visualisierung getan wird, und diese dann an den jeweiligen Zweck anzupassen. Vielmehr muß für jede neue Repräsentation eine Kopie der Default-Visualisierung, oder einer anderen, bereits weiterentwickelten Visualisierung, angelegt und diese dann verändert werden. Diese Methodik ist unter dem Begriff Versionsmanagement bereits etabliert [DT87, KC88, CJ92, Sci91]. Konzepte können von dort direkt übernommen werden. Die Veränderung der Visualisierungen wird Verfeinerung genannt, da die allgemeinen (groben) Default-Visualisierungsfunktionen durch speziellere (feinere) Visualisierungsfunktionen ersetzt werden können. Visualisierungen können verfeinert werden, indem

- andere Visualisierungsfunktionen eingetragen werden,
- Parameterwerte und Bindungen für Visualisierungsfunktionen angegeben, gelöscht oder geändert werden,
- für Tupel-Visualisierungsknoten die Anzahl und/oder Reihenfolge der Nachfolgerknoten verändert wird oder
- Konstruktions-Visualisierungsknoten eingefügt oder gelöscht werden.

An einem Visualisierungsknoten dürfen nur Visualisierungsfunktionen eingetragen werden, die dem Knotentyp entsprechen. Eine komplexe Visualisierungsfunktion kann durch eine atomare ersetzt werden, wenn diese eine spezielle Visualisierungsfunktion ist, die zur komplexen Struktur paßt. Dann werden alle Nachfolgerknoten, bzw. die entsprechenden Teilbäume des ehemals komplexen Visualisierungsknotens gelöscht. Wenn z.B. ein Address-Tupel (siehe Beispiel 2.1 auf Seite 46, Abb. 2.3(b)) durch eine spezielle Visualisierungsfunktion dargestellt werden soll, dann werden der Tupel-Visualisierungsknoten und seine vier nachfolgenden, atomaren Visualisierungsknoten durch einen atomaren Visualisierungsknoten ersetzt.

Für Tupel-Visualisierungsfunktion kann angegeben werden, welche Komponenten dargestellt werden (entsprechend der Projektion im relationalen Datenmodell), und in welcher Reihenfolge diese dargestellt werden. Dies geschieht, indem im Visualisierungsknoten die Anzahl bzw. Reihenfolge der Nachfolgerknoten manipuliert wird. Das Löschen eines Nachfolgerknotens entspricht dem Ausblenden der entsprechenden Komponente. Wie sich die Reihenfolge der Nachfolgerknoten auf die Reihenfolge der Komponenten in der Repräsentation auswirkt, ist zwar von der gewählten Tupel-Visualisierungsfunktion abhängig. Sie kann z.B. so realisiert sein, daß sie die Reihenfolge der Komponenten selbständig beeinflußt, etwa indem zuerst atomare Komponenten und danach komplexe Komponenten dargestellt werden. Im Allgemeinen entspricht jedoch die Reihenfolge der Nachfolgerknoten

der Reihenfolge der Komponenten in der Repräsentation, so daß durch die Veränderung der Reihenfolge der Nachfolgerknoten auch die Reihenfolge der Komponenten in der Repräsentation verändert wird.

Startet man bei der Verfeinerung von Visualisierungen nicht bei den Blättern des Baumes, kann dies zeitweilig zu einer inkonsistenten Visualisierung führen. Dann könnten nämlich die Anforderungen einer Visualisierungsfunktion an ihre Sub-Visualisierungsfunktion nicht erfüllt sein.

Beispiel 3.16 Inkonsistente Verfeinerung

Angenommen, der Visualisierung aus Abb. 3.10(c) auf Seite 87 soll so verändert werden, daß die Menge der vorausgesetzten Vorlesungen in einem vertikal rollbaren Viewport dargestellt wird. Dieser soll so hoch wie drei Elemente (Vorlesungs-ID Course) sein, der äußere Viewport Courses soll so hoch wie ein Element (Vorlesungs-Tupel Course) sein. Wird zuerst die Parametrisierung des äußeren Viewports vorgenommen[16], ist dessen Längenanforderung einer festen Höhe der Sub-Sub-Visualisierungsfunktion noch nicht erfüllt, da das Tupel Course noch die variabel hohe Menge Req enthält. □

Aus den Beispielen in Abschnitt 3.3.2 wird bereits deutlich, daß die Eigenschaften einer angewendeten Visualisierungsfunktion von den aktuellen Parametern abhängen können. Daraus folgt, daß die Längenberechnung einer Visualisierung nach der Änderung eines Parameters ungültig werden kann.

Nach dem Verfeinern ist es daher notwendig, die Breiten und Höhen der modifizierten und der davon abhängigen Visualisierungsknoten durch Aufruf der Sizes-Methode neu zu berechnen. Da es bei nicht erfüllten Anforderungen einer Visualisierungsfunktion an ihre Sub-Visualisierungsfunktionen zu Fehlermeldungen kommt, werden dadurch auch diesbezügliche Inkonsistenzen der Visualisierung aufgedeckt.

Das in Abschnitt 3.2.1 angesprochene Löschen und Einfügen von Komponenten in Tupeln wird durch den Mechanismus der Nachfolgerknoten, die die Repräsentation von Tupelkomponenten definieren, behandelt. Neu eingefügte Komponenten werden von einer Tupel-Visualisierungsfunktion nicht dargestellt, solange die Visualisierung diese nicht durch neue Nachfolgerknoten reflektiert. Für gelöschte Komponenten müssen in der Visualisierung die entsprechenden Nachfolgerknoten mit den dazugehörigen Teilbäumen gelöscht werden.

Ausgetauschte Kollektions-Konstruktoren sollen von allen Kollektions-Visualisierungsfunktionen gehandhabt werden können. Veränderte atomare Datentypen werden von einigen atomaren Visualisierungsfunktionen selbständig kompensiert (z. B. von der atomaren Default-Visualisierungsfunktionen atom, siehe Abschnitt 5.3.1). Ist das nicht der Fall, muß die Visualisierungsfunktionen des betroffenen Visualisierungsknotens durch eine geeignete ersetzt werden.

[16] Wird die Visualisierungsfunktion scrollV relativ zur Höhe einer Sub--Sub-Visualisierungsfunktion parametrisiert (z. B. `height=1s`), muß deren Höhe fest sein (siehe Abschnitt 5.5.1).

Die Verfeinerung von Visualisierungen durch Ausblenden von Tupelkomponenten hat eine starke Affinität zur Projektion im relationalen Datenmodell. Setzt man für Kollektionen Visualisierungsfunktionen mit Filterfunktionalität ein, die ihre Elemente selektiv darstellen, entsprechen Visualisierungen quasi einem relationalen $\pi\sigma$-View, d. h. einer Sicht, die nur durch Projektion und Selektion definiert ist.

Faßt man die Verfeinerung einer Visualisierung als Sichtbildung auf und definiert man weiterhin die Generierung der Default-Visualisierung als Abbildung $\mathrm{vis}(\cdot, *)$ und die Verfeinerung einer Visualisierung bzw. die Schematransformation einer Sicht als Abbildung $\mathrm{view}(\cdot, r)$, erhält man folgendes Diagramm (S sei ein Schema, r die Sichtdefinition):

$$\begin{array}{ccc} S & \xrightarrow{\mathrm{vis}(\cdot,*)} & \mathrm{vis}(S,*) = {V_S}^* \\ {\scriptstyle \mathrm{view}(\cdot,r)}\downarrow & & \downarrow{\scriptstyle \mathrm{view}(\cdot,r)} \\ \mathrm{view}(S,r) = S' & \xrightarrow[\mathrm{vis}(\cdot,*)]{} & \mathrm{vis}(S',*) = {V_{S'}}^* = {V_S}' = \mathrm{view}({V_S}^*, r) \end{array}$$

Das bedeutet, daß die Default-Visualisierung des Sichtschemas identisch mit der verfeinerten Visualisierung des Ausgangsschemas ist, d. h. $\mathrm{vis}(\cdot, *)$ und $\mathrm{view}(\cdot, r)$ kommutieren:

$$\mathrm{vis}(\mathrm{view}(S, r), *) = \mathrm{view}(\mathrm{vis}(S, *), r)$$

3.6.3 Schemata, Tabellen und Visualisierungen

Ein Schema ist die strukturelle Definition komplexer Datenobjekte. Jedes Schema kann quasi als Schablone für viele Tabellen verwendet werden. Zwischen Schemata und Tabellen besteht also eine 1:n-Beziehung. Ein Schema ist aber auch Vorlage für jede Visualisierung, die aus dem Schema abgeleitet wurde. Also besteht auch zwischen Schemata und Visualisierungen eine 1:n-Beziehung. Wie ist nun die Beziehung zwischen Tabellen und Visualisierungen? Für jede Tabelle, die zu einem Schema definiert wurde, kann mit jeder Visualisierung, die aus dem Schema abgeleitet wurde, eine Repräsentation generiert werden. Zwischen Tabellen und Visualisierungen besteht demnach eine m:n-Beziehung. Der Zusammenhang zwischen Schemata, Tabellen und Visualisierungen ist in Abb. 3.17 dargestellt.

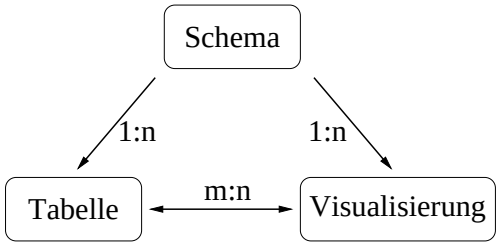


Abbildung 3.17: Zusammenhang zwischen Schemata, Tabellen und Visualisierungen

3.7 Visualisierungen für Schemata

Die für X-ESCHER beschriebene Repräsentation der zu einem Datenobjekt passenden Struktur (siehe Abschnitt 3.1.2) wird erreicht, indem geeignete Visualisierungsfunktionen für das Tupel Attribute und die Menge Sub-Attributes des Boot-Schemas (siehe Abschnitt 2.3.3) verwendet werden. Dadurch wird das Potential des hier vorgestellten, generischen und parametrisierbaren Visualisierungsverfahrens voll ausgenutzt; im Gegensatz dazu wird in X-ESCHER für die Repräsentation des Schemas prinzipiell ein eigenständiges, von der Repräsentation der Tabellen unabhängiges Visualisierungsverfahren benutzt.

Im Folgenden wird ein Algorithmus formuliert, der für eine gegebene Visualisierung, die zu einem Schema paßt, eine Visualisierung des Schemas selbst generiert. Genauer gesagt soll dieser Algorithmus aus einer Visualisierung V, die zu einem Schema S paßt, eine Visualisierung V' erzeugen, die zum Boot-Schema paßt. Diese Visualisierung V' soll für das Schema S, als Tabelle zum Boot-Schema, eine Repräsentation definieren, die zu Repräsentationen paßt, die durch die Visualisierung V für Tabellen zum Schema S definiert sind. Eine Visualisierung für ein Schema heißt **Schemavisualisierung** (oder **Schemavisualisierungsbaum**). Falls keine Verwechslungen möglich sind, werden auch Schemavisualisierungen nur „Visualisierung“ genannt.

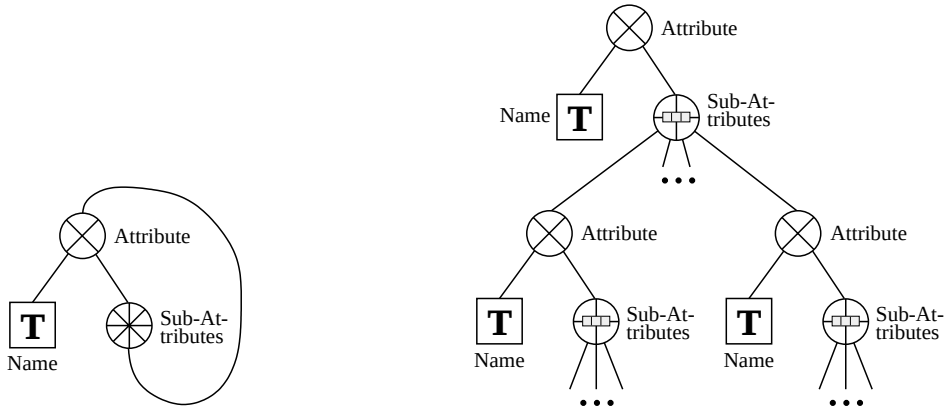
Der Algorithmus trägt an den Visualisierungsknoten, die die Attribute Name, Attribute und Sub-Attributes repräsentieren, geeignete Visualisierungsfunktionen ein und parametrisiert diese entsprechend.

Ein Problem, das hier auftritt, ist die rekursive Definition des Boot-Schemas. Dieses Problem kann für das Visualisierungsverfahren prinzipiell auf zwei Weisen gelöst werden: Entweder wird die Definition der Visualisierungen so erweitert, daß diese auch Zyklen enthalten dürfen[17], d. h. die Visualisierung wird quasi rekursiv. Oder die Visualisierung wird datenabhängig generiert, d. h. die Rekursion des Boot-Schemas wird in der Visualisierung durch Replikation explizit gemacht. Dies ist keine Einschränkung, da mit dem rekursiv definierten Boot-Schema nur endliche Strukturen beschrieben werden.

In Abb. 3.18 auf der nächsten Seite sind die beiden Möglichkeiten skizziert, wobei nur die Attribute Name und Sub-Attributes des Boot-Schemas in den Visualisierungen enthalten sind. Das Symbol ⊕ in Abb. 3.18(b) repräsentiert die spezielle Visualisierungsfunktion subAttr, mit der die Mengen Sub-Attributes des Schemas dargestellt werden. Diese Verwendung einer datenabhängigen Visualisierung ist nur für Daten sinnvoll, die als invariant eingestuft werden können. Genau dieses trifft auf Schemata zu, die bereits als strukturell definierende Grundlage für Tabellen und Visualisierungen verwendet werden.

Eine rekursiv definierte Visualisierung hat den Nachteil, daß die Breite der atomaren Attribute, die mit der Breite in der Datenobjekt-Repräsentation übereinstimmen muß (siehe

[17] Im graphentheoretischen Sinn darf dann nicht mehr von einem Visualisierungsbaum gesprochen werden, hier wird jedoch die Bezeichnung Visualisierungsbaum auch weiterhin verwendet.



(a) Rekursive Visualisierung (b) Datenabhängige Replikation

Abbildung 3.18: Visualisierungen für das Boot-Schema

Abschnitt 3.1), nur durch eine umständliche, datenabhängige Parametrisierung des Visualisierungsknotens für das Attribut Name erreicht werden kann. Diese Parametrisierung müßte außerdem durch eine spezielle atomare Visualisierungsfunktion interpretiert werden. Weiterhin wäre es nicht möglich, Konstruktions-Visualisierungsfunktionen nach Bedarf in die Visualisierung einzufügen.

Die erwähnten Nachteile entstehen nicht, wenn die Visualisierung datenabhängig generiert wird. In diesem Fall hat jede Instanz des Attributes Name einen eigenen Repräsentanten in der Visualisierung, an dem die jeweilige Parametrisierung vorgenommen werden kann. Die Visualisierungsfunktion atom (siehe Abschnitt 5.3.1) ist dann geeignet, um die Attributnamen darzustellen, aber die oben erwähnte spezielle Visualisierungsfunktion subAttr muß implementiert werden.

Die Tupel im Boot-Schema (Attribut Attribute) entsprechen der Definition eines Attributes im Schemabaum. Atomare Attribute sind dadurch gekennzeichnet, daß sie leere Sub-Attribut-Mengen haben. In der Repräsentation sollen diese leeren Mengen aber nicht dargestellt werden, sondern die entsprechenden Tupel sollen nur durch die Komponente Name repräsentiert werden. Zu diesem Zweck wird eine spezielle, atomare Visualisierungsfunktion atomAttr definiert, die genau die eben beschriebene Darstellung realisiert (siehe Abschnitt 5.3.6). Sie wird an den Blättern der Visualisierung eingetragen und jeweils so parametrisiert, daß sie die Visualisierungsbreiten der atomaren Attribute reflektiert.

Komplexe Attribute sollen dargestellt werden, indem der Attributname oberhalb der Struktur des Attributes dargestellt wird. Für die Darstellung dieser Tupel kann die Default-Tupel-Visualisierungsfunktion tpl (siehe Abschnitt 5.7.1) verwendet werden. Die entsprechenden Visualisierungsknoten werden für eine vertikale Orientierung parametrisiert, so daß der Attributname (Komponente Name) über den Sub-Attributen (Komponente Sub-

Attributes) dargestellt wird. Dabei wird die Breite des Tupels immer durch die Breite der Sub-Attribute bestimmt. Auf diese Weise werden ESCHERs Pseudo-Tupel nicht nur korrekt, d. h. ohne zusätzlichen Attributnamen im Schema dargestellt; durch Verfeinerung der Visualisierung können sogar genuine Tupel wie die Pseudo-Tupel in X-ESCHERs Schemarepräsentation dargestellt werden (Löschen des Nachfolgerknotens Name im entsprechenden Tupel-Visualisierungsknoten).

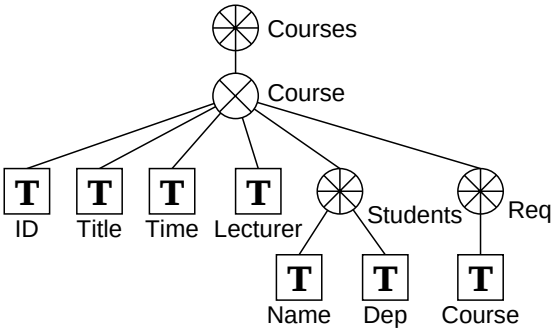
Die Mengen Sub-Attributes im Boot-Schema entsprechen der Struktur eines komplexen Attributes im Schemabaum, indem sie die Struktur der Sub-Attribute definieren. Je nachdem, welche Visualisierungsfunktion für den zugehörigen, komplexen Visualisierungsknoten eingetragen ist, muß im Visualisierungs eine dazu passende Visualisierungsfunktion eingetragen werden. Die Information, welche Visualisierungsfunktion zu einer anderen als **Schemavisualisierungsfunktion** paßt, wird als Eigenschaft einer Visualisierungsfunktion als Meta-Information gespeichert (siehe Abschnitt 3.8.2). Für die Default-Visualisierungsfunktionen von Kollektionen und Tupeln wird die Visualisierungsfunktion subAttr (siehe Abschnitt 5.7.3) verwendet. Diese stellt die Komponenten, die die Sub-Attribute repräsentieren, nebeneinander dar.

Beispiel 3.17 Schemarepräsentation

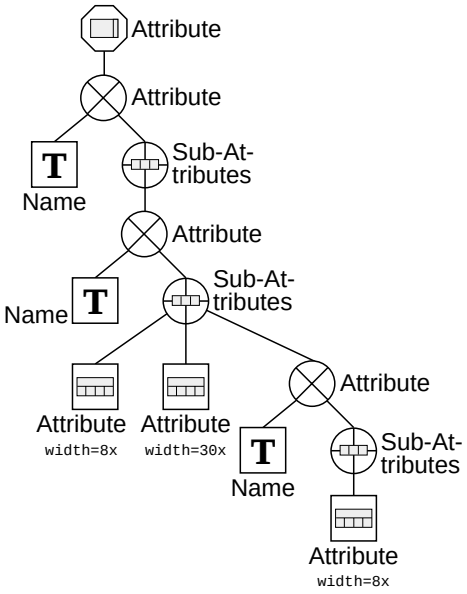
In Abb. 3.19 auf der nächsten Seite sind für die Vorlesungstabelle alle „definierenden Bäume“ dargestellt, die für die Generierung der ebenfalls dargestellten Repräsentation verwendet werden. Im einzelnen sind in Abb. 3.19(a) der Schemabaum des Schemas Courses, in Abb. 3.19(c) eine dazu passende Visualisierung, in Abb. 3.19(b) die zugehörige Schemavisualisierung, sowie in Abb. 3.19(d) eine damit generierte Repräsentation dargestellt. Die spezielle Visualisierungsfunktion atomAttr wird durch das Symbol repräsentiert, auf das Symbol wird weiter unten eingegangen. □

Sind in einer Visualisierung Konstruktions-Visualisierungsfunktionen enthalten, müssen diese ggf. auch in der generierten Schemavisualisierung berücksichtigt werden. Die Verbindung zwischen einer Konstruktions-Visualisierungsfunktion und einer entsprechenden Schemavisualisierungsfunktion wird wieder über die Meta-Informationen hergestellt.

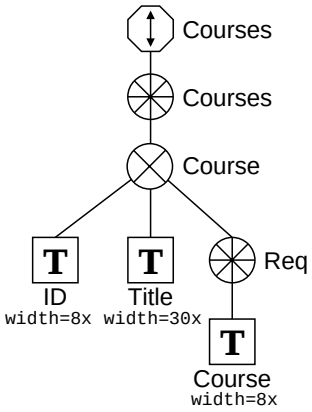
Beispiele für zu berücksichtigende Konstruktions-Visualisierungsfunktionen sind scrollH und scrollV. Erstere muß berücksichtigt werden, damit eine synchrone Viewport-Veränderung in Struktur- und Datenobjektrepräsentation erreicht wird. Dafür muß bei der Generierung beider Repräsentationen für eine Kopplung der entsprechenden Rollbalken gesorgt werden. Letztere muß berücksichtigt werden, obwohl das vertikale Rollen in den Daten unabhängig vom Schema ist. Zu beachten sind aber generell die Längen der graphischen Elemente, die von Konstruktions-Visualisierungsfunktionen angebracht werden, damit die visuelle Korrelation zwischen Schema und Datenobjekt erhalten bleibt. Für scrollV muß also im Schema an Stelle des Rollbalkens ein Platzhalter angelegt werden. Dieser ist in Abb. 3.19(b) durch das Symbol angedeutet.



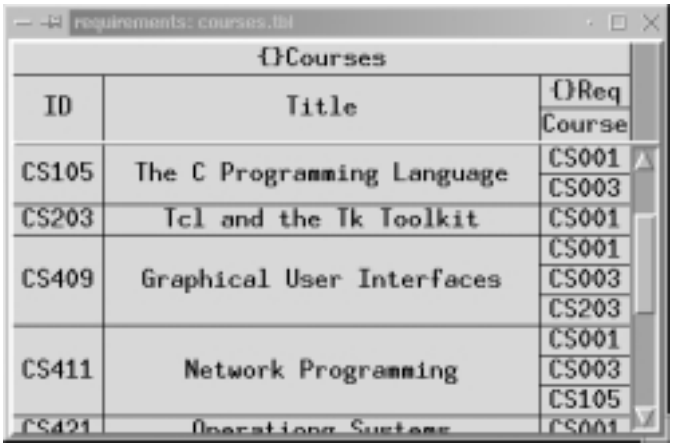
(a) Schemabaum



(b) Schemavisualisierung



(c) Visualisierung



(d) Repräsentation

Abbildung 3.19: Repräsentation einer Tabelle mit Schema

3.7.1 Generierung der Schemavisualisierung

Der Algorithmus wird als Methode der Visualisierungsknoten formuliert, die dann, für den Wurzelknoten einer Visualisierung aufgerufen, eine Schemavisualisierung generiert.

Methode Schema

Klasse:
visNode
Objekt:
Visualisierungsknoten v
Parameter:
—
Berechnung:

1. Bestimme die Visualisierungsfunktion f von v.
2. Bestimme die f entsprechende Schemavisualisierungsfunktion f_s.
3. Wenn f eine Konstruktions-Visualisierungsfunktion ist, dann:
   3.1. Generiere einen der Visualisierungsklasse von f_s entsprechenden Visualisierungsknoten v_s mit der Visualisierungsfunktion f_s.
   3.2. Bestimme den Nachfolgerknoten v' von v.
   3.3. Führe die Schema-Methode von v' aus und bestimme dessen Schemavisualisierungsknoten v'_s.
   3.4. Trage v'_s als Nachfolgerknoten von v_s ein.
4. Wenn f eine Kollektions-Visualisierungsfunktion ist, dann:
   4.1. Generiere einen Tupel-Visualisierungsknoten v_s mit der Visualisierungsfunktion tpl in vertikaler Orientierung.
   4.2. Generiere einen atomaren Visualisierungsknoten v_a für das Attribut Name des Boot-Schemas mit der Visualisierungsfunktion atom.
   4.3. Trage v_a als ersten Nachfolgerknoten von v_s ein.
   4.4. Generiere einen der Visualisierungsklasse von f_s entsprechenden Visualisierungsknoten v_c mit der Visualisierungsfunktion f_s.
   4.5. Trage v_c als zweiten Nachfolgerknoten von v_s ein.
   4.6. Bestimme den Element-Visualisierungsknoten v' von v.
   4.7. Führe die Schema-Methode von v' aus und bestimme dessen Schemavisualisierungsknoten v'_s.
   4.8. Trage v'_s als einzigen Nachfolgerknoten von v_c ein.
5. Wenn f eine Tupel-Visualisierungsfunktion ist, dann:
   5.1. Generiere einen Tupel-Visualisierungsknoten v_s mit der Visualisierungsfunktion tpl in vertikaler Orientierung.

5.2. Generiere einen atomaren Visualisierungsknoten v_a für das Attribut Name des Boot-Schemas mit der Visualisierungsfunktion atom.

5.3. Trage v_a als ersten Nachfolgerknoten von v_s ein.

5.4. Generiere einen der Visualisierungsklasse von f_s entsprechenden Visualisierungsknoten v_t mit der Visualisierungsfunktion f_s.

5.5. Trage v_t als zweiten Nachfolgerknoten von v_s ein.

5.6. Für alle Nachfolgerknoten v^i von v:

5.6.1. Führe die Schema-Methode von v^i aus und bestimme dessen Schemavisualisierungsknoten v_s^i.

5.6.2. Trage v_s^i als Nachfolgerknoten von v_t ein.

6. Wenn f eine atomare Visualisierungsfunktion ist, dann:

6.1. Generiere einen der Visualisierungsklasse von f_s entsprechenden Visualisierungsknoten v_s mit der Visualisierungsfunktion f_s.

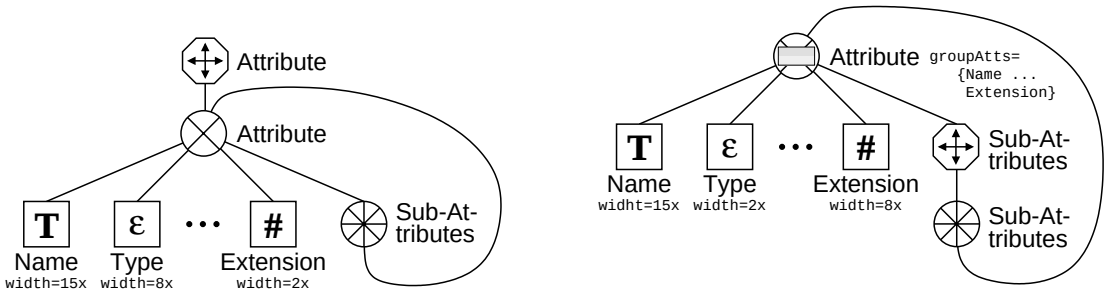
Rückgabe:
Visualisierungsknoten v_s

3.7.2 Visualisierungen für das Boot-Schema

Einen Schritt weiter muß gegangen werden, wenn eine Visualisierung des Boot-Schemas definiert werden soll. Diese Visualisierung muß jedes Schema als Datenobjekt darstellen können. In diesem Fall kann keine datenabhängige, explizite Replikation generiert werden, da die zu visualisierenden Datenobjekte verschieden sind. Statt dessen kann auf die obige Alternative der nicht zyklenfreien Visualisierungen zurückgegriffen werden. Jetzt entstehen die oben erwähnten Nachteile nicht mehr: Alle atomaren Attribute werden, in der durch ihre einmalige Definition in der Visualisierung gegebenen Breite, mit der atomaren Default-Visualisierungsfunktion dargestellt. Konstruktions-Visualisierungsfunktionen können integriert werden, werden dann aber, entsprechend der zyklischen Definition der Visualisierung, in der Repräsentation zyklisch dargestellt.

In Abb. 3.20 auf der gegenüberliegenden Seite sind zwei mögliche Visualisierungen zum Boot-Schema dargestellt. Die in Abb. 3.20(a) gezeigte Default-Visualisierung definiert eine Repräsentation, die genau der in X-ESCHER implementierten Repräsentation eines Schemas als Tabelle zum Boot-Schema entspricht. Die Wurzel wird hier durch eine Konstruktions-Visualisierungsfunktion mit einem vertikal und horizontal adjustierbaren Viewport gebildet, da in dieser Repräsentation die Schemata als Datenobjekte durch die Rekursion variable Breite und Höhe haben. Werden die Sub-Attribute wie in der in Abb. 3.20(b) gezeigten alternativen Visualisierung selbst in diesem Viewport dargestellt, hat die Repräsentation feste Längen.

Der konsequente nächste Schritt ist die Definition einer Schemavisualisierung für das Boot-Schema, so daß auch das Boot-Schema als Tabelle zum Boot-Schema, zusammen mit sich

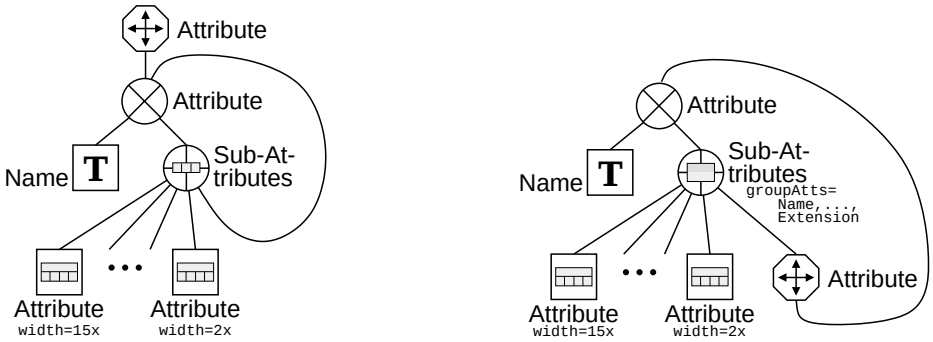


(a) Default-Visualisierung (b) alternative Visualisierung

Abbildung 3.20: Visualisierungen für das Boot-Schema

selbst als Strukturinformation, dargestellt werden kann. Realisiert wird diese Visualisierung durch die Kombination der datenabhängigen, expliziten Replikation mit einer Visualisierung, die einen Zyklus enthält[18]. Die datenabhängige, explizite Replikation ist hier sinnvoll, da das Boot-Schema genau einmal existiert und invariant ist.

Abb. 3.21 zeigt die Schemavisualisierungen, die zu den Visualisierungen aus Abb. 3.20 passen. Die jeweiligen Attributbreiten und die Parameterspezifikation der Visualisierungsfunktion tplLabel für das Attribut Attribute aus Abb. 3.20(b) sind in Abb. 3.21(b) übernommen worden. tplLabelScm (Symbol ⊕) soll die tplLabel entsprechende Schemavisualisierungsfunktion sein.



(a) Passend zu Abb. 3.20(a) (b) Passend zu Abb. 3.20(b)

Abbildung 3.21: Schemavisualisierungen für das Boot-Schema

[18]Es gilt die Aussage der Fußnote 17 auf Seite 103.

3.8 Meta-Daten

Genauso wie Schemata und Tabellen persistente Objekte sind, sollen auch Visualisierungen persistente Objekte sein. Dadurch stellt sich in natürlicher Weise die Frage, in welcher Form Visualisierungen gespeichert werden soll.

Bereits für die Speicherung von Schemata hat sich in ESCHER das Konzept der Tabelle als geeignet erwiesen, d. h. durch die Definition des Boot-Schemas können die Schemata selbst als Tabelle gespeichert werden. Auch das Boot-Schema wird, sich selbst beschreibend (siehe Abschnitt 2.3.3), als Tabelle gespeichert. Seit es im Forschungsprojekt System R eingeführt wurde [ABC+76], ist es in Datenbanksystemen mittlerweile Standard, Meta-Daten, also z. B. Schemainformation, in der Datenbank selbst zu speichern [MR86, KS91] (z. B. in Oracle [Ora97]). Auch die Speicherung von Informationen über die Benutzerschnittstelle einer Datenbank in der Datenbank selber hat sich bereits bewährt (z. B. im FADS-Projekt [Row92] oder im FaceKit-System [KN92]).

Ein Ziel dieser Arbeit ist, die angemessene Repräsentation von komplex strukturierten Objekten zu ermöglichen. Eine angemessene Repräsentation umfaßt dabei auch die Manipulierbarkeit durch Benutzerinteraktion. Beide Aspekte sind auch wichtig für die Visualisierungen, die sicherlich als komplex strukturierte Objekte bezeichnet werden können. Daher ist es insgesamt naheliegend, die Visualisierungen als Tabelle zu speichern. Durch eine geeignete Repräsentation dieser Tabellen (siehe Abschnitt 3.6.2) kann dann ein Benutzerschnittstelle zur Verfeinerung von Visualisierungen definiert werden.

Auch die Informationen über Visualisierungsfunktionen und deren Klassen müssen persistent gespeichert sein, da sie aus den Visualisierungen referenziert werden. Der nächste Abschnitt beschäftigt sich mit den Visualisierungen, die Abschnitte 3.8.2 und 3.8.3 beschäftigen sich mit den Visualisierungsfunktionen bzw. der Modellierung der Klassenhierarchie aus Abschnitt 3.3.1. In Abschnitt 6.1.2 werden im Rahmen einer Beispielanwendung Visualisierungen und damit generierte Repräsentationen der im Folgenden definierten Schemata dargestellt.

3.8.1 Visualisierungsschema

Die Struktur der Tabellen, in der die Visualisierungen gespeichert werden, d. h. das Schema aller Visualisierungen, wird **Visualisierungsschema** genannt. Das Visualisierungsschema muß so beschaffen sein, daß es eine Visualisierung gemäß Abschnitt 3.4 modelliert. Der Schemabaum des Visualisierungsschemas ist in Abb. 3.22 auf der gegenüberliegenden Seite dargestellt.

Um die Meta-Informationen nicht redundant zu speichern, wird der Typ Link benutzt und auf Informationen aus anderen Meta-Tabellen verwiesen, z. B. auf Schemata, Attribute und Visualisierungsfunktionen.

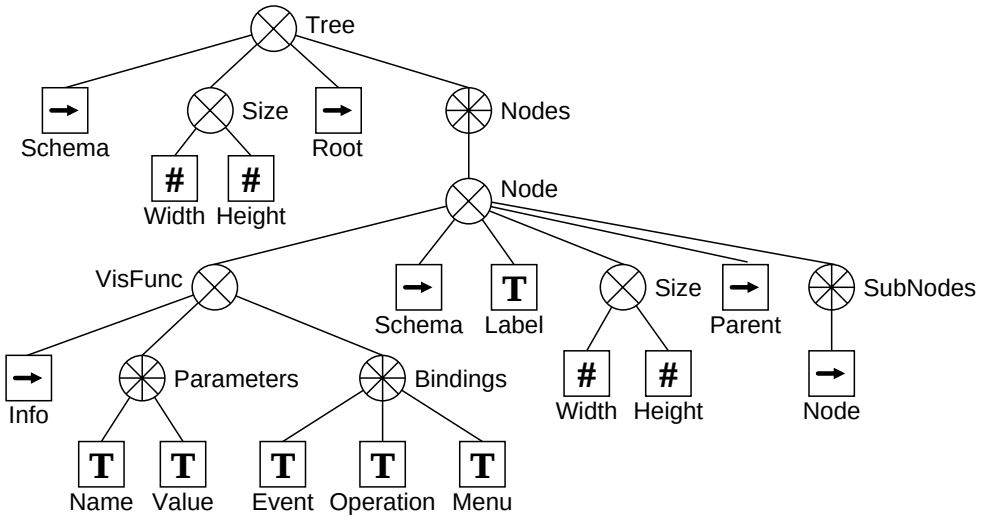


Abbildung 3.22: Visualisierungsschema

Das Visualisierungsschema enthält mit dem Attribut Schema des Tupels Tree zunächst einen Verweis auf das Schema, für das die Visualisierung definiert ist (Link auf ein Table-Tupel der Meta-Tabelle Applications, siehe Beispiel 2.2 auf Seite 47). Die Visualisierung ist durch seine Knotenmenge Nodes repräsentiert. Die Kantenrelation ist auf die einzelnen Knoten als Liste der Nachfolgerknoten (Attribut SubNodes) verteilt. Als Einstieg in die Visualisierung ist das Attribut Root ein Verweis auf den Wurzelknoten (Link auf ein Node-Tupel).

Die Visualisierungsknoten werden jeweils durch ein Tupel repräsentiert, dessen Struktur an die Definition aus den vorangehenden Abschnitten angelehnt ist. Für einen Knoten $v = (f, s, a, P, B, w, h)$ werden

- die Visualisierungsfunktion f durch einen Verweis auf die Meta-Information der Visualisierungsfunktion (die Komponente Info des Tupels VisFunc ist ein Link auf ein VisFunc-Tupel in der Tabelle der Visualisierungsfunktionen, die im Folgenden Abschnitt 3.8.2 beschrieben wird),
- die Struktur s und das Attribut a durch einen Verweis auf die Attributdefinition im Schema (das Attribut Schema ist ein Link auf ein Attribute-Tupel in der das Schema repräsentierenden Tabelle),
- P und B durch je eine Menge von Parametern und Bindungen (Komponenten Parameters und Bindings des Tupels VisFunc),
- und die Breite w und die Höhe h durch die Komponenten Width und Height des Tupels Size repräsentiert.

Die aktuellen Parameter werden als Name-Wert-Paare angegeben (Komponenten Name und Value der Menge Parameters). Für Bindungen werden jeweils vier Komponenten ange-

geben: Ein Tastaturereignis, die zugehörige Operation und ein Menüeintrag (die Attribute heißen Event, Operation und Menu). Die Komponenten Info, Parameters und Bindings sind in dem Tupel VisFunc zusammengefaßt, das eine Komponente des Node-Tupels ist. Der Menüeintrag der Bindungen wird später dazu benutzt, um kontextsensitive Pop-Up-Menüs zu definieren (siehe Abschnitt 4.3).

Die Breite und Höhe eines Knotens hängen von der Visualisierungsfunktion, ggf. von den aktuellen Parametern, sowie ggf. von den Breiten und Höhen der Nachfolgerknoten ab. Sie können jedoch nicht immer für alle Knoten einer Visualisierung im Voraus bestimmt werden, z. B. wenn die Höhe einer Kollektion von der Elementanzahl abhängt. In diesen Fällen sind die Längen undefiniert, was sich, abhängig von den Eigenschaften der Visualisierungsfunktionen, nach außen fortpflanzen kann (siehe Abschnitt 3.5). Die Breite und Höhe der gesamten Visualisierung werden in der Komponente Size des Tupels Tree auf äußerster Ebene nochmal gespeichert.

Jeder Knoten enthält die Liste seiner Nachfolgerknoten (Komponente SubNodes, die Elemente Node der Liste sind jeweils Verweise auf ein Node-Tupel). Zur Unterstützung der Navigation enthält jedes Node-Tupel mit der Komponente Parent einen Verweis auf das Node-Tupel, dessen Nachfolgerknoten es repräsentiert (der Parent-Link des Wurzelknotens ist Null).

Die Modellierung eines Zyklus in Visualisierungen, die in Visualisierungen des Boot-Schemas notwendig ist (siehe Abschnitt 3.7.2), ist leicht möglich, da das Link-Ziel eines Node-Links ein beliebiger Knoten des „Baumes“ sein kann, also auch ein Vorgängerknoten.

Die Komponente Label der Node-Tupel dient der Benennung von Attributen (entsprechend der Umbenennung im relationalen Datenmodell oder der Benennung von Spalten mit „AS“ in SQL). Der dargestellte Attributname kann so vom Attributnamen, der durch das zur Visualisierung passende Schema definiert wird, verschieden sein. Ein Beispiel für eine Visualisierungsfunktion, die Attributnamen auch in der Datenobjekt-Repräsentation darstellt, ist tplLabel (siehe Abschnitt 5.7.2).

Wichtig für die Verfeinerung von Visualisierungen (siehe Abschnitt 3.6.2) ist, daß die Visualisierungen selber angemessen dargestellt werden. Die Repräsentation von Visualisierungen muß folgendes ermöglichen:

- Navigation in der Visualisierung,
- Einfügen und Löschen von Konstruktions-Visualisierungsknoten,
- Löschen und Umordnen der Nachfolgerknoten von Tupel-Visualisierungsknoten,
- Auswahl von Visualisierungsfunktionen, insbesondere auch von speziellen Visualisierungsfunktionen für Objekttypen,
- Definition und Veränderung von Parametern und Bindungen,
- Ausführung der Längenberechnung.

Diese Punkte müssen angemessen umgesetzt werden, z. B. durch

- die Navigation unterstützende Bindungen,

- wertabhängige Pop-Up-Menüs, d. h.
  - Auswahl der Visualisierungsfunktion abhängig vom Knotentyp und
  - Manipulation von Parametern und Bindungen abhängig von der Visualisierungsfunktion,
- sowie eine Bindung der Methode Sizes.

Die Anforderungen werden durch die Visualisierung Wizard, die in Abschnitt 6.1.2 für das Visualisierungsschema definiert wird, erfüllt. Dort ist auch eine Repräsentation einer Visualisierung als durch das Visualisierungsschema strukturiertes Datenobjekt dargestellt (Abb. 6.6 auf Seite 236).

3.8.2 Visualisierungsfunktionen

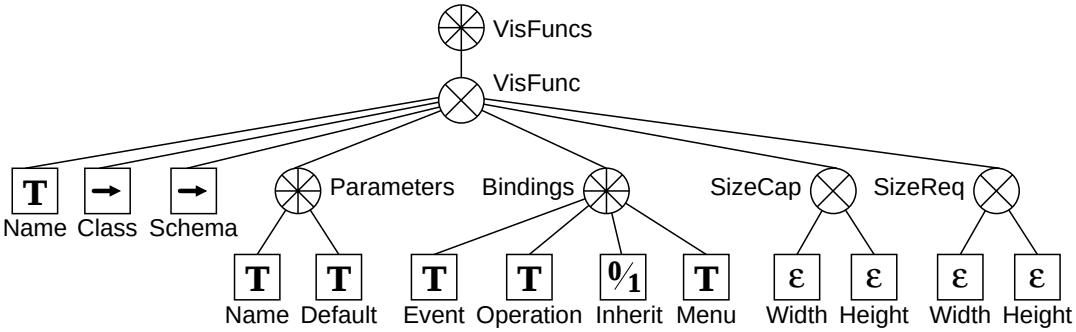


Abbildung 3.23: Schema der Visualisierungsfunktionen-Tabelle

Die Informationen über Visualisierungsfunktionen werden in einer Tabelle gespeichert, die nach dem in Abb. 3.23 dargestellten Schema strukturiert ist. Diese Tabelle wird **Visualisierungsfunktionen-Tabelle** genannt. Das Schema enthält für jede Visualisierungsfunktion ein Tupel (Attribut VisFunc) mit den folgenden Komponenten:

- der Namen der Visualisierungsfunktion (Attribut Name),
- ein Verweis auf die Visualisierungsklasse (das Attribut Class ist ein Link auf ein Class-Tupel in der Visualisierungsklassen-Tabelle, die im Folgenden Abschnitt 3.8.3 beschrieben wird),
- ein Verweis auf die Schemavisualisierungsfunktion (das Attribut Schema ist ein Link auf ein VisFunc-Tupel in derselben Tabelle),
- eine Menge von Parametern (Attribute Parameters), die Tupel aus Parameternamen (Komponente Name) und Default-Werten (Komponente Default) enthält,

- eine Menge von Bindungen (Attribut Bindings), deren Elementtupel zusätzlich zu den Komponenten des gleichnamigen Attributes aus dem vorhergehenden Abschnitt 3.8.1 ein Vererbungs-Flag enthalten (Attribut Inherit),
- die Spezifikation der Längeneigenschaften (Tupel SizeCap) und
- die Spezifikation der Längenanforderungen (Tupel SizeReq).

Da die Default-Werte der Parameter von verschiedenen Typen sein können, wird hier ihre textuelle Repräsentation als String modelliert. Längeneigenschaften und -anforderungen werden als Tupel angegeben, deren Komponenten jeweils die Breite und Höhe repräsentieren, wobei die Dimensionen unabhängig voneinander sind (eine Kollektions-Visualisierungsfunktion kann in der einen Dimension fest, in der anderen variabel sein). Längenanforderungen werden nur von Visualisierungsfunktionen gestellt, die sich auf Sub-Visualisierungsfunktionen beziehen, also von komplexen Visualisierungsfunktionen; für atomare Visualisierungsfunktionen werden hier Nullwerte eingetragen. Die Spezifikation erfolgt über den Aufzählungstyp SizeReq, der folgende Werte umfaßt:

- „fix“: Fest, d. h. alle Sub-Visualisierungsfunktionen müssen nach außen eine feste Länge zur Verfügung stellen.
- „none“: Keine, d. h. die Längen der Sub-Visualisierungsfunktionen können variabel sein.

Die Längeneigenschaften werden durch den Aufzählungstyp SizeCap spezifiziert, der folgende Werte umfaßt:

- „fix“: Fest, d. h. die äußere Länge hängt nicht von den Sub-Visualisierungsfunktionen oder dem zu visualisierenden Datenobjekt ab.
- „dep“: Abhängig, d. h. die Eigenschaft einer Länge hängt von der Parametrisierung oder den Längeneigenschaften der Sub-Visualisierungsfunktionen ab.
- „var“: Variabel, d. h. die Eigenschaft einer äußeren Länge hängt von dem zu visualisierenden Datenobjekt ab.

Der Inhalt der Visualisierungsfunktionen-Tabelle wird beispielhaft in Kapitel 5 bei der Beschreibung der Visualisierungsfunktionen dargestellt (siehe z. B. Abb. 5.3 auf Seite 185).

3.8.3 Klassenhierarchie

Bindungen für Visualisierungsfunktionen werden u. a. über Vererbung durch die Klassenhierarchie definiert (siehe Kapitel 5). Dieser Mechanismus muß ebenfalls als Meta-Information modelliert werden. Das Schema der **Visualisierungsklassen-Tabelle** ist in Abb. 3.24 auf der gegenüberliegenden Seite dargestellt. Es enthält für jede Visualisierungsklasse (Tupel VisClass)

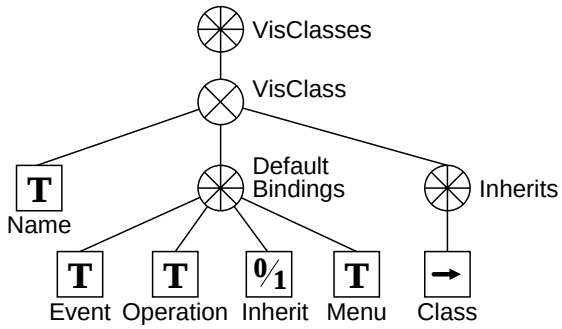


Abbildung 3.24: Schema der Visualisierungsklassen-Tabelle

- den Namen der Klasse (Attribut Name),
- deren Default-Bindungen (Menge Default Bindings) und
- eine Liste mit Verweisen auf die Klassen, von denen die entsprechende Klasse erbt (Attribut Inherits; die Elemente Class sind jeweils Verweise auf ein VisClass-Tupel).

3.9 Erweiterungen

Jede Verfeinerung einer Visualisierung ist eigentlich bereits eine Erweiterung, nämlich eine Erweiterung der visuellen Schnittstelle des Datenbank-Editors. Durch die Implementierung weiterer Visualisierungsfunktionen kann auch das Visualisierungsverfahren selbst, im vorgegebenen Rahmen, erweitert werden. Aber auch der durch die Klassenhierarchie „vorgegebene Rahmen“ ist erweiterbar. Zusätzlich kann noch die Erweiterbarkeit des Datenmodells betrachtet sowie von einer Erweiterbarkeit der Interaktion gesprochen werden.

3.9.1 Datenmodell

Durch geeignete Visualisierungsfunktionen können Erweiterungen des vordefinierten Typsystems bzw. des zugrunde liegenden Datenmodells simuliert werden. Ein Aufzählungstyp kann z. B. realisiert werden, indem der Wertebereich eines skalaren Datentyps eingeschränkt und durch eine entsprechende Visualisierungsfunktion dargestellt wird. Ist z. B. der Typ „Geschlecht“ als Integer definiert, kann durch eine Darstellung der Werte 1 als „weiblich“ und 0 als „männlich“ einerseits eine intuitive Repräsentation erreicht, andererseits die Wertemenge auf den zulässigen Bereich eingeschränkt werden. Als Darstellung könnte z. B. die textuelle Repräsentation der Label dienen, die Eingabe könnte durch ein Pop-Up-Menü, das den zulässigen Wertebereich darstellt, geschehen (siehe Beispiel 5.4 auf Seite 184).

Weitere Kandidaten für die Erweiterung des Datenmodells durch Visualisierungsfunktionen sind die in [The96] eingeführten Felder, bzw. deren Spezialformen Matrix (zweidimensional)

und Vektor (eindimensional). Wie auch diese Arbeit ist [The96] im Umfeld von ESCHER entstanden. Felder sind jedoch bisher nicht in X-ESCHER umgesetzt worden. Sie können nun simuliert werden, indem eine Kollektions-Visualisierungsfunktion definiert wird, die Kollektionen mit der entsprechenden Elementanzahl als Felder darstellt. Insbesondere können Matrizen durch horizontale und vertikale Konkatenation der Matrixelemente dargestellt werden.

Auch durch Funktionen berechnete Werte können so in das Datenmodell integriert werden. Durch die Verbindung mit Tcl/DB können als Attributwerte gespeicherte Skripten durch eine Visualisierungsfunktion ausgeführt und ihre Ergebnisse dargestellt werden. Noch weiter geht die Verwendung von Skripten, die durch die Verbindung mit Tcl/Tk die Repräsentation direkt beeinflussen.

Eine solche „Erweiterung“ des Datenmodells muß durch eine geeignete Visualisierungsfunktion definieren werden. Damit nicht für jeden Typ eine neue Visualisierungsfunktion definiert werden muß, sollte die Parametrisierung der Visualisierungsfunktionen ausgenutzt werden, um für bestimmte Klassen von Typen quasi generische Visualisierungsfunktionen zu definieren.

Für Aufzählungstypen könnte eine Tabelle definiert werden, die für jeden Aufzählungstyp die Zuordnung der skalaren Werte zu den Labeln (als String-Werte) festlegt. Aus dieser Tabelle könnte die Visualisierungsfunktion für Aufzählungstypen dann mit dem konkreten Typ und dem skalaren Wert eines Aufzählungstyp-Datenobjektes dessen Repräsentation erstellen. Außerdem ließe sich aus der Tabelle das Pop-Up-Menü für die Eingabe generieren.

Für berechnete Werte wären „virtuelle“ Attribute denkbar, wenn für alle Instanzen eines Attributes die gleiche Funktion verwendet wird. Die Berechnungsvorschriften solcher Funktionen könnten wieder als Tcl-Skripten in Tabellen gespeichert und mit den Attributtypen assoziiert werden.

3.9.2 Interaktion

Auch die Interaktion mit Datenobjekten kann erweitert werden. Spezielle Visualisierungsfunktionen können zum Editieren der von ihnen dargestellten Datenobjekte völlig unabhängige Interaktionskonzepte realisieren. Z. B. kann eine graphische Polygon-Visualisierungsfunktion die direkte Manipulation von Polygon-Ecken mit der Maus implementieren, so daß die Koordinaten der Ecken simultan verändert werden. Visualisierungsfunktionen können das durch die Default-Bindungen der Visualisierungsklassen vordefinierte Interaktionskonzept durch eigene Bindungen überschreiben und/oder erweitern und so für ihre speziellen Anforderungen anpassen (siehe Abschnitte 3.3.3, 3.8 und 4.3). Visualisierungen können im Zuge der Verfeinerung mit speziellen Bindungen versehen werden, so daß für jede Repräsentation individuelle, anwendungsspezifische Erweiterungen des Interaktionskonzeptes vorgenommen werden können.

3.9.3 Visualisierungsverfahren

Das Visualisierungsverfahren ist bereits dadurch erweiterbar, daß durch die Definition von Visualisierungsfunktionen neue Darstellungsalternativen realisiert werden können. Diese müssen aber stets aus den vier vorgegebenen Visualisierungsklassen generiert werden und sind daher in einem, wenn auch flexiblen, Rahmen gehalten.

Doch auch dieses Schema ist nicht starr, sondern erweiterbar, da die Definition neuer Visualisierungsklassen nicht ausgeschlossen wurde. Es ist z. B. vorstellbar, daß die ursprüngliche Klassenhierarchie ohne die Visualisierungsklasse constrVF definiert war und durch ihr Hinzufügen erweitert worden ist. Eine neue Visualisierungsklasse muß natürlich in der Definition der Visualisierungen, in deren Realisierung als Meta-Daten, und in den Methoden der Visualisierungsknoten berücksichtigt werden. Diese Erweiterbarkeit der Klassenhierarchie ist wichtig, um auch neue, jetzt noch nicht berücksichtigte Darstellungsarten in das Visualisierungsverfahren integrieren und so für alle komplexen Strukturen angemessene Repräsentationen erzeugen zu können („A high level framework simplifies the definition of applications because [...] custom direct manipulation editors can be developed for each object type.“ [Row92, S. 9]).

Kapitel 4

Visualisierungen — dynamischer Teil

Im statischen Teil des Visualisierungsverfahrens wurde beschrieben, wie für ein gegebenes Datenobjekt (Tabelle) bzw. dessen Struktur (Schema) Visualisierungen definiert und diese durch den Anwender verfeinert werden können. Dabei wurden Datenstrukturen (Visualisierung, Visualisierungsschema, -funktion und -klasse) festgelegt und Algorithmen (Generieren von Default- und Schemavisualisierung) entwickelt, die unter den Begriff der Visualisierung als „Festlegung, wie Datenobjekte visuell repräsentiert werden“ fallen. In dem hier beschriebenen dynamischen Teil werden ebenfalls Algorithmen entwickelt, die aber jetzt unter den Begriff der Visualisierung als „Generierung der visuellen Repräsentation von Datenobjekten“ fallen. Dynamisch ist dieser Teil in dem Sinn, daß durch die hier beschriebenen Algorithmen die Visualisierungen „zum Leben erweckt“ werden. Dieses Leben soll neben der visuellen Repräsentation von komplexen Datenobjekten und Fingern auch die Interaktion, d. h. die Manipulation der Datenobjekte und die Navigation mit Fingern umfassen.

4.1 Voraussetzungen

Bereits im ersten Prototyp von ESCHER war die Generierung der visuellen Repräsentation eines komplexen Datenobjektes algorithmisch so umgesetzt, daß nicht sichtbare Sub-Objekte das System nicht unnötig durch Berechnungszeit oder Ressourcen-Anforderungen belasten (anwendungsseitiges *clipping*). Dies hatte einerseits zum Ziel, die Interaktion des Benutzers mit dem Editor möglichst „glatt“[1] ablaufen zu lassen, d. h. die Änderung der visuellen Anzeige als Antwort auf eine Benutzereingabe sollte möglichst schnell ausgeführt werden. Andererseits ist es nicht möglich, die Repräsentationen von potentiell riesigen Datenobjekten im beschränkten Hauptspeicher vollständig zu berechnen, um nur kleine Ausschnitte davon anzuzeigen. Zudem war dieser erste Prototyp in einer Umgebung mit stark eingeschränkten System-Ressourcen implementiert.

[1] Im Sinne des englischen *„smoothly“*, also „verzögerungsfrei“ und „ohne Ruckeln“.

Auch in jüngeren Versionen des Prototyps wird konsequent darauf geachtet, die Repräsentation eines komplexen Datenobjektes möglichst wenig durch nicht sichtbare Sub-Objekte zu verlangsamen. Dies wurde insbesondere nach der Einführung von multimedialen Datentypen und deren graphischer Repräsentation notwendig, um die Interaktionskonzepte des Editors nicht durch inakzeptable Reaktionszeiten unbrauchbar zu machen.

4.1.1 Späte Visualisierung

Der Aspekt der Effizienz muß auch in diesem Ansatz berücksichtigt werden. Der Name **späte Visualisierung** wurde gewählt, weil die Repräsentation eines Sub-Objektes im Sinne einer *lazy evaluation* möglichst spät berechnet werden soll, vorzugsweise erst dann, wenn das Sub-Objekt sichtbar wird. Angenommen, ein komplexes Datenobjekt besteht aus einer sehr großen Menge von Sub-Objekten. Zunächst wird nur ein kleiner Teil dieser Menge dargestellt werden müssen, also auch nur eine kleine Anzahl der Sub-Objekte. Im Verlauf der Interaktion kann der Benutzer, z. B. durch Navigation in der Menge, weitere Sub-Objekte in den sichtbaren Bereich bringen, deren Repräsentationen dann generiert werden müssen.

Aus der Struktur komplexer Objekte und ihrer Visualisierungen läßt sich ableiten, daß bestimmte Eigenschaften der Repräsentation eines komplexen Objektes von den Eigenschaften der Repräsentationen seiner Sub-Objekte abhängen. Insbesondere gilt dieses für die Höhen und Breiten von Repräsentationen, z. B. ist die Höhe einer Kollektion, die durch vertikale Konkatenation dargestellt wird, von der Höhe ihrer Elemente abhängig. Bei der Formulierung der Algorithmen wird ganz allgemein ein Vektor von Eigenschaften $\vec{e}$ betrachtet.

Für die Repräsentation des sichtbaren Teils eines komplexen Objektes müssen in einigen Fällen nicht die Eigenschaften der Repräsentationen aller Sub-Objekte bekannt sein; insbesondere sind die Eigenschaften nicht sichtbarer Sub-Objekte manchmal vernachlässigbar. Es gibt jedoch komplexe Visualisierungsfunktionen, deren generierte Repräsentationen von den Eigenschaften aller, auch der nicht sichtbaren Repräsentationen von Sub-Objekten abhängen. Für diese müssen die Eigenschaften berechnet worden sein, auch wenn die Repräsentationen der Sub-Objekte selbst noch nicht, bzw. noch nicht vollständig, generiert worden sind. Z. B. muß für die Repräsentation des sichtbaren Teils der Kollektion deren Höhe nicht unbedingt bekannt sein. Und damit müssen auch nicht die Höhen aller Elemente, sondern nur die Höhen der sichtbaren Elemente bekannt sein. Soll jedoch die Kollektion mit einem Rollbalken versehen werden, der die Position und Länge des sichtbaren Ausschnitts relativ zur „Größe“ der Kollektion durch die Position und Länge des Balkens im Rollbereich symbolisiert, muß die Höhe der Kollektion bekannt sein.

Um die Bestimmung der Eigenschaften von der Generierung der Repräsentation zumindest teilweise zu entkoppeln, wird die Generierung in zwei Phasen aufgeteilt: In der **Pre-Vi-**

sualisierung[2] wird die Generierung der Repräsentation des Datenobjektes (mindestens) vorbereitet, wobei eine eindeutige **Visualisierungidentifikation** bestimmt und die Eigenschaften der Repräsentation berechnet werden. In der **Post-Visualisierung** wird die Repräsentation fertiggestellt, wozu ggf. die Repräsentationen von Sub-Objekten generiert und verwendet werden. Die möglichst späte Ausführung der Post-Visualisierung soll von der Pre-Visualisierung gesteuert werden.

Etwas konkreter kann angenommen werden, daß ein komplexes Datenobjekt pre-visualisiert wird, indem zunächst ein Behälter für die Aufnahme der Sub-Objekte erzeugt wird. In Tcl/Tk könnten dies z. B. `frame` Widgets [Ous94, S. 158 ff.] sein. In [Kli96, S. 112 f.] heißen sie Behälter-Dialogbausteine, hier werden sie auch Behälter-Widgets genannt. Der „Name“ des Behälters ist dessen Identifikation (siehe Abschnitt 4.1.3).

Sind beide Dimensionen (Höhe und Breite) des Behälters bekannt und ist das Datenobjekt nicht sichtbar, kann die Post-Visualisierung und damit die Pre-Visualisierung der Sub-Objekte auf später verschoben werden, die Pre-Visualisierung des Datenobjektes ist abgeschlossen.

Ist eine Dimension des Behälters unbekannt, d. h. von noch nicht berechneten Eigenschaften der Repräsentationen von Sub-Objekten abhängig, müssen zur Berechnung der unbekannten Dimension die Sub-Objekte pre-visualisiert werden, die Post-Visualisierung wird also sofort ausgeführt. Ein Beispiel für unbekannte Behältergrößen in der Pre-Visualisierung ist die Höhe einer Kollektion, deren Elemente selbst variabel hoch sind.

Ist das komplexe Objekt auch nur teilweise sichtbar, kann sofort die Post-Visualisierung und damit die Pre-Visualisierung der sichtbaren Sub-Objekte ausgeführt werden. Die Formulierung „kann sofort“ wurde gewählt, um deutlich zu machen, daß bei einer sofortigen Ausführung der Post-Visualisierung diese unabhängig von einem ereignisgesteuerten Mechanismus ist; zu einem späteren Zeitpunkt würde sie von einem Sichtbarkeitsereignis ausgelöst werden.

In Tabelle 4.1 ist die Abhängigkeit des Post-Visualisierungs-Zeitpunktes von Dimensionenbekanntheit und Objektsichtbarkeit zusammengefaßt. In Abb. 4.1 auf der nächsten Seite ist die späte Visualisierung an einem Beispiel erläutert.

	Objekt sichtbar	Objekt nicht sichtbar
alle Dimensionen bekannt	kann sofort erfolgen	kann später erfolgen
eine Dimension unbekannt	muß sofort erfolgen	muß sofort erfolgen

Tabelle 4.1: Die Post-Visualisierung in Abhängigkeit von Objektsichtbarkeit und Kenntnis der Dimensionen

[2] Im Deutschen müßte diese Phase eigentlich „Prä-Visualisierung“ genannt werden. Die englische Form „Pre“ der deutschen Vorsilbe „Prä“ (lat. für vor, vorher, vorne) hat sich in der Informatik aber eingebürgert. Das Präfix „Pre“ kann auch als Abkürzung des englischen Wortes *preliminary* (dt. vorbereitend, vorläufig) verstanden werden.

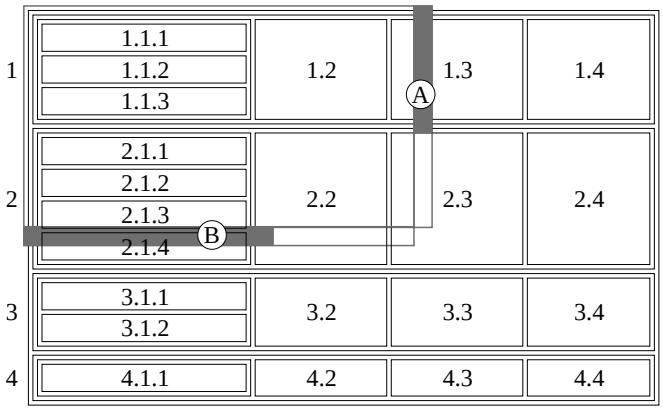


Abbildung 4.1: Späte Visualisierung

Beispiel 4.1 Späte Visualisierung

Die Abbildung zeigt schematisch die Repräsentation eines Datenobjektes, das aus einer Menge von Tupeln (bezeichnet mit 1, 2, 3 und 4) besteht. Dieb Tupel haben jeweils vier Komponenten (bezeichnet mit 1.1, 1.2, ..., 2.1, 2.2, ...); die erste Komponente ist jeweils eine Menge mit atomaren Elementen (bezeichnet mit 1.1.1, 1.1.2, ..., 2.1.1, 2.1.2, ...). Das Datenobjekt ist in einem vertikal und horizontal rollbaren Viewport dargestellt.

Um die Länge der Rollbalken Ⓐ und Ⓑ zu bestimmen, müssen Höhe und Breite der Repräsentation bekannt sein. Die Breite der Repräsentation ist datenunabhängig und daher bereits bekannt, die Höhe ist durch die Höhe der äußeren Menge bestimmt. Deren Höhe hängt von den Höhen der Tupel ab, die wiederum von den Höhen der inneren Mengen bestimmt sind. Die atomaren Elemente der inneren Mengen haben feste Höhe, daher können die Höhen der inneren Mengen berechnet werden, ohne die Elemente zu visualisieren.

Für die Generierung der Repräsentation des Datenobjektes wird die äußere Menge pre-visualisiert, also ein Mengenbehälter erzeugt. Dessen Breite ist bekannt, nicht aber seine Höhe. Daher wird die Post-Visualisierung der äußeren Menge, und damit die Pre-Visualisierung der Mengenelemente (d. h. der Tupel), ausgeführt. Deren Höhe ist ebenfalls unbekannt, daher wird jeweils nach der Tupel-Behältererzeugung die Post-Visualisierung der Tupel, und damit die Pre-Visualisierung der Tupelkomponenten, ausgeführt. Die ersten Tupelkomponenten sind jeweils die inneren Mengen. Deren Visualisierungshöhe kann aus der Elementhöhe und -anzahl berechnet werden (siehe Abschnitt 5.8.1), also muß die Post-Visualisierung nur für sichtbare Objekte (1.1 und 2.1) gleich ausgeführt werden. Die Post-Visualisierung der zur Zeit nicht sichtbaren Objekte (3.1 und 4.1) kann auf später verschoben werden. Für die anderen Tupelkomponenten gilt auch, daß nur die sichtbaren Objekte (1.2, 1.3, 2.2 und 2.3) sofort post-visualisiert werden. Alle weiteren Objekte (1.4, 2.4 sowie die Sub-Objekte von 3 und 4) können später post-visualisiert werden. □

Diese Aufteilung der Generierung von Repräsentationen in zwei Phasen sollen alle Visualisierungsklassen reflektieren, indem ihre Schnittstellen jeweils eine Pre- und Post-Methode umfassen. Eine genauere Beschreibung dieser Methoden und ihrer Aufgaben wird weiter unten angegeben (siehe Abschnitte 4.2.1 bis 4.2.3). Die zweite Phase soll möglichst spät stattfinden, wobei der genaue Zeitpunkt von den einzelnen Visualisierungsfunktionen durch ihre Implementierung selbst bestimmt werden kann. Im einfachsten Fall wird eine komplexe Visualisierungsfunktion die Pre-Visualisierung aller, auch nicht sichtbaren, Sub-Objekte zulassen, um den von der Pre-Visualisierung realisierten Mechanismus der späten Post-Visualisierung zu nutzen. Aufwendigere Implementierungen, die insbesondere „kleine" Ausschnitte „großer" Objekte effizienter visualisieren, werden zunächst nur die Pre-Visualisierung der sichtbaren Sub-Objekte initiieren und eigene Mechanismen für die spätere, bedarfsgesteuerte Generierung weiterer Repräsentationen von Sub-Objekten realisieren.

4.1.2 Sichtbarer Ausschnitt und aktuelle Visualisierungsposition

Die „Sichtbarkeit" von Objekten bzw. der „sichtbare Ausschnitt" wurde bereits öfter angesprochen. Diese Begriffe sollen nun näher erläutert werden. Von außen kommend, d. h. aus der Sicht des Benutzers eines Fenstersystems, ist der sichtbare Ausschnitt durch das Fenster, das die Repräsentation enthält, definiert. Dieser ist durch maximale Abmessungen, z. B. die Bildschirmgröße, begrenzt, so daß keine beliebig großen Datenobjekte vollständig dargestellt werden können. Weiter nach innen gehend, d. h. in die Repräsentation eines komplexen Datenobjektes und seiner Sub-Objekte hinein, wird der sichtbare Ausschnitt durch die Repräsentation des jeweils umgebenden Datenobjektes definiert.

Visualisierungsfunktionen können bzgl. ihrer Behandlung des sichtbaren Ausschnitts in zwei Klassen unterteilt werden, wobei diese Klassifizierung unabhängig von der Klassenhierarchie aus Abschnitt 3.3.1 ist. Die einen schränken den sichtbaren Ausschnitt nicht ein, d. h. ihre Repräsentationen müssen den von ihnen beanspruchten Platz bekommen. Dazu gehören z. B. die Default-Visualisierungsfunktionen für Tupel, Kollektionen und atomare Daten. Die anderen schränken den sichtbaren Ausschnitt, vertikal, horizontal, oder in beiden Richtungen ein und erlauben die Veränderung der Lage der Repräsentation im sichtbaren Ausschnitt.

Mit dem Begriff **Viewport** wird der Bereich bezeichnet, den eine Visualisierungsfunktion als ihren sichtbaren Ausschnitt ansieht. Er wird von der Repräsentation des jeweils umgebenden Datenobjektes bzw. dem Fenstersystem für das ganze Datenobjekte bestimmt. Einige Konstruktions-Visualisierungsfunktionen sind ausschließlich zur Einschränkung des sichtbaren Ausschnitts konzipiert; diese werden auch **Viewport-Visualisierungsfunktionen** genannt.

Nachdem der Begriff des sichtbare Ausschnitt erklärt ist, stellt sich die Frage: Woher weiß ein Datenobjekt, ob es sichtbar ist? Oder anders gefragt: Wie wird bestimmt, ob die Repräsentation eines Datenobjekt im Viewport liegt? Bei der Darstellung von graphischen

Objekten wird von einem Koordinatensystem ausgegangen, das links oben seinen Ursprung hat, und nach rechts (X-Achse, Breite), bzw. unten (Y-Achse, Höhe), wächst. Die Plazierung der Repräsentationen von Datenobjekten wird relativ zum Ursprung des Viewports von der „aktuellen Visualisierungsposition“ (im Folgenden auch einfach Offset genannt) bestimmt. Nach der Plazierung der Repräsentation eines Sub-Objektes durch eine Visualisierungsfunktion muß von dieser entschieden werden, ob die Repräsentation eines nächsten und wenn ja welches Sub-Objektes generiert und integriert werden soll. Außerdem muß der Offset so verändert werden, daß er die korrekte relative Lage zum Ursprung reflektiert.

Auch Visualisierungsfunktionen, die den sichtbaren Ausschnitt nicht wirklich einschränken, können den Viewport verkleinern. Sie müssen dann auch den Offset entsprechend aktualisieren und können so Repräsentationen von Sub-Objekten relativ zu einem „logischen Koordinatenursprung“ plazieren.

Beispiel 4.2 Viewport und Offset

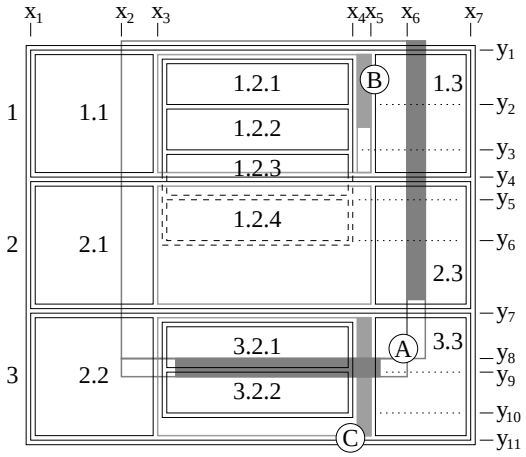


Abbildung 4.2: Beispiel zu Viewport und Offset

In Abb. 4.2 ist wiederum schematisch die Repräsentation eines Datenobjektes dargestellt. Das Datenobjekt besteht aus einer Menge von Tupeln (bezeichnet mit 1, 2 und 3); die Tupel haben jeweils drei Komponenten (bezeichnet mit 1.1, 1.2, 1.3, 2.1, ...); die zweite Komponente ist jeweils eine Menge mit atomaren Elementen (bezeichnet mit 1.2.1, 1.2.2, ..., 3.2.1, 3.2.2). Tupel und Mengen werden mit den Default-Visualisierungsfunktionen dargestellt (siehe Kapitel 5). Das Datenobjekt ist in einem vertikal und horizontal rollbaren Viewport Ⓐ, die inneren Mengen sind jeweils in einem vertikal rollbaren Viewport (Ⓑ und Ⓒ) dargestellt. Auf die Darstellung der inneren Menge des Tupels 2 wurde wegen der Übersichtlichkeit verzichtet. Die Offsets im Koordinatensystem sind mit x_1, x_2, ... (horizontale Achse) und y_1, y_2, ... (vertikale Achse) bezeichnet. Das Beispiel wird in der folgenden Aufzählung fortgesetzt. □

- Für die Tupel wird nach jeder Plazierung einer Komponente der Offset auf der X-Achse um die Visualisierungsbreite der Komponente und die Breite der vertikalen

Linie erhöht. Liegt der aktualisierte Offset noch im Viewport, muß die nächsten Komponente pre-visualisiert werden. Z. B. wird nach der Plazierung der Komponente 1.1 der X-Offset um $x_3 - x_1$ erhöht, nach der Plazierung der inneren Menge 1.2 in ihrem Viewport wird der X-Offset um $x_5 - x_3$ erhöht.

- Für die Kollektionen wird nach jeder Plazierung eines Elementes der Offset auf der Y-Achse um die Visualisierungshöhe des Elementes und die Höhe der horizontalen Linie erhöht. Liegt der aktualisierte Offset noch im Viewport, muß die nächste Komponente pre-visualisiert werden. Z. B. wird nach der Plazierung des Elements 1.2.1 der Y-Offset um $y_2 - y_1$ erhöht. Nach der Plazierung des Elements 1.2.3 wird der Y-Offset um $y_5 - y_3$ erhöht; der resultierende Y-Offset ist größer als die Höhe des Viewports $(y_4 - y_1)$, so daß das Element 1.2.4 nicht mehr pre-visualisiert werden muß.

- Wird eine Matrix dargestellt, indem die Elemente spalten- und zeilenweise angeordnet werden, wird nach jeder Plazierung eines Elementes der Offset auf der X-Achse um die Visualisierungsbreite des Elementes erhöht. Liegt der aktualisierte Offset noch im Viewport, muß die nächste Komponente der aktuellen Zeile pre-visualisiert werden. Liegt der aktualisierte Offset nicht mehr im Viewport, kann die nächste Komponente der aktuellen Zeile später pre-visualisiert werden; dann wird der Offset auf der X-Achse zurückgesetzt und auf der Y-Achse um die Visualisierunghöhe des Elementes erhöht. Liegt der jetzige aktualisierte Offset noch im Viewport, muß das erste Element der nächsten Zeile pre-visualisiert werden.

- Viewports schränken den sichtbaren Ausschnitt ein. Im Beispiel hat Viewport Ⓐ eine Breite von $x_6 - x_2$ und eine Höhe von $y_8 - y_1$. Dieser Viewport legt den sichtbaren Bereich für die äußere Menge fest. Viewport Ⓑ hat eine Breite von $x_4 - x_3$, eine Länge von $y_6 - y_1$ und definiert den sichtbaren Bereich der inneren Menge 1.2. Viewport Ⓒ liegt teilweise außerhalb von Viewport Ⓐ, so daß der sichtbare Bereich der Menge 3.2 nur eine Höhe von $y_8 - y_7$ hat.

Die Festlegung des Viewports und des initialen Offsets ist eine Aufgabe der Pre-Methode von Visualisierungsfunktionen (siehe Abschnitt 4.2.1). Das Fortschreiben des Offsets für Tupel Kollektionen übernimmt die Sub-Methode der entsprechenden Visualisierungsfunktionen (siehe Abschnitt 4.2.2), die Entscheidung über das nächste zu visualisierende Sub-Objekt wird von den Iterate-Methoden getroffen (ebenfalls Abschnitt 4.2.2). Werden nicht alle Sub-Objekte pre-visualisiert, muß die Visualisierungsfunktion selbst sicherstellen, daß die Repräsentation der Sub-Objekte bei Bedarf generiert werden.

4.1.3 Visualisierungsidentifikation

Eine Teilaufgabe der Pre-Visualisierung ist die Bestimmung einer **Visualisierungsidentifikation**. Dieser Begriff wurde bewußt „ungenau“ formuliert, um ein möglichst flexibles

Konzept zu erhalten. Sind keine Verwechslungen möglich, werden Visualisierungsidentifikationen im Folgenden kurz mit „**Identifikation**“ bezeichnet.

Abhängig von der Umsetzung sind hier verschiedene Möglichkeiten denkbar, z. B.:

- X Windows ordnet jedem Widget einen eindeutigen Identifikator (*X window identifier*) zu [Jon88],
- in Microsoft Windows gibt es einen *HWND* genannten Fenster-Handle [Sch93, Mic93], und
- Tcl/Tk identifiziert seine Widgets durch eindeutige, hierarchische Pfadnamen. Innerhalb eines `canvas` Widget werden einzelne graphische Elemente durch Elementidentifikatoren (*item ids*) identifiziert [Ous94, S. 209 ff.].

Die ersten beiden Beispiele beziehen sich auf eine Implementierung mittels einer direkten Programmierung des Fenstersystem-APIs. Am letzten Beispiel wird deutlich, daß auch heterogene Identifikationen möglich sein können; außerdem ist die Generierung von Repräsentationen nicht an eine Realisierung durch einen Baum von Behälter-Widgets (*widget tree*) gebunden, sondern kann z. B. auch durch graphische Objekte in einer sog. „Drawing-Area“ erstellt werden.

Die Identifikation ist immer eindeutig einem Paar (o, v), bestehend aus einem Datenobjekt o und einem Visualisierungsknoten v, zugeordnet. Weder das Datenobjekt noch der Visualisierungsknoten sind allein ausreichend bzw. eindeutig: Konstruktions-Visualisierungsfunktionen haben kein „eigenes“ Datenobjekt; sie werden über das Datenobjekt der Sub-Visualisierungsfunktion identifiziert. Alle Elemente einer Kollektion sind verschiedene Datenobjekte, die entsprechenden Visualisierungsidentifikationen werden aber über denselben Visualisierungsknoten zugeordnet.

4.2 Generierung von Repräsentationen

In diesem Abschnitt wird die von Visualisierung und Datenobjekt gesteuerte Generierung von Repräsentationen durch Pre- und Post-Visualisierung näher beschrieben. Die dazu notwendigen Algorithmen, die sich wechselseitig rekursiv ausführen, werden formuliert. Die Pre- und Post-Visualisierung werden durch die Methoden Pre bzw. Post eines Visualisierungsknotens realisiert. Da die Generierung von Repräsentationen von Datenobjekt *und* Visualisierung gesteuert wird, haben diese beide Visualisierungsknoten-Methoden mindestens das Datenobjekt als Parameter.

4.2.1 Pre-Visualisierung

Aufgabe der Pre-Visualisierung ist es, die Generierung der Repräsentation eines Datenobjektes vorzubereiten und die Eigenschaften der Repräsentation zu berechnen. Kann eine Visualisierungsfunktion die Eigenschaften der Repräsentation durch die Ausführung der Pre-Methode berechnen und ist das Datenobjekt zunächst nicht sichtbar, wird die Post-Visualisierung mit einem geeigneten Mechanismus auf einen späteren Zeitpunkt verschoben. Der Zeitpunkt der Post-Visualisierung ist dann von den nachfolgenden Benutzerinteraktionen abhängig. Wenn eine Visualisierungsfunktion die Eigenschaften der Repräsentation nicht durch die Ausführung ihrer Pre-Methode berechnen kann oder das Datenobjekt auch nur teilweise sichtbar ist, wird die Post-Visualisierung direkt nach der Pre-Visualisierung durchgeführt. Die Post-Visualisierung liefert die Eigenschaften als Ergebnis, so daß sie auch als Ergebnis der Pre-Visualisierung zur Verfügung stehen. In der Pre-Visualisierung wird also auf jeden Fall die Post-Visualisierung initiiert, indem sie entweder direkt ausgeführt oder zur späteren Ausführung vorbereitet wird.

Durch die Pre-Methode werden außerdem Viewport und Offset neu bestimmt. Diese sind aber nur für komplexe Visualisierungsknoten bzw. deren Nachfolgerknoten von Bedeutung und werden in dem Aufruf der Post-Methoden weitergereicht.

Methode Pre

Klasse:
visNode

Objekt:
Visualisierungsknoten v

Parameter:
Datenobjekt o, Viewport Π, Offset Ω

Berechnung:

1. Bestimme die Identifikation id von (o, v).
2. Bestimme die Visualisierungsfunktion f von v.
3. Führe die Pre-Methode von f mit den Parametern o, v, id, Π und Ω aus und bestimme wenn möglich die Eigenschaften $\vec{e}$ von id, sowie, falls $f \in$ cplxVF, Viewport Π' und Offset Ω'.
4. Wenn nicht alle Eigenschaften berechnet werden konnten oder id mindestens teilweise sichtbar ist, dann
   - 4.1. Führe die Post-Methode von v mit den Parametern o, id, Π' und Ω' aus und bestimme die Eigenschaften $\vec{e}$ von id,
   - 4.2. sonst initiiere für v die späte Post-Visualisierung.

Rückgabe:
Identifikation id, Eigenschaften $\vec{e}$

Der oben erwähnte „geeignete Mechanismus“ kann in einer Implementierung auf Basis von Tcl/Tk z. B. realisiert werden, indem an das Sichtbarkeitsereignis `<Expose>` für das *id* entsprechende Behälter-Widget ein Skript gebunden wird, das die Post-Methode mit den Parametern o, *id*, Π' und Ω' ausführt.

4.2.2 Post-Visualisierung

Für jede der vier Visualisierungsklassen (siehe Abschnitt 3.3) müssen in der Post-Visualisierung verschiedene Aktionen durchgeführt werden. Insbesondere müssen für Kollektions- und Tupel-Visualisierungsfunktionen die jeweiligen Sub-Objekte und Nachfolgerknoten in der Visualisierung bestimmt werden. Während der Post-Visualisierung wird die Repräsentation eines Datenobjektes durch den Aufruf der Post-Methode der Visualisierungsfunktion fertiggestellt. Sie wird für komplexe Datenobjekte durch die Pre-Visualisierung von Elementen (für Kollektionen) bzw. Komponenten (für Tupel) und/oder durch den Aufruf von Sub-Methoden (für Konstruktionen, Kollektionen und Tupel) vorbereitet.

Während der Post-Visualisierung werden durch die jeweiligen Visualisierungsfunktionen-Methoden Post die Eigenschaften der generierten Repräsentationen bestimmt, so daß diese als Ergebnis der Visualisierungsknoten-Methode Post in der Pre-Visualisierung zur Verfügung stehen. Die Visualisierungsfunktionen-Methoden werden ohne die Parameter Viewport und Offset aufgerufen, da diese bereits bei der Pre-Visualisierung berücksichtigt wurden.

Im Folgenden werden die Aktionen vorgestellt, die für die unterschiedlichen Visualisierungsklassen jeweils durchzuführen sind. Dabei wird Bezug auf die weiter unten folgende Beschreibung der Visualisierungsknoten-Methode Post.

atomVF: Die Visualisierungsfunktionen-Methode Post wird mit den während der Pre-Visualisierung bestimmten Parametern Datenobjekt, Visualisierungsknoten und Identifikation aufgerufen.

constrVF: Zunächst wird die Pre-Visualisierung des Datenobjektes ausgeführt, wozu der Nachfolgerknoten in der Visualisierung bestimmt werden muß. Ergebnis sind die Identifikation und Eigenschaften der Repräsentation des Datenobjektes. Dann wird die Sub-Methode der Konstruktions-Visualisierungsfunktion aufgerufen, um die Repräsentation des Datenobjektes zu integrieren.

Schließlich wird die Visualisierungsfunktionen-Methode Post aufgerufen, um die Repräsentation fertigzustellen. An sie werden die während der Pre-Visualisierung bestimmten Eigenschaften der Repräsentation des Datenobjektes übergeben, um die Eigenschaften der durch die Konstruktions-Visualisierungsfunktion generierten Repräsentation berechnen zu können.

tplVF: Unter Verwendung der First- und Iterate-Methoden der Tupel-Visualisierungsfunktion wird zunächst über die zu visualisierenden Komponenten iteriert und jeweils

- der zugehörige Nachfolgerknoten und das entsprechende Sub-Objekt bestimmt,
- die Pre-Visualisierung ausgeführt und
- die Sub-Methode der Tupel-Visualisierungsfunktion aufgerufen.

Der Viewport bei der Pre-Visualisierung der Komponenten ist wiederum wie während der Pre-Visualisierung des Tupels bestimmt. Der Offset wird jedoch während der Iteration so fortgeschrieben, daß er die Visualisierungsposition der entsprechenden Komponente reflektiert. Die First-Methode stellt sicher, daß der Offset für die Pre-Visualisierung der ersten Komponente korrekt ist; die Iterate-Methode hat die Aufgabe, den Offset für die Pre-Visualisierung der jeweils nächsten Komponente zu ändern. Ergebnis der Komponenten-Pre-Visualisierungen sind wiederum jeweils die Identifikation und Eigenschaften der generierten Repräsentationen.

Die Sub-Methode der Tupel-Visualisierungsfunktion integriert die Repräsentationen der Komponenten. Sie gibt jeweils die Größe der Komponenten-Repräsentation innerhalb der Tupel-Repräsentation zurück, welche an die Iterate-Methode zur Offset-Berechnung weitergegeben wird. Diese Größe muß nicht mit der Größe übereinstimmen, die für die Repräsentation einer Komponenten in deren Pre-Visualisierung berechnet wurde. Z. B. berücksichtigt die Tupel-Visualisierungsfunktion tplLabel (siehe Abschnitt 5.7.2) zusätzlich die Höhe des Attributnamens für jede Komponente.

Neben der Fortschreibung des Offsets hat die Iterate-Methode zusätzlich die Aufgabe, zu bestimmen, ob eine und wenn ja welche weitere Komponente pre-visualisiert werden muß. Die Entscheidung wird dabei aufgrund der Nachfolgerknoten getroffen, denn nicht jede Komponente des Datenobjektes muß in der Visualisierung definiert sein, und die Reihenfolge der Komponenten in der Repräsentation kann von der durch das Schema definierten Reihenfolge der Komponenten verschieden sein. Eine weitere Entscheidungsgrundlage kann das Verhältnis von Viewport, Offset, und Visualisierungsgrößen der Komponenten sein. Letztere kann aus deren Eigenschaften, die während der Pre-Visualisierung ermittelt wurden, abgeleitet werden. Liegt z. B. der neue Offset, berechnet aus dem alten Offset und der Größe, außerhalb des Viewports, kann eine Visualisierungsfunktion auf die Pre-Visualisierung weiterer Komponenten verzichten. Aus den beiden eben genannten Entscheidungsgrundlagen läßt sich folgern, daß auch die erste Komponente, die Pre-Visualisiert werden soll, von der Visualisierungsfunktion bestimmt werden muß. Dies wird von der First-Methode getan, die zu Beginn der Iteration aufgerufen wird.

Abschließend wird wiederum die Post-Methode der Tupel-Visualisierungsfunktion ausgeführt. Das aus der Differenz des zuletzt berechneten Offsets und des Offsets bei Aufruf der Post-Visualisierung berechnete **Visualisierungs-Delta** wird neben den Parametern Datenobjekt, Visualisierungsknoten und Identifikation als Eigenschaft übergeben. Dies ist insbesondere für Tupel-Visualisierungsfunktionen wichtig, die variabel hohe (und/oder variabel breite) Sub-Visualisierungsfunktionen haben. Für

solche kann erst aus diesem Delta die tatsächliche Höhe (und/oder Breite) festgelegt werden.

clnVF: zunächst wird unterschieden, ob das zu visualisierende Datenobjekt eine nullwertige, eine leere, oder eine nicht-leere Kollektion ist[3]. In den beiden ersten Fällen werden die speziellen Null- und Empty-Methoden der Kollektions-Visualisierungsfunktionen aufgerufen. Diese Methoden aktualisieren ebenfalls den Offset, da vor der Ausführung der Visualisierungsfunktionen-Methode Post wie bei tplVF das Visualisierungs-Delta berechnet wird.

Für nicht-leere Kollektionen wird analog tplVF eine Iteration durchgeführt. Dabei bleibt jedoch der die Repräsentation der Elemente definierende Nachfolgerknoten konstant und wird vor der Iteration bestimmt. Es wird über die Elemente des Datenobjektes iteriert, wobei auch hier die Reihenfolge und Anzahl der generierten Repräsentationen für Elemente von der Kollektions-Visualisierungsfunktion abhängen. Für jedes zu visualisierende Element wird

- das Sub-Objekt) bestimmt,
- dessen Pre-Visualisierung ausgeführt und
- die Sub-Methode der Kollektions-Visualisierungsfunktion aufgerufen.

Der Mechanismus der Offset-Fortschreibung ist genau der gleiche wie bei tplVF, die Bemerkungen zu den First-, Iterate- und Post-Methoden gelten analog.

Schritt 1 der folgenden Methode stellt die Verbindung zwischen Benutzeroperationen und generierten Repräsentationen her. Die Diskussion der Interaktion findet in Abschnitt 4.3 statt, wo auch die Visualisierungsknoten-Methode Bind beschrieben wird.

Methode Post

Klasse:
visNode

Objekt:
Visualisierungsknoten v

Parameter:
Datenobjekt o, Identifikation id, Viewport Π, Offset Ω

Berechnung:

1. Führe die Bind-Methode von v mit dem Parameter id, aus und installiere die Bindungen für v.
2. Bestimme die Visualisierungsfunktion f von v.
3. Wenn f eine atomare Visualisierungsfunktion ist ($f \in$ atomVF), dann

[3]Diese Unterscheidung ist für tplVF nicht notwendig, da Tupel im verwendeten Datenmodell ESCHER$^+$ weder nullwertig noch leer sein können (siehe Abschnitt 2.3.1).

3.1. führe die Post-Methode von f mit den Parametern o, v und id aus und bestimme die Eigenschaften $\vec{e}$ der generierten Repräsentation von o.

4. Wenn f eine Konstruktions-Visualisierungsfunktion ist ($f \in$ constrVF), dann:

4.1. Bestimme den Nachfolgerknoten v_{sub} von v.

4.2. Führe die Pre-Methode von v_{sub} mit den Parametern o, Π und Ω aus und bestimme Identifikation id_{sub} und Eigenschaften $\vec{e}_{sub}$ der Repräsentation.

4.3. Führe die Sub-Methode von f mit den Parametern o, v, id, v_{sub}, id_{sub}, Π, Ω und $\vec{e}_{sub}$ aus.

4.4. Führe die Post-Methode von f mit den Parametern o, v, id und $\vec{e}_{sub}$ aus und bestimme die Eigenschaften $\vec{e}$ der generierten Repräsentation.

5. Wenn f eine Tupel-Visualisierungsfunktion ist ($f \in$ tplVF), dann:

5.1. Führe die First-Methode von f mit den Parametern o, v, id, Π und Ω aus und bestimme den Nachfolgerknoten v^1_{sub} der im Sinne von f ersten zu visualisierenden Komponente von v und den aktualisierten Offset Ω^1.

5.2. Solange eine Komponente zu visualisieren ist:

5.2.1. Bestimme das v^i_{sub} entsprechende Sub-Objekt o^i_{sub}.

5.2.2. Führe die Pre-Methode von v^i_{sub} mit den Parametern o^i_{sub}, Π und Ω^i aus und bestimme Identifikation id^i_{sub} und Eigenschaften $\vec{e}^{\,i}_{sub}$ der Repräsentation von o^i_{sub}.

5.2.3. Führe die Sub-Methode von f mit den Parametern o, v, id, o^i_{sub}, v^i_{sub}, $id^{\,i}_{sub}$, Ω^i und $\vec{e}^{\,i}_{sub}$ aus und bestimme die Größe s^i von id^i_{sub} in id.

5.2.4. Führe die Iterate-Methode von f mit den Parametern o, v, id, o^i_{sub}, v^i_{sub}, $id^{\,i}_{sub}$, Π, Ω^i und s^i aus und bestimme den Nachfolgerknoten v^{i+1}_{sub} der im Sinne von f nächsten zu visualisierenden Komponente von v und den aktualisierten Offset Ω^{i+1}.

5.3. Bestimme das Visualisierungs-Delta $\vec{\Delta} = \Omega^n - \Omega$ aus der Differenz zwischen aktuellem und initialem Offset.

5.4. Führe die Post-Methode von f mit den Parametern o, v, id und $\vec{\Delta}$ aus und bestimme die Eigenschaften $\vec{e}$ der generierten Repräsentation von o.

6. Wenn f eine Kollektions-Visualisierungsfunktion ist ($f \in$ clnVF), dann:

6.1. Falls o eine nullwertige Kollektion ist, führe die Null-Methode von f mit den Parametern o, v, id, Π und Ω aus und bestimme den aktualisierten Offset Ω^1.

6.2. Falls o eine leere Kollektion ist, führe die Empty-Methode von f mit den Parametern o, v, id, Π und Ω, aus und bestimme den aktualisierten Offset Ω^1.

6.3. Falls o weder eine nullwertige noch eine leere Kollektion ist:

6.3.1. Bestimme den Nachfolgerknoten v_{sub} von v.

6.3.2. Führe die First-Methode von f mit den Parametern o, v, id, Π und Ω aus und bestimme das im Sinne von f erste zu visualisierende Element (Sub-Objekt) o^1_{sub} von o und den aktualisierten Offset Ω^1.

6.3.3. Solange eine Element zu visualisieren ist:

6.3.3.1. Führe die Pre-Methode von v_{sub} mit den Parametern o^i_{sub}, Π und Ω^i aus und bestimme Identifikation id^i_{sub} und Eigenschaften $\vec{e}^{\,i}_{sub}$ der Repräsentation von o^i_{sub}.

6.3.3.2. Führe die Sub-Methode von f mit den Parametern o, v, id, o^i_{sub}, v_{sub}, id^i_{sub}, Ω^i und $\vec{e}^{\,i}_{sub}$ aus und bestimme die Größe s^i von id^i_{sub} in id.

6.3.3.3. Führe die Iterate-Methode von f mit den Parametern o, v, id, o^i_{sub}, v^i_{sub}, $id^{\,i}_{sub}$, Π, Ω^i und s^i aus und bestimme das im Sinne von f nächste zu visualisierende Element (Sub-Objekt) o^{i+1}_{sub} von o und den aktualisierten Offset Ω^{i+1}.

6.4. Bestimme das Visualisierungs-Delta $\vec{\Delta} = \Omega^n - \Omega$ aus der Differenz zwischen aktuellem und initialem Offset.

6.5. Führe die Post-Methode von f mit den Parametern o, v, id und $\vec{\Delta}$ aus und bestimme die Eigenschaften $\vec{e}$ der generierten Repräsentation von o.

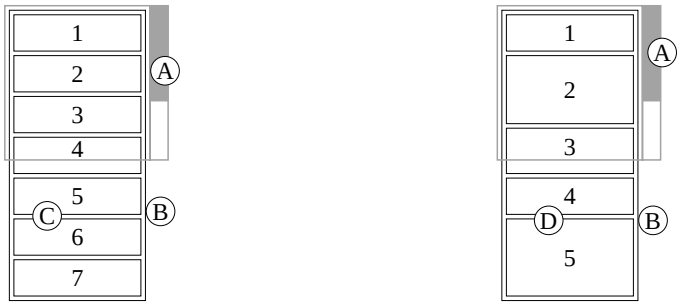
Rückgabe:
Eigenschaften $\vec{e}$

In Abb. 4.3 auf der gegenüberliegenden Seite wird ein weiteres Beispiel betrachtet, um den verzahnten, wechselseitig rekursiven Aufruf der Visualisierungsknoten-Methoden Pre und Post zu verdeutlichen. Insbesondere wird noch einmal auf die Auswirkung variabler Längen von Sub-Visualisierungsfunktionen für die Pre- und Post-Visualisierung von komplexen Datenobjekten eingegangen.

4.2.3 Visualisierungsmethoden von Visualisierungsfunktionen

Nachdem die Methoden Pre und Post für Visualisierungsknoten formuliert wurden, können die Schnittstellen der Visualisierungsfunktionen, soweit sie die Pre- bzw. Post-Visualisierung betreffen, beschrieben werden.

Allen Klassen gemeinsam sind nur die Methoden Pre und Post. Komplexe Visualisierungsfunktionen integrieren durch Sub-Visualisierungsfunktionen generierte Repräsentationen und haben daher eine Sub-Methode. Tupel- und Kollektions-Visualisierungsfunktionen haben zusätzlich First- und Iterate-Methoden. Schließlich haben Kollektions-Visualisierungsfunktionen noch Null- und Empty-Methoden für die Generierung von Repräsentationen nullwertiger und leerer Kollektionen. In Tabelle 4.2 auf Seite 138 ist eine Übersicht der Visualisierungsmethoden für die einzelnen Visualisierungsklassen dargestellt.



(a) Feste Höhen (b) Variable Höhen

Abbildung 4.3: Pre- und Post-Visualisierung

Beispiel 4.3 Pre- und Post-Visualisierung

In der Abbildung sind schematisch zwei Versionen der Repräsentation einer Menge Ⓑ (Visualisierungsfunktion cln, siehe Abschnitt 5.8.1) mit atomaren Elementen (Ⓒ bzw. Ⓓ) in einem vertikal rollbaren Viewport Ⓐ (Visualisierungsfunktion scrollV, siehe Abschnitt 5.5.1) dargestellt. In Abb. 4.3(a) haben die Elemente Ⓒ feste Höhen (Visualisierungsfunktion atom, siehe Abschnitt 5.3.1), in Abb. 4.3(b) sind die Elemente Ⓓ variabel hoch (Visualisierungsfunktion text, siehe Abschnitt 5.3.2).

Bei der Generierung der Repräsentation wird zunächst die Pre-Visualisierung für den Viewport ausgeführt. Dessen Größen sind bekannt; er ist aber sichtbar, so daß die Post-Visualisierung sofort ausgeführt wird. Dadurch wird die Pre-Visualisierung für die Menge ausgeführt. Haben deren Elemente feste Höhe, könnte die Höhe der Menge berechnet werden, ohne die Post-Visualisierung auszuführen. Da die Menge sichtbar ist, wird in beiden Fällen sofort die Post-Visualisierung ausgeführt, wodurch die Elemente pre-visualisiert werden. Von den beiden Pre-Methoden der atomaren Visualisierungsfunktionen atom und text können die Elementhöhen jeweils berechnet werden, so daß nur für die sichtbaren Elemente (links 1–4, rechts 1–3) die Post-Visualisierungen ausgeführt werden. Die aus den Pre-Visualisierungen der Elemente berechneten Visualisierungs-Deltas werden in der Post-Visualisierung der Menge verwendet. Deren Visualisierungs-Delta wird wiederum in der Post-Visualisierung des Viewports benutzt. □

Methode Pre

Klasse:
visFunc
Objekt:
Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id, Viewport Π, Offset Ω
Aufgabe:
Bereite die Generierung der Repräsentation von o mit der Identifikation id von (o, v) gemäß f vor. Falls notwendig, beachte dabei Π und Ω. Versuche, die Eigenschaften von id zu bestimmen. Aktualisiere ggf. Π und Ω.
Rückgabe:
Eigenschaften $\vec{e}$ von id und Boole'scher Wert, der angibt, ob alle Eigenschaften bestimmt werden konnten

Methode Post

Klasse:
atomVF
Objekt:
atomare Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id
Aufgabe:
Stelle die Repräsentation von o gemäß f fertig und bestimme ihre Eigenschaften $\vec{e}$.
Rückgabe:
Eigenschaften $\vec{e}$ von id

Methode Post

Klasse:
constrVF
Objekt:
Konstruktions-Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id, Eigenschaften $\vec{e}_{sub}$
Aufgabe:
Stelle die Repräsentation o gemäß f fertig und bestimme ihre Eigenschaften $\vec{e}$. Falls notwendig, beachte dabei $\vec{e}_{sub}$.
Rückgabe:
Eigenschaften $\vec{e}$ von id

Methode Post

Klasse:
cplxDataVF

Objekt:
Kollektions- oder Tupel-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id, Visualisierungs-Delta $\vec{\Delta}$

Aufgabe:
Stelle die Repräsentation von o gemäß f fertig und bestimme ihre Eigenschaften. Falls notwendig, beachte dabei $\vec{\Delta}$ (Differenz zwischen Offset nach Aufruf der Pre-Methode und nach letztem Aufruf der Sub-Methode).

Rückgabe:
Eigenschaften $\vec{e}$ von id

Methode Sub

Klasse:
constrVF

Objekt:
Konstruktions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Nachfolgerknoten v_{sub}, Identifikation id_{sub} von (o, v_{sub}), Viewport Π, Offset Ω, Eigenschaften $\vec{e}_{sub}$

Aufgabe:
Integriere id_{sub} in id.

Rückgabe:
—

Methode Sub

Klasse:
cplxDataVF

Objekt:
Tupel- oder Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Sub-Objekt o_{sub}, Nachfolgerknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub}), Viewport Π, Offset Ω, Eigenschaften $\vec{e}_{sub}$

Aufgabe:
Integriere id_{sub} in id und aktualisiere Ω.

Rückgabe:
Größe s von id_{sub} in id

Methode First

Klasse:
tplVF
Objekt:
Tupel-Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id, Viewport Π, Offset Ω
Aufgabe:
Bestimme den im Sinne von f ersten Nachfolgerknoten v_{sub} von v und aktualisiere falls notwendig Ω.
Rückgabe:
Nachfolgerknoten v_{sub}, falls dieser bestimmt werden konnte.

Methode First

Klasse:
clnVF
Objekt:
Kollektions-Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id, Viewport Π, Offset Ω
Aufgabe:
Bestimme das im Sinne von f erste Element (Sub-Objekt) o_{sub} von o und aktualisiere falls notwendig Ω.
Rückgabe:
Sub-Objekt o_{sub}, falls dieses bestimmt werden konnte

Methode Iterate

Klasse:
tplVF
Objekt:
Tupel-Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Sub-Objekt o_{sub}, Nachfolgerknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub}), Viewport Π, Offset Ω, Eigenschaften $\vec{e}_{sub}$

Aufgabe:
Bestimme, falls vorhanden, den Nachfolgerknoten v'_{sub} der im Sinne von f nächsten Komponente von v.

Rückgabe:
Nachfolgerknoten v'_{sub}, falls dieser bestimmt werden konnte

Methode Iterate

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Sub-Objekt o_{sub}, Nachfolgerknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub}), Viewport Π, Offset Ω, Eigenschaften $\vec{e}_{sub}$

Aufgabe:
Bestimme, falls vorhanden, das im Sinne von f nächste Element (Sub-Objekt) o'_{sub} von o.

Rückgabe:
Sub-Objekt o'_{sub}, falls dieses bestimmt werden konnte

Methode Null

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id, Viewport Π, Offset Ω

Aufgabe:
Generiere die Repräsentation einer nullwertigen Kollektion und aktualisiere Ω.

Rückgabe:
—

Methode Empty

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:

Datenobjekt o, Visualisierungsknoten v, Identifikation id, Viewport Π, Offset Ω

Aufgabe:

Generiere die Repräsentation einer leeren Kollektion und aktualisiere Ω.

Rückgabe:

—

Klasse	Methoden
atomVF	Pre, Post
constrVF	Pre, Post, Sub
tplVF	Pre, Post, Sub, First, Iterate
clnVF	Pre, Post, Sub, First, Iterate, Null, Empty

Tabelle 4.2: Visualisierungsfunktionen-Methoden für die Pre- und Post-Visualisierung

4.2.4 Explizite Generierung von Repräsentation

Bisher wurde davon ausgegangen, daß Repräsentationen von Datenobjekten „als ganzes“ generiert werden. Jetzt wird erläutert, wie wie auch Repräsentation von Sub-Objekten eines komplexen Datenobjektes generiert werden können.

Die Generierung von Repräsentationen erfolgt durch den wechselseitig rekursiven Aufruf der Visualisierungsknoten-Methoden Pre- und Post, wodurch Repräsentationen von Datenobjekten und Sub-Objekten generiert und zusammengesetzt werden. Die Methoden der Visualisierungsfunktionen werden dabei implizit gemäß der Definition von Visualisierung und Datenobjekt aufgerufen. Sie werden mit Parametern versorgt, die den Visualisierungsknoten, das Datenobjekt bzw. Sub-Objekt, den Viewport und den Offset sowie für komplexe Datenobjekte die bereits generierten Repräsentationen von Sub-Objekten spezifizieren.

Initiiert wird die Repräsentation eines Datenobjektes o gemäß einer Visualisierung V durch den Aufruf der Pre-Methode des Wurzelknotens v_{root} von V. Als Parameter werden o, ein durch das Fenstersystem bestimmter, maximaler Viewport sowie ein den Ursprung repräsentierender Offset übergeben.

Wie bereits angemerkt wurde, können auch spezielle Visualisierungsfunktionen Repräsentationen von Sub-Objekten integrieren, und in der Diskussion über die späte Visualisierung wurde von „eigenen Mechanismen zur bedarfsgesteuerten Generierung weiterer Repräsentationen“ gesprochen (siehe Abschnitt 4.1.1). Diese Repräsentationen müssen dann aber explizit generiert werden, d. h. die Pre-Methode des entsprechenden Visualisierungsknotens muß explizit aufgerufen und mit geeigneten Parametern versorgt werden. Dadurch, daß die Repräsentationen dann wieder durch die Mechanismen der Pre- und Post-Visualisierung generiert werden, halten auch sie die Kriterien der späten Visualisierung ein. Ein Beispiel

für die explizite Generierung von Repräsentation wird bei der Beschreibung der Visualisierungsfunktion clnForm in Abschnitt 5.8.2 gegeben, die zu einem Zeitpunkt nur ein Element einer Kollektion darstellt und weitere Repräsentationen von Elementen bedarfsgesteuert generiert.

Ganz bewußt soll durch die Möglichkeit der expliziten Generierung von Repräsentationen, die die Kriterien der späten Visualisierung einhalten, Modularität erreicht werden. Damit wird ein Baukastenprinzip realisiert, so daß die generierten Repräsentationen komponentenartig verwendet werden können. Z. B. könnten durch Visualisierungen definierte Repräsentationen als Ein-/Ausgabeschnittstelle zum Testen von bzw. zur Fehlersuche in Datenbankanwendungen mit komplexen Objekten verwendet werden. In Verbindung mit der prototypischen Implementierung mit Tcl/Tk und Tcl/DB sind folgenden Aussagen von Ousterhout interessant; sie belegen die Notwendigkeit eines modularen Visualisierungskonzeptes zur Integration von Anwendungen, die komplexe Objekte verwenden:

- „As the importance of integrating applications has increased, so has the use of scripting languages such as Tcl.“ [Ous99].
- „The most important factor is a shift in the application mix toward gluing applications. Three examples of this shift are GUIs, the Internet, and component frameworks.“ [Ous98, S. 28].

4.3 Interaktion

Die Repräsentation eines Datenobjektes soll nicht nur seine visuelle Darstellung, sondern auch die Interaktion mit dem Datenobjekt definieren. Zur Integration der Interaktion wird das im ESCHER-Projekt entwickelte Fingerkonzept (siehe Abschnitt 2.3.2) adaptiert.

Die in Abschnitt 3.2.1 angesprochene strukturelle Umwandlung von Datenobjekten durch *nest*- und *unnest*-Operationen [TF86] wird in dieser Arbeit nicht berücksichtigt, da sie zu sehr komplexen Änderungen der Repräsentation führen können. Auch die dynamische strukturelle Änderung von Repräsentationen, d. h. deren strukturelle Änderung zur Laufzeit, wird nicht betrachtet. Diese Aspekte sind eigenständige Themen für weitere Forschungsarbeiten (siehe Abschnitt 6.3).

Die Interaktion kann als Folge von einzelnen Interaktionsschritten angesehen werden. Jeder einzelne Interaktionsschritt ist die Ausführung einer **Benutzeroperation**[4]. Die Navigation in und die Manipulation von Datenobjekten wird an das Fingerkonzept gebunden, indem die Benutzeroperationen durch Fingeroperationen realisiert werden. Eine Benutzeroperation wird mit einem Eingabeereignis identifiziert, so daß die Interaktion letztendlich aus einer Folge von Eingabeereignissen besteht, die durch die Ausführung von Fingermethoden

[4] Der Begriff „Benutzeroperation“ wird hier im Sinn einer „kleinsten Einheit“ verwendet, die auch Teil von komplexeren Operationen sein können.

realisiert werden. Unter einem **Eingabeereignis** wird ein Maus- oder ein Tastaturereignis verstanden, also z. B. ein Maustasten-Klick in eine Repräsentation oder das Drücken einer Taste.

Die navigierende Interaktion mit einem Finger erlaubt das Selektieren eines Datenobjektes, bzw. eines seiner Sub-Objekte, und definiert dadurch einen Nimbus (vgl. [Rod96]). Der **Nimbus** eines Fingers ist die Menge, die durch den in Abschnitt 2.3.2 erwähnten Dereferenzierungsoperator $*$ definiert wird: Er umfaßt das Datenobjekt, auf das der Finger zeigt, sowie alle direkten und indirekten Sub-Objekte davon. Zeigt der Finger z. B. auf ein Tupel, umfaßt der Nimbus auch dessen Komponenten, usw.

Soll der Benutzer mittels Fingern interagieren, muß er die Finger sehen können. Die visuelle Repräsentation eines Fingers (Fingervisualisierung, Visualisierung eines Nimbus in der Repräsentation eines Datenobjektes) muß alle im Nimbus liegenden Datenobjekte umfassen. Sie wird in Abschnitt 4.3.3 beschrieben.

Finger werden hier als Objekte einer Klasse visFinger betrachtet; die Fingeroperationen werden durch die Methoden dieser Klasse definiert. Diese Interaktionsmethoden werden wiederum mit Methoden realisiert, die von den Visualisierungsfunktionen zur Verfügung gestellt werden müssen. Die Interaktionsmethoden der Fingerobjekte und der Visualisierungsfunktionen werden in den Abschnitten 4.3.1 bzw. 4.3.2 besprochen.

Die zwei Klassen von Benutzeroperationen, die zur Interaktion mit einem Datenobjekt notwendig sind, wurden bereits in Abschnitt 2.3.2 beschrieben: Die Navigations- und die Änderungsoperationen.

Das Navigieren in einem Datenobjekt ist gleichbedeutend mit dem Setzen und Bewegen eines Fingers, also mit dem Verändern eines Nimbus. Zeigt ein Finger auf ein komplexes Datenobjekt, umfaßt der Nimbus auch alle Sub-Objekte. Wird der Finger in das komplexe Datenobjekt hineinbewegt (Fingeroperation *push*), zeigt er auf genau eines seiner direkten Sub-Objekte, d. h. alle anderen Sub-Objekte müssen aus dem Nimbus genommen werden. Beim Bewegen eines Fingers von einem Sub-Objekt auf sein umgebendes Objekt (Fingeroperation *pop*) müssen alle anderen Sub-Objekte des umgebenden Objektes in den Nimbus aufgenommen werden. Wird der Finger auf ein anderes Sub-Objekt der gleichen Ebene bewegt (*move*-Fingeroperationen), muß der „alte“ Nimbus durch den „neuen“ ersetzt werden. Änderungsoperationen beeinflussen den Nimbus nur implizit durch die Positionierung des Fingers auf einem neuen bzw. anderen Datenobjekt.

Welche Operationen auf einen Finger angewendet werden können, ist von seinem Kontext abhängig. Der **Kontext** eines Fingers ist durch den Typ und die Eigenschaften des Datenobjektes, auf das er zeigt und ggf. auch durch den Typ und die Eigenschaften des umgebenden Objektes sowie die jeweiligen Visualisierungsknoten definiert. Wird im Folgenden vom Typ eines Fingers gesprochen, ist damit der Typ des Objektes gemeint, auf das ein Finger zeigt. Der Kontext eines Fingers definiert auch die Semantik der Fingeroperationen.

- Für *move*-Operationen ist der umgebende Typ maßgeblich: Zeigt ein Finger auf ein Element einer Kollektion, wird er auf ein anderes Element bewegt, zeigt er auf eine Komponente eines Tupels, wird er auf eine andere Komponente bewegt.

- Für die Operation *push* ist der Typ des Fingers maßgeblich: Zeigt ein Finger auf eine Kollektion oder ein Tupel, wird er auf das erste Element bzw. die erste Komponente bewegt. Es steht jedoch nicht a priori fest, welches Sub-Objekt das erste ist, die Visualisierungsfunktion muß dieses bestimmen. Zeigt ein Finger auf ein atomares Datenobjekt oder ein komplexes Datenobjekt, das durch eine spezielle, atomare Visualisierungsfunktion dargestellt wird, leitet die *push*-Operation das Editieren des Datenobjektes ein.

- Die Operation *pop* positioniert einen Finger auf dem umgebenden Objekt.

- Zeigt ein Finger auf eine nullwertige Kollektion, kann diese mit der Operation *to_empty* zu einer leeren Kollektion gemacht werden.

- Zeigt ein Finger auf eine leere Kollektion, kann diese mit der Operation *insert* (bzw. *to_null*) zu einer einelementigen (bzw. nullwertigen) Kollektion gemacht werden.

- Zeigt ein Finger auf ein Element einer Kollektion, fügt die Operationen *insert* ein neues Element ein, *delete* entfernt ein Element. Das Löschen des letzten Elementes einer Kollektion muß implizit wie eine *to_empty*-Operation behandelt werden.

Die Umsetzung der Benutzer- in Fingeroperationen, d. h. die Zuordnung zwischen Eingabeereignissen und dem Aufruf von Fingermethoden, wird durch die bereits erwähnten Bindungen der Visualisierungsfunktionen definiert. Dadurch kann z. B. eine Kollektions-Visualisierungsfunktion, die ihre Elemente vertikal konkateniert, die Benutzeroperation *down* in die Fingeroperation *next* umsetzen, indem sie an das Drücken der ↓-Taste, was ein entsprechendes Tastaturereignis auslöst, den Aufruf der Fingermethode Move mit dem Parameter $m =$ `next` bindet (siehe Seite 147).

Die für die einzelnen Visualisierungsfunktionen definierten Default-Bindungen können bei der Verfeinerung von Visualisierungen überschrieben und ergänzt werden, so daß auch die Interaktion konfigurierbar und erweiterbar ist. Die hier beschriebenen Fingermethoden bilden zusammen mit den in Kapitel 5 beschriebenen Default-Bindungen und Interaktionsmethoden der Visualisierungsfunktionen also nur die Basis des Interaktionskonzeptes.

Auf einem Datenobjekt sollen auch mehrere Finger gleichzeitig operieren können. Trotzdem ist eine Benutzeroperation immer nur einem Finger zugeordnet, der **aktiver Finger** genannt wird. Diese Beobachtung ist ganz analog zu graphischen Benutzerschnittstellen, die viele graphische Objekte gleichzeitig darstellen: Auch hier werden Tastaturereignisse nur einem Objekt (Dialogbaustein) zugeordnet; dieses Objekt hat den **Tastaturfokus**.

Eingabeereignisse können nicht direkt an Fingerobjekte gebunden werden, da die graphischen Objekte der Benutzerschnittstelle die generierten Repräsentationen sind. In Abschnitt 4.2.2 wurde bereits die Methode Bind der Klasse visNode erwähnt, die die Verbindung zwischen Repräsentationen (genauer: Identifikationen) und Benutzeroperationen hergestellt, z.B. indem die Tastaturereignisse an ein Behälter-Widget gebunden werden. Durch die Veränderung des Tastaturfokus ist es so einerseits möglich, die Veränderung des Kontexts bei Fingerbewegungen zu reflektieren; andererseits können dadurch die Benutzeroperationen dem aktiven Finger zugeordnet werden.

Die Methode Bind stellt zunächst fest, welche Bindungen für einen Visualisierungsknoten zu installieren sind. Das sind die Bindungen, die an dem Knoten eingetragen sind und die geerbten Bindungen von Vorgängerknoten. Aus den Menüangaben (Attribut Menu im Visualisierungsschema) wird ein Pop-Up-Menü generiert.

Methode Bind

Klasse:
visNode
Objekt:
Visualisierungsknoten v
Parameter:
Identifikation id
Berechnung:

1. Bestimme die Visualisierungsfunktion f von v.
2. Bestimme die Bindungen b von f.
3. Wenn v nicht die Wurzel der Visualisierung ist, dann:
   3.1. Bestimme den Vater-Visualisierungsknoten v_p von v und dessen Visualisierungsfunktion f_p.
   3.2. Bestimme die Bindungen b_p, die von f_p vererbt werden.
4. Generiere ein Pop-Up-Menü für b.
5. Installiere die Bindungen b und ggf. b_p für id.

Rückgabe:
—

4.3.1 Interaktionsmethoden von Fingern

Interaktionsmethoden von Fingern realisieren die Fingeroperationen und sorgen für die Aktualisierung der Repräsentation und der Fingervisualisierungen, wobei sie entsprechende Methoden von Visualisierungsfunktionen aufrufen.

Einige Fingermethoden werden aus dem Kontext eines Sub-Objektes aufgerufen. Für die Fingermethode Move heißt das z. B., daß der aufrufende Finger auf dem Sub-Objekt des komplexen Objektes steht, von dessen Visualisierungsfunktion die Move-Methode aufgerufen werden soll. Alle diese Methoden überprüfen zunächst, ob ein entsprechendes, den Finger umgebendes Objekt existiert. Die dadurch evtl. resultierenden Fehler führen zum Abbruch einer Benutzeroperation und müssen dem Benutzer angemessen, d. h. visuell und/oder akustisch [FS95] signalisiert werden.

Fingeroperationen sind immer datenbezogen; da Konstruktions-Visualisierungsfunktionen nicht datenbezogen sind, werden für sie keine Bindungen definiert und keine Interaktionsmethoden implementiert und sie werden auch bei der Veränderung des Tastaturfokus nicht berücksichtigt. Wird eine Fingermethode aus dem Kontext eines Sub-Objektes aufgerufen, dessen Visualisierungsknoten einen Konstruktions-Visualisierungsknoten als Vorgängerknoten hat, müssen der Interaktionsmethode der Visualisierungsfunktion, die zum umgebenden komplexen Datenobjekt gehört, nicht die Informationen der Repräsentation des Sub-Objektes, sondern die dem Konstruktions-Visualisierungsknoten entsprechenden übergeben werden.

Um die Fingeroperationen einem Datenobjekt zuordnen zu können, muß von einem Fingerobjekt sein Kontext abgeleitet werden können. Zur Visualisierung des Fingers F und der durch Operationen bewirkten Änderungen werden die folgenden Informationen benötigt:

- Das Datenobjekt o, auf das F zeigt,
- das o umgebende Datenobjekt[5] o_p,
- die zu o und o_p gehörenden Visualisierungsknoten v und v_p,
- der Nachfolgerknoten[6] v_{sub} von v_p,
- die Visualisierungsfunktionen f von v, f_p von v_p und f_{sub} von v_{sub} und
- die Identifikationen id von (o, v), id_p von (o_p, v_p) und id_{sub} von (o, v_{sub}).

Einige Fingermethoden subsumieren die Funktionalität von mehreren Fingeroperationen, so daß nicht für jede Fingeroperation eine dedizierte Fingermethode existiert. Andere Fingermethoden werden indirekt aufgerufen und sind daher keiner Fingeroperationen direkt zugeordnet. In Tabelle 4.3 auf der nächsten Seite werden die Zusammenhänge zwischen implementierten Fingermethoden, realisierten Fingeroperationen und benutzten Methoden von Visualisierungsfunktionen dargestellt.

Im Folgenden werden die einzelnen Interaktionsmethoden von Fingern beschrieben. Zur Erhaltung der Konsistenz von Repräsentationen werden aus den Methoden nach dem Bewegen eines Fingers oder Verändern eines Datenobjektes die Visualisierungsknoten-Metho-

[5] Falls F kein **Wurzelfinger** ist, d. h. auf die Wurzel eines Objektbaumes zeigt.

[6] Der Nachfolgerknoten v_{sub} eines umgebendes Datenobjekt o_p ist identisch mit dem Visualisierungsknoten v des Datenobjektes o, wenn zwischen v und v_p keine Konstruktions-Visualisierungsknoten liegen. Hat v jedoch einen solchen als Vorgängerknoten, dann ist v_{sub} die erste Konstruktions-Visualisierungsknoten auf dem Pfad von v_p nach v.

Fingeroperation	Finger-methode	indirekt aufgerufene Fingermethode	Visualisierungsfunk-tionen-Methode
push	Push		Push, Edit
to_empty		Null_Empty[a]	Null_To_Empty
to_singleton		Empty_To_Singleton[b]	
pop	Pop		Pop,
move	Move		Move_...
(*first*, *next*, ...)			Move_First, ...
insert,	Insert		Insert
insert_before			
delete,	Delete		Delete
delete_before			
		Singleton_To_Empty[c]	Singleton_To_Empty
to_null	Null_Empty		Empty_To_Null

Tabelle 4.3: Zusammenhang zwischen Interaktionsoperationen und -methoden

[a] Auf nullwertigen Kollektionen.
[b] Auf leeren Kollektionen.
[c] Auf einelementigen Kollektionen.

den Adjust (zur Anpassung des Viewports) und Resize (zur Anpassung von veränderten Abmessungen) aufgerufen. Diese werden in den Abschnitten 4.3.4 und 4.3.5 besprochen.

Die Veränderung der Fingerposition wird nach einer Fingerbewegung durch den Aufruf der Fingermethode Show in der Repräsentation reflektiert. Je nach Beeinflussung des Nimbus wird sie für die alte und/oder neue Fingerposition aufgerufen. Die alte Fingerposition ist die Position des Fingers vor der Bewegung und wird durch das Anlegen einer Kopie des Fingers gespeichert. Bei der Formulierung der Fingerinteraktionsmethoden wird auf die sorgfältige Speicherverwaltung verzichtet — die Fingerkopien werden nicht explizit wieder gelöscht.

Bei der Beschreibung der Methoden wird häufig die Formulierung „Aktualisiere den Finger F, so daß er auf ... zeigt“ benutzt. Damit ist einerseits gemeint, daß die Datenstruktur, die den Finger repräsentiert, so aktualisiert werden soll, daß sie die neue Fingerposition reflektiert. Andererseits muß der Tastaturfokus entsprechend der neuen Fingerposition geändert werden. Zur Änderung des Tastaturfokus kann in einer Tcl/Tk-Implementierung das Kommando `focus` mit dem Widget-Pfad, der der Identifikation einer Repräsentation entspricht, verwendet werden.

Für die Generierung von neuen Repräsentationen durch die Pre-Methode eines Visualisierungsknotens muß in einigen Methoden ein Viewport und ein Offset bestimmt werden. Dieses kann durch Aufruf der Position-Methode des umgebenden Visualisierungsknotens erfolgen.

Push

Die Fingermethode Push dient dem Navigieren in ein komplexes Objekt hinein und dem Editieren von atomaren Objekten. Letzteres wird durch den Aufruf der Edit-Methode einer atomaren Visualisierungsfunktion realisiert. Für Kollektionen, die nullwertig oder leer sind, wird von Push eine Umwandlung in eine leere bzw. einelementige Kollektion ausgeführt. Dazu werden die Fingermethoden Null_Empty bzw. Empty_To_Singleton aufgerufen. Für atomare Objekte und bei der Umwandlung einer nullwertigen Kollektion in eine leere Kollektion wird der Finger nicht bewegt, so daß Push vorzeitig beendet werden kann. Die Fingerbewegung bei der Umwandlung einer leeren Menge in eine einelementige Menge wird von Empty_To_Singleton ausgeführt, so daß auch in diesem Fall Push vorzeitig beendet werden kann.

Größenänderungen können beim Editieren atomarer Objekte und bei der Umwandlung von nullwertigen und leeren Kollektionen auftreten. Die letzten beiden Fälle werden in den von Push aufgerufenen Fingermethoden Null_Empty und Empty_To_Singleton bearbeitet. Push bleibt also die Behandlung von Größenänderungen nach dem Editieren atomarer Objekte. Die Methode Edit der atomaren Visualisierungsfunktionen gibt dazu eine durch die Wertänderung veränderte Visualisierungsgröße als Differenz zwischen alter und neuer Visualisierungsgröße (**Resize-Delta**) zurück.

Für atomare Objekte wird vor dem Aufruf der Edit-Methode die Adjust-Methode des atomaren Visualisierungsknotens aufgerufen, um sicherzustellen, daß deren Repräsentation im sichtbaren Ausschnitt liegt. Außerdem werden vor dem Aufruf der Edit-Methode die aktuellen Einstellungen des Fenstersystems bzgl. des Tastatur- und Mausfokus (Empfänger von Eingabeereignissen) gespeichert, und nach dem Aufruf der Edit-Methode wiederhergestellt. Das hat erstens den Sinn, die Implementierung der Edit-Methode von dieser Aufgabe zu entlasten. Zweitens bietet das die Sicherheit, daß diese Einstellungen nicht versehentlich in einem von der Edit-Methode veränderten Zustand bleiben.

Wird der Finger auf ein Sub-Objekt bewegt, bestimmt die Push-Methode der Visualisierungsfunktion, welches Sub-Objekt das erste ist. Da der alte Nimbus den neuen umfaßt, muß nach der Viewport-Anpassung nur die Repräsentation an der alten Fingerposition aktualisiert werden.

Methode Push

Klasse:
visFinger

Objekt:
Finger F

Parameter:
—

Berechnung:

1. Bestimme zu F den Visualisierungsknoten v, das Datenobjekt o, die Identifikation id von (o, v) und die Visualisierungsfunktion f von v.
2. Wenn f eine atomare Visualisierungsfunktion ist, dann:
   2.1. Führe die Adjust-Methode von v mit dem Parameter o aus.
   2.2. Speichere die Einstellungen von Tastatur- und Mausfokus.
   2.3. Führe die Edit-Methode von f mit den Parametern o, v und id aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
   2.4. Restauriere die Einstellungen von Tastatur- und Mausfokus.
   2.5. Wenn F kein Wurzelfinger ist, dann:
      2.5.1. Bestimme zu F den Vater-Visualisierungsknoten v_p und das umgebende Datenobjekt o_p.
      2.5.2. Führe die Resize-Methode von v_p mit den Parametern o_p und $\vec{\Delta}_r$ aus.
   2.6. Ende.
3. Wenn f eine Kollektions-Visualisierungsfunktion ist, dann:
   3.1. Wenn o null ist, dann:
      3.1.1. Führe die Null_Empty-Methode von F mit dem Parameter $m =$ `to_empty` aus.
      3.1.2. Ende.
   3.2. Wenn o leer ist, dann:
      3.2.1. Führe die Empty_To_Singleton-Methode von F aus.
      3.2.2. Ende.
4. Lege eine Kopie F' von F an.
5. Wenn f eine Tupel-Visualisierungsfunktion ist, dann:
   5.1. Führe die Push-Methode von f mit den Parametern o, v und id aus und bestimme den Nachfolgerknoten v_{sub} der im Sinne von f ersten Komponente von v.
   5.2. Bestimme das v_{sub} entsprechende Sub-Objekt o_{sub} von o.
6. Wenn f eine Kollektions-Visualisierungsfunktion ist, dann:
   6.1. Führe die Push-Methode von f mit den Parametern o, v und id aus und bestimme das im Sinne von f erste Element (Sub-Objekt) o_{sub} von o.
   6.2. Bestimme den Nachfolgerknoten v_{sub} von v.
7. Führe die Adjust-Methode von v_{sub} mit dem Parameter o_{sub} aus.
8. Führe die Show-Methode von F' aus.
9. Aktualisiere den Finger F, so daß er auf o_{sub} zeigt.

Rückgabe:

—

Pop

Mit der Methode Pop wird der Nimbus ausgeweitet. Dazu muß nur die Repräsentation an der neuen Fingerposition aktualisiert werden, da der neue Nimbus den alten umfaßt.

Methode Pop

Klasse:
visFinger
Objekt:
Finger F
Parameter:
—
Berechnung:
1. Wenn F Wurzelfinger ist, dann signalisiere einen Fehler.
2. Bestimme zu F den Visualisierungsknoten v, das Datenobjekt o, die Identifikation id von (o, v) und die Visualisierungsfunktion f von v.
3. Führe die Pop-Methode von f mit den Parametern o, v und id aus.
4. Bestimme den Vater-Visualisierungsknoten v_p von v und das o umgebende Datenobjekt o_p.
5. Führe die Adjust-Methode von v_p mit dem Parameter o_p aus.
6. Führe die Show-Methode von F aus.
7. Aktualisiere den Finger F, so daß er auf o_p zeigt.

Rückgabe:
—

Move

Die Methode Move implementiert alle Fingerbewegungen innerhalb der selben Hierarchieebene. Diese haben gemeinsam, daß nach einer erfolgreichen Bewegung alte und neue Fingerposition disjunkt sind[7]. Die Schnittstelle der Methode umfaßt einen Parameter (Modus m), der die Bewegungsart spezifiziert (z. B. *next*). Visualisierungsfunktionen, die die Fingermethode Move benutzen, müssen für jeden Modus eine eigene Methode implementieren. Z. B. benutzt die Visualisierungsfunktion cln (siehe Abschnitt 5.8.1) für den Modus die Werte „next“, „back“, „first“ und „last“, um mittels ihrer Methoden Move_Next, Move_Back, Move_First und Move_Last die *next*-, *back*-, *first*- und *last*-Fingeroperationen zu realisieren.

[7] Eine Bewegung ist nicht erfolgreich, falls ein Finger nicht wie gefordert bewegt werden kann, z. B. weil er vor einer *next*-Operation bereits am Ende einer Kollektion steht, oder falls ein Finger durch eine Operation nicht bewegt wird, z. B. weil er vor einer *last*-Operation bereits am Ende einer Kollektion steht.

Methode Move

Klasse:
visFinger
Objekt:
Finger F
Parameter:
Modus m
Berechnung:

1. Wenn F Wurzelfinger ist, dann signalisiere einen Fehler.
2. Bestimme zu F den Visualisierungsknoten v, den Vater-Visualisierungsknoten v_p und dessen Nachfolgerknoten v_{sub}.
3. Bestimme zu F das Datenobjekt o und das umgebende Datenobjekt o_p.
4. Bestimme die Identifikationen id_p von (o_p, v_p) und id_{sub} von (o, v_{sub}).
5. Lege eine Kopie F' von F an.
6. Bestimme die Visualisierungsfunktion f_p von v_p.
7. Wenn f_p weder eine Tupel- noch eine Kollektions-Visualisierungsfunktion ist ($f_p \notin \mathsf{tplVF} \cup \mathsf{clnVF}$), dann signalisiere einen Fehler.
8. Wenn f_p eine Tupel-Visualisierungsfunktion ist, dann führe die Move_m-Methode von f_p mit den Parametern o_p, v_p, id_p, o, v_{sub} und id_{sub} aus und bestimme falls möglich einen neuen Visualisierungsknoten v'.
9. Wenn f_p eine Kollektions-Visualisierungsfunktion ist, dann führe die Move_m-Methode von f_p mit den Parametern o_p, v_p id_p, o, v_{sub} und id_{sub} aus und bestimme falls möglich ein neues Datenobjekt o'.
10. Wenn ein neuer Visualisierungsknoten oder ein neues Datenobjekt bestimmt wurde, dann:
    - 10.1. Wenn f_p eine Tupel-Visualisierungsfunktion ist, dann bestimme das neue Datenobjekt o' zu v'.
    - 10.2. Wenn f_p eine Kollektions-Visualisierungsfunktion ist, dann bestimme den neuen Visualisierungsknoten v' zu o'.
    - 10.3. Aktualisiere den Finger F, so daß er auf o' zeigt.
    - 10.4. Führe die Show-Methode von F' aus.
    - 10.5. Führe die Adjust-Methode von v' mit dem Parameter o' aus.
    - 10.6. Führe die Show-Methode von F aus.
11. Sonst:
    - 11.1. Führe die Adjust-Methode von v mit dem Parameter o aus.
    - 11.2. Signalisiere einen Fehler.

Rückgabe:
—

Insert

Mit der Methode Insert können Elemente in Kollektionen eingefügt werden. Der Parameter Modus m steuert, ob das neue Element vor (m = `before`) oder nach (m = `after`) dem Finger eingefügt werden soll. Für die Realisierung spezieller Visualisierungsfunktionen, die komplex strukturierte Datenobjekte als atomare Repräsentationen darstellen, kann ein einzufügendes Datenobjekt übergeben werden. Ansonsten wird ein nullwertiges Element[8] generiert und eingefügt. Mit dem Einfügen ist implizit eine *move*-Operation verbunden, d. h. der Finger wird nach dem Einfügen auf dem neu eingefügten Element positioniert. Diese Eigenschaft wird beim Aufruf der Pre-Methode des Visualisierungsknotens ausgenutzt, da dort die Werte für Viewport und Offset der aktuellen Fingerposition als Parameter übergeben werden.

Methode Insert

Klasse:
visFinger

Objekt:
Finger F

Parameter:
Modus m, neues Datenobjekt o' (optional)

Berechnung:

1. Wenn F Wurzelfinger ist, dann signalisiere einen Fehler.
2. Bestimme zu F den Visualisierungsknoten v, den Vater-Visualisierungsknoten v_p und dessen Nachfolgerknoten v_{sub}.
3. Bestimme zu F das Datenobjekt o und das umgebende Datenobjekt o_p.
4. Bestimme die Identifikationen id_p von (o_p, v_p) und id_{sub} von (o, v_{sub}).
5. Lege eine Kopie F' von F an.
6. Bestimme die Visualisierungsfunktion f_p von v_p.
7. Wenn f_p keine Kollektions-Visualisierungsfunktion ist ($f_p \notin$ clnVF), dann signalisiere einen Fehler.
8. Bestimme den Viewport Π von (o_p, v_p) und den Offset Ω von (o, v_{sub}) in (o_p, v_p).
9. Wenn o' nicht als Argument übergeben wurde, dann generiere ein neues Sub-Objekt o' von o_p.
10. Führe die Pre-Methode von v_{sub} mit den Parametern o', Π und Ω aus und bestimme die Identifikation id' und die Eigenschaften $\vec{e}'$ der Repräsentation von o'.

[8] Siehe die Bemerkungen zu Tupeln und Nullwerten in den Abschnitten 2.3.1 und 2.3.2.

11. Führe die Insert-Methode von f_p mit den Parametern o_p, v_p, id_p, o, v_{sub}, id_{sub}, m, o', id', $\vec{e}'$, Π und Ω aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
12. Aktualisiere den Finger F, so daß er auf o' zeigt.
13. Führe die Resize-Methode von v_p mit den Parametern o_p und $\vec{\Delta}_r$ aus.
14. Führe die Show-Methode von F' aus.
15. Führe die Adjust-Methode von v_{sub} mit dem Parameter o' aus.
16. Führe die Show-Methode von F aus.

Rückgabe:

—

Delete

Mit der Methode Delete können Elemente aus Kollektionen gelöscht werden. Da auch das letzte (i. S. v. einzige) Element einer Kollektion gelöscht werden kann, implementiert Delete auch die Funktionalität von Singleton_To_Empty. Der Parameter Modus steuert, ob das aktuelle oder das vorige Element gelöscht werden soll. Ist $m =$ `back`, wird das vorige Element von der Move_Back-Methode der Kollektions-Visualisierungsfunktion bestimmt und der Finger wird nicht bewegt. Ist $m =$ `current`, wird vor dem Löschen das Objekt bestimmt, auf das der Finger nach dem Löschen zeigt. Dazu werden dann die Move_Back-Methode, falls das letzte Element gelöscht wird, oder sonst die Move_Next-Methode benutzt.

Methode Delete

Klasse:

visFinger

Objekt:

Finger F

Parameter:

Modus m

Berechnung:

1. Wenn F Wurzelfinger ist, dann signalisiere einen Fehler.
2. Bestimme zu F den Visualisierungsknoten v, den Vater-Visualisierungsknoten v_p und dessen Nachfolgerknoten v_{sub}.
3. Bestimme zu F das Datenobjekt o und das umgebende Datenobjekt o_p.
4. Bestimme die Identifikationen id_p von (o_p, v_p), id von (o, v) und id_{sub} von (o, v_{sub}).
5. Bestimme die Visualisierungsfunktion f_p von v_p.

6. Wenn f_p keine Kollektions-Visualisierungsfunktion ist ($f_p \notin$ clnVF), dann signalisiere einen Fehler.
7. Wenn o das einzige Element von o_p ist, dann:
   7.1. Führe die Pop-Methode von f mit den Parametern o, v und id aus.
   7.2. Führe die Singleton_To_Empty-Methode von f_p mit den Parametern o_p, v_p, id_p, o, v_{sub} und id_{sub} aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
   7.3. Aktualisiere den Finger F, so daß er auf o_p zeigt.
   7.4. Setze $o' := o_p$ und $v' := v_p$.
8. Sonst:
   8.1. Bestimme das m entsprechende zu löschende Datenobjekt o_{del} und die Visualisierungsfunktion id_{del} von (o_{del}, v_{sub}).
   8.2. Bestimme das m entsprechende Datenobjekt o', auf das F nach dem Löschen zeigen soll.
   8.3. Führe die Delete-Methode von f_p mit den Parametern o_p, v_p, id_p, o_{del}, v_{sub} und id_{del} aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
   8.4. Wenn $o' \neq o$ gilt, dann aktualisiere den Finger F, so daß er auf o' zeigt.
9. Führe die Resize-Methode von v_p mit den Parametern o_p und $\vec{\Delta}_r$ aus.
10. Führe die Adjust-Methode von v' mit dem Parameter o' aus.
11. Wenn $o' \neq o$ gilt, dann aktualisiere die Repräsentation an der neuen Fingerposition.

Rückgabe:

—

Null_Empty

Mit der Methode Null_Empty können leere Kollektionen in nullwertige Kollektionen gewandelt werden und umgekehrt. Sie implementiert also die Fingerpoerationen *to_null* und *to_empty*. Der Parameter Modus steuert, in welcher Richtung umgewandelt werden soll. Die Fingerposition wird nicht verändert; trotzdem muß die Repräsentation an der gegenwärtigen Fingerposition aktualisiert werden, um die korrekte Fingervisualisierung im gerade umgewandelten Datenobjekt sicherzustellen. Diese Methode wird mit $m =$ `to_empty` auch von der Methode Push aufgerufen, falls die Fingeroperation in auf eine nullwertige Menge angewendet wird.

Methode Null_Empty

Klasse:

visFinger

Objekt:
Finger F
Parameter:
Modus m
Berechnung:

1. Bestimme zu F den Visualisierungsknoten v, das Datenobjekt o, die Identifikation id von (o, v) und die Visualisierungsfunktion f von v.
2. Wenn f keine Kollektions-Visualisierungsfunktion ist ($f \notin$ clnVF), dann signalisiere einen Fehler.
3. Wenn $m =$ `to_null` ist, dann:
   3.1. Mache o zu einer nullwertigen Kollektion.
   3.2. Führe die Empty_To_Null-Methode von f mit den Parametern o, v und id aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
4. Wenn $m =$ `to_empty` ist, dann:
   4.1. Mache o zu einer leeren Kollektion.
   4.2. Führe die Null_To_Empty-Methode von f mit den Parametern o, v und id aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
5. Führe die Resize-Methode von v mit den Parametern o und $\vec{\Delta}_r$ aus.
6. Führe die Adjust-Methode von v mit dem Parameter o aus.
7. Führe die Show-Methode von F aus.

Rückgabe:
—

Empty_To_Singleton

Die Methode Empty_To_Singleton implementiert das Umwandeln einer leeren in eine einelementige Kollektion. Genau wie bei der Methode Insert kann für die Realisierung spezieller Visualisierungsfunktionen ein einzufügendes Datenobjekt übergeben werden. Ansonsten wird ein nullwertiges Element generiert[9]. Diese Methode wird auch von der Methode Push aufgerufen, falls die Fingeroperation in auf eine leere Menge angewendet wird. Mit dem Finger wird implizit eine *push*-Operation ausgeführt, d. h. er wird auf dem neuen, einzigen Element positioniert.

Methode Empty_To_Singleton

Klasse:
visFinger

[9] Es gilt die Aussage von Fußnote 8 auf Seite 149.

Objekt:
Finger F

Parameter:
neues Datenobjekt o_{sub} (optional)

Berechnung:

1. Lege eine Kopie F' von F an.
2. Bestimme zu F den Visualisierungsknoten v, das Datenobjekt o, die Identifikation id von (o, v) und die Visualisierungsfunktion f von v.
3. Bestimme den Nachfolgerknoten v_{sub} von v.
4. Bestimme Viewport Π und Offset Ω von (o, v).
5. Wenn o_{sub} als Argument übergeben wurde, dann
   5.1. mache es zum einzigen Element von o,
   5.2. sonst generiere o_{sub} als einziges, nullwertiges Element von o.
6. Führe die Pre-Methode von v_{sub} mit den Parametern o_{sub}, Π und Ω aus und bestimme die Identifikation id_{sub} und die Eigenschaften $\vec{e}_{sub}$ der Repräsentation von o_{sub}.
7. Führe die Empty_To_Singleton-Methode von f mit den Parametern o, v, id, o_{sub}, v_{sub}, id_{sub}, $\vec{e}_{sub}$, Π und Ω aus und bestimme das Resize-Delta $\vec{\Delta}_r$.
8. Führe die Resize-Methode von v mit den Parametern o und $\vec{\Delta}_r$ aus.
9. Führe die Adjust-Methode von v_{sub} mit dem Parameter o_{sub} aus.
10. Aktualisiere den Finger F, so daß er auf o_{sub} zeigt.
11. Führe die Show-Methode von F' aus.

Rückgabe:
—

4.3.2 Interaktionsmethoden von Visualisierungsfunktionen

Nachdem die Interaktionsmethoden der Finger beschrieben wurden, soll im Folgenden analog zu Abschnitt 4.2.3 die Erweiterung der Schnittstelle der Visualisierungsklassen beschrieben werden. In Tabelle 4.4 auf Seite 158 ist eine Übersicht der Interaktionsmethoden für die einzelnen Visualisierungsklassen dargestellt.

Methode Push

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id

Aufgabe:
Bestimme das im Sinne von f erste Element (Sub-Objekt) o_{sub} von o.

Rückgabe:
Datenobjekt o_{sub}

Methode Push

Klasse:
tplVF

Objekt:
Tupel-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id

Aufgabe:
Bestimme den Nachfolgerknoten v_{sub} der im Sinne von f ersten Komponenten von v.

Rückgabe:
Visualisierungsknoten v_{sub}

Methode Edit

Klasse:
atomVF

Objekt:
atomare Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id

Aufgabe:
Erlaube dem Benutzer die Änderung des atomaren Datenobjektes o. Reflektiere eine Wertänderung von o durch id und berechne das Resize-Delta für id.

Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Methode Pop

Klasse:
visFunc

Objekt:
Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id

Aufgabe:
— (Wird trotzdem aufgerufen und kann benutzt werden, um interne Aufgaben der Visualisierungsfunktion zu bearbeiten.)

Rückgabe:
—

Methode Move_...[10]

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Element (Sub-Objekt) o_{sub}, Element-Visualisierungsknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub})

Aufgabe:
Bestimme falls möglich ein neues Element (Sub-Objekt) o'_{sub} von o relativ zu o_{sub}.

Rückgabe:
Sub-Objekt o'_{sub}, falls dieses bestimmt werden konnte

Methode Move_...[11]

Klasse:
tplVF

Objekt:
Tupel-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Komponente (Sub-Objekt) o_{sub}, Komponenten-Visualisierungsknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub})

Aufgabe:
Bestimme falls möglich einen neuen Komponenten-Visualisierungsknoten v'_{sub} von v relativ zu v_{sub}.

Rückgabe:
Komponenten-Visualisierungsknoten v'_{sub} falls dieser bestimmt werden konnte

[10] Durch die Punkte soll angedeutet werden, daß tatsächlich mehrere Move-Methoden definiert werden: Für jeden Modus m, mit dem die Fingermethode Move in durch die Visualisierungsfunktion definierten Bindungen aufgerufen wird, muß eine Move-Methoden definiert sein.

[11] Es gilt die Aussage von Fußnote 10 auf dieser Seite.

Methode Insert

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Element (Sub-Objekt) o_{sub}, Element-Visualisierungsknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub}), Modus m, neues Element (Sub-Objekt) o'_{sub}, Identifikation id'_{sub} von (o'_{sub}, v_{sub}), Eigenschaften $\vec{e}'_{sub}$, Viewport Π und Offset Ω

Aufgabe:
Integriere id'_{sub} in id, und zwar gemäß m relativ zu id_{sub}. Falls notwendig, beachte dabei den Viewport Π von id, den Offset Ω von id_{sub} in id und die Eigenschaften $\vec{e}'_{sub}$ von id'_{sub} und berechne das Resize-Delta für id.

Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Methode Delete

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Element (Sub-Objekt) o_{sub}, Element-Visualisierungsknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub})

Aufgabe:
Entferne id_{sub} aus id und berechne das Resize-Delta für id.

Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Methode Null_To_Empty

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id

Aufgabe:
Ersetze in id die Repräsentation der nullwertigen Kollektion durch die Repräsentation der leeren Kollektion und berechne das Resize-Delta für id.
Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Methode Empty_To_Singleton

Klasse:
clnVF
Objekt:
Kollektions-Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Element (Sub-Objekt) o_{sub}, Element-Visualisierungsknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub}), Eigenschaften $\vec{e}'_{sub}$, Viewport Π, Offset Ω
Aufgabe:
Ersetze in id die Repräsentation der leeren Kollektion durch id_{sub}. Falls notwendig, beachte dabei den Viewport Π von id, den Offset Ω und die Eigenschaften $\vec{e}_{sub}$ von id_{sub} und berechne das Resize-Delta für id.
Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Methode Singleton_To_Empty

Klasse:
clnVF
Objekt:
Kollektions-Visualisierungsfunktion f
Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Element (Sub-Objekt) o_{sub}, Element-Visualisierungsknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub})
Aufgabe:
Ersetze in id id_{sub} durch die Repräsentation der leeren Kollektion und berechne das Resize-Delta für id.
Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Methode Empty_To_Null

Klasse:
clnVF

Objekt:
Kollektions-Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id

Aufgabe:
Ersetze in id die Repräsentation der leeren Kollektion durch die Repräsentation der nullwertigen Kollektion und berechne das Resize-Delta für id.

Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Klasse	Methoden
atomVF:	Edit, Pop
constrVF:	—
tplVF:	Push, Pop, Move_...[a]
clnVF:	Push, Pop, Move_...[a], Insert, Delete, Null_To_Empty, Empty_To_Singleton, Singleton_To_Empty, Empty_To_Null

Tabelle 4.4: Interaktionsmethoden von Visualisierungsfunktionen

[a]Es gilt die Aussage von Fußnote 10 auf Seite 155.

4.3.3 Fingervisualisierung

Eine Fingervisualisierung kann, wie auch die Repräsentation des Datenobjektes selber, verschiedene Formen annehmen. Dazu können verschiedene Attribute der Fingervisualisierung variiert werden, z. B. Vorder- bzw. Hintergrundfarbe, Schriftart, Reliefart, usw. Durch Vertauschen der Vorder- mit der Hintergrundfarbe kann z. B. eine invertierte Darstellung des Fingers, wie sie in Text-Editoren zur Darstellung des Cursors üblich ist, erreicht werden.

Da mehrere Finger auf oder in ein Datenobjekt zeigen können[12], müssen auch mehrere Finger dargestellt und in der Repräsentation unterschieden werden können. Dazu ist jedem Finger eine Fingerart zugeordnet, die für alle in einer Repräsentation enthaltenen Finger unterschiedlich ist. Jeder Repräsentation ist eine Liste der in ihr visualisierten Finger zugeordnet[13]. Diese Liste definiert eine totale Ordnung $\lessdot$ auf den Fingern, die kompatibel mit der durch $\Subset$ (siehe Abschnitt 3.1.3) definierten partiellen Ordnung ist. Die Ordnung ist aufgrund der Definition mittels einer Liste vom Visualisierungsverfahren beeinflußbar. Ist ein Datenobjekt im Nimbus mehrerer Finger, wird in seiner Repräsentation der erste Finger in der Liste visualisiert.

[12]Die Formulierung „in ein Datenobjekt zeigen“ soll andeuten, daß ein Finger auf ein direktes oder indirektes Sub-Objekt eines komplexen Datenobjektes zeigt.

[13]Nicht jeder Finger muß visualisiert sein; z. B. werden Finger, die in Tcl/DB-Skripten verwendet werden, i. d. R. nicht visualisiert.

Jede Visualisierungsfunktion definiert durch die Implementierung einer Finger-Methode selber, wie „ihre“ Fingervisualisierung aussieht. Die Methode hat als Parameter u. a. die Fingerart, die die Darstellung verschiedener bzw. sich überlappender Finger steuert. Außerdem soll die Finger-Methode auch den Zustand darstellen, daß ein Datenobjekt in keinem Nimbus enthalten ist. Diese Funktionalität wird ebenfalls über den Parameter Fingerart gesteuert. Die Fingerart kann z.B. ein Index in eine Farbtabelle sein. Ist er Null, wird das Objekt mit der Hintergrundfarbe dargestellt (also ohne Finger), ansonsten in der dem Index entsprechenden Farbe.

Die in Abschnitt 4.3 beschrieben Fingerobjekte haben eine Verbindung zu der obigen Fingerliste. Muß die Repräsentation für eine Fingerposition[14] aktualisiert werden, kann für jedes Datenobjekt, das im durch die Fingerposition definierten Nimbus liegt, festgestellt werden, mit welcher Fingerart die Finger-Methode der Visualisierungsfunktion aufgerufen werden muß: Liegt das Datenobjekt in keinem Nimbus eines Fingers der Liste, ist die Fingerart Null, ansonsten bestimmt der erste Finger der Liste, dessen Nimbus das Datenobjekt umfaßt, die Fingerart. Der Test, ob ein Datenobjekt im Nimbus eines Fingers liegt, ist durch die auf einem Stapel von Objektidentifikationen basierende Realisierung von Fingern sehr effizient ausführbar.

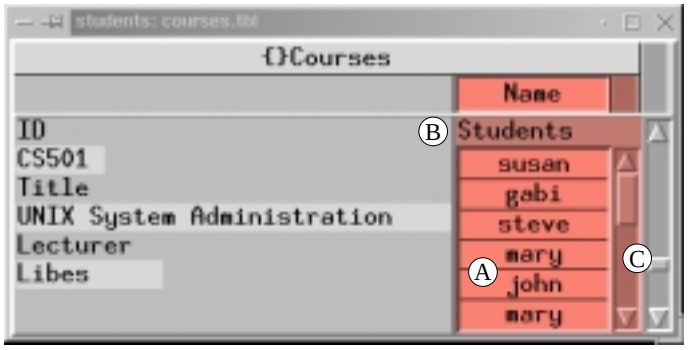
Zeigt ein Finger auf ein Sub-Objekt eines komplexen Datenobjektes, also entweder eine Komponente eines Tupels oder ein Element einer Kollektion, kann zur Darstellung des Fingers auch die Visualisierungsfunktion des umgebenden Objektes, also die Tupel- bzw. die Kollektions-Visualisierungsfunktion, beitragen. Für diese Funktionalität wird die Schnittstelle der Klassen cplxDataVF um die Methode Sub_Finger erweitert. Das Beispiel in Abb. 4.4 auf der nächsten Seite erläutert den Sachverhalt.

Zur Fingervisualisierung wird ein Verfahren benötigt, das die Visualisierung, beginnend bei der Fingerposition, bis zu den atomaren Komponenten, durchläuft, und die Finger- und Sub_Finger-Methoden der Visualisierungsfunktionen aufruft. Auch dieses Verfahren soll, wie die Pre- und Post-Visualisierung, durch Viewport und Offset gesteuert sein, damit keine unnötigen Fingervisualisierungen in nicht sichtbaren Sub-Objekten ausgeführt werden. Zunächst wird die Fingermethode Show beschrieben, die als Einstiegspunkt für die folgende Visualisierungsknoten-Methode Finger dient. Letztere durchläuft die Visualisierung ähnlich wie die Visualisierungsknoten-Methoden Pre und Post und ruft dabei die Visualisierungsfunktionen-Methoden Finger mit der jeweiligen Fingerart auf.

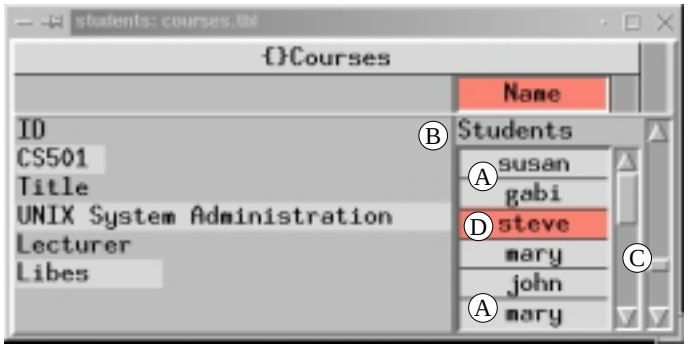
Methode Show

Klasse:
visFinger
Objekt:
Finger F

[14] Eine Fingerpositionen als Kopie eines Fingers vor seiner Bewegung (siehe Abschnitt 4.3.1) ist nicht in der Fingerliste enthalten.



(a) mit Fingervisualisierung im umgebenen Objekt



(b) ohne Fingervisualisierung im umgebenen Objekt

Abbildung 4.4: Fingervisualisierung von Sub-Objekten

Beispiel 4.4 Fingervisualisierung von Sub-Objekten

Die Tupel-Visualisierungsfunktion tplLabel (siehe Abschnitt 5.7.2) dekoriert Komponenten zusätzlich mit ihren Attributnamen. Zeigt ein Finger auf eine der Komponenten, wird dies auch in der zugehörigen Darstellung des Attributnamens reflektiert.

In der Abbildung ist eine tplLabel verwendende Repräsentation der Vorlesungstabelle (siehe Beispiele 3.10 auf Seite 87 und 3.12 auf Seite 89) in zwei Situationen mit verschiedenen Fingerpositionen dargestellt. In Abb. 4.4(a) zeigt der Finger auf die Menge Students Ⓐ. Der Finger ist durch die Färbung der ganzen Menge hervorgehoben. Die Menge ist selbst eine Tupelkomponente, der Komponentenattributname Ⓑ ist ebenfalls koloriert. Weiterhin ist zu sehen, daß auch der Rollbalken Ⓒ der Konstruktions-Visualisierungsfunktion koloriert ist. In Abb. 4.4(b) zeigt der Finger auf ein Mengenelement Ⓓ. Der Komponentenattributname, die Menge und der Rollbalken sind nicht mehr gefärbt. □

Parameter:

—

Berechnung:

1. Bestimme zu F den Visualisierungsknoten v, das Datenobjekt o, die Fingerliste $\vec{F}$, den Viewport Π und den Offset Ω.
2. Führe die Finger-Methode von v mit den Parametern F, o, Π und Ω aus.

Rückgabe:

—

Methode Finger

Klasse:

visNode

Objekt:

Visualisierungsknoten v

Parameter:

Fingerliste $\vec{F}$, Datenobjekt o, Viewport Π, Offset Ω

Berechnung:

1. Bestimme die Identifikation id von (o, v) und die Visualisierungsfunktion f von v.
2. Bestimme die Fingerart k für o aus $\vec{F}$.
3. Wenn o Sub-Objekt von o_p ist, dann:
   - 3.1. Bestimme den Vater-Visualisierungsknoten v_p von v.
   - 3.2. Bestimme die Identifikation id_p von (o_p, v_p).
   - 3.3. Bestimme die Visualisierungsfunktion f_p von v_p.
   - 3.4. Führe die Sub_Finger-Methode von f_p mit den Parametern o_p, v_p, id_p, o, v, id und k aus.
4. Führe die Finger-Methode von f mit den Parametern o, v, id und k aus.
5. Wenn f eine Tupel-Visualisierungsfunktion ist, dann:
   - 5.1. Führe die First-Methode von f mit den Parametern o, v, id, Π und Ω aus und bestimme den Nachfolgerknoten v^1_{sub} der im Sinne von f ersten visualisierten Komponente von v und den aktualisierten Offset Ω^1.
   - 5.2. Solange eine Fingervisualisierung zu verändern ist:
     - 5.2.1. Bestimme das v^i_{sub} entsprechende Sub-Objekt o^i_{sub} von o
     - 5.2.2. Führe die Finger-Methode von v^i_{sub} mit den Parametern $\vec{F}$, o^i_{sub}, Π und Ω^i aus.
     - 5.2.3. Bestimme die Identifikation id^i_{sub} von (o^i_{sub}, v^i_{sub}).

5.2.4. Führe die Iterate-Methode von f mit den Parametern o, v, id, o^i_{sub}, v^i_{sub}, id^i_{sub}, Π und Ω^i aus und bestimme den Nachfolgerknoten v^{i+1}_{sub} der im Sinne von f nächsten visualisierten Komponente von v und den aktualisierten Offset Ω^{i+1}.

6. Wenn f eine Kollektions-Visualisierungsfunktion ist, dann:
   6.1. Bestimme den Nachfolgerknoten v_{sub} von v.
   6.2. Führe die First-Methode von f mit den Parametern o, v und id aus und bestimme das im Sinne von f erste visualisierte Element (Sub-Objekt) o^1_{sub} von o und den aktualisierten Offset Ω^1.
   6.3. Solange eine Fingervisualisierung zu verändern ist:
      6.3.1. Führe die Finger-Methode von v_{sub} mit den Parametern $\vec{F}$, o^i_{sub}, Π und Ω^i aus.
      6.3.2. Bestimme die Identifikation id^i_{sub} von (o^i_{sub}, v_{sub}).
      6.3.3. Führe die Iterate-Methode von f mit den Parametern o, v, id, o^i_{sub}, v^i_{sub}, id^i_{sub}, Π, Ω^i und s^i aus und bestimme das im Sinne von f nächste visualisierte Element (Sub-Objekt) o^{i+1}_{sub} von o und den aktualisierten Offset Ω^{i+1}.

7. Wenn f eine Konstruktor-Visualisierungsfunktion ist, dann:
   7.1. Bestimme den Nachfolgerknoten v_{sub} von v
   7.2. Führe die Finger-Methode von v_{sub} mit den Parametern $\vec{F}$, o, v_{sub}, Π und Ω aus.

Rückgabe:

—

Methode Finger

Klasse:

visFunc

Objekt:

Visualisierungsfunktion f

Parameter:

Datenobjekt o, Visualisierungsknoten v, Identifikation id, Fingerart k

Aufgabe:

Visualisiere einen Finger in id gemäß der Fingerart k.

Rückgabe:

—

Wird ein Finger bewegt, kann das auch Auswirkungen auf andere in der Repräsentation visualisierte Finger haben. Z. B. können Teile eins Fingers, die vorher verdeckt waren, sichtbar werden. Es sollen aber nicht *alle* Fingervisualisierungen jedesmal neu berechnet werden,

wenn *ein* Finger bewegt wird. Vielmehr muß ein möglichst minimales „Finger-Delta“ berechnet und ausgeführt werden. Dieses Finger-Delta ist von der Fingerbewegung abhängig: Nach einer *in*- bzw. *out*-Operation muß der jeweils umfangreichere Bereich (d. h. die Fingerposition vor dem *in* bzw. nach dem *out*) der Repräsentation aktualisiert werden, nach einer *move*-Bewegung muß die Repräsentation sowohl an der alten als auch an der neuen Fingerposition aktualisiert werden. Dieses wurde in Abschnitt 4.3.1 bei der Formulierung der Interaktionsmethoden von Fingern bereits berücksichtigt.

Werden Teile einer Repräsentation durch den Mechanismus der späten Visualisierung (siehe Abschnitt 4.1.1) erst nachträglich generiert, müssen in diesen auch die Finger dargestellt werden. Dieser Aspekt wurde in den Abschnitten 4.2.1 und 4.2.2 bei der Formulierung der Visualisierungsknoten-Methoden Pre und Post weggelassen, da zu viele erklärende Vorgriffe notwendig gewesen wären. Im Hinblick auf den hiesigen Abschnitt wird klar, daß die Fingervisualisierung recht einfach integriert werden kann. Die Post-Methode muß lediglich Zugriff auf die Fingerliste haben, um die Fingerart für eine gerade generierte Repräsentation zu bestimmen. Die jeweilige Visualisierungsfunktionen-Methode Finger wird dann am Ende der Post-Methode (nach Schritt 6 auf Seite 132) aufgerufen.

Diese Integration der Fingervisualisierung beinhaltet auch, daß durch die Bereitstellung einer Fingerliste vor der Generierung einer Repräsentation die durch die Liste definierten Finger sofort dargestellt werden. Dieser Aspekt ist z. B. bei der Generierung von Repräsentationen als Anzeige von Link-Zielen wichtig.

In den folgenden Abschnitten werden die Begriffe „Fingergröße“ und „Finger-Offset“ verwendet. Damit sind die Größe und der Offset der Fingervisualisierung in der Repräsentation eines Datenobjektes gemeint.

4.3.4 Anpassung des Viewports

Wenn ein Finger auf ein (direktes oder indirektes) Sub-Objekt eines komplexen Datenobjektes zeigt, dessen Repräsentation veränderbare Viewports enthält, dann kann die Auswirkung einer Fingerbewegung sein, daß der Finger den sichtbaren Bereich verläßt. Das gilt nicht nur für den offensichtlichen Fall, daß am Rand eines Viewports eine *next*- oder *back*-Operation ausgeführt wird. Auch durch eine *in*- oder *out*-Operation kann eine Anpassung des Viewports notwendig sein, wie in Abb. 4.5 auf der nächsten Seite an einem Beispiel gezeigt wird.

Ein Verfahren wird benötigt, das für Repräsentation mit veränderbaren Viewports diese so anpassen kann, daß ein Finger sichtbar bleibt. Nach einer Fingerbewegung muß dieses Verfahren angewendet werden, um die Sichtbarkeit des Fingers sicherzustellen. Das Verfahren muß dabei alle umgebenden Viewports betrachten, da nicht nur jeder umgebende sichtbare Ausschnitt Ursache der Verdeckung des Fingers sein kann, sondern auch mehrere umgebende, ineinander geschachtelte Viewports Ursachen der Verdeckung sein können. Ein Beispiel dazu ist in Abb. 4.6 auf Seite 165 dargestellt.

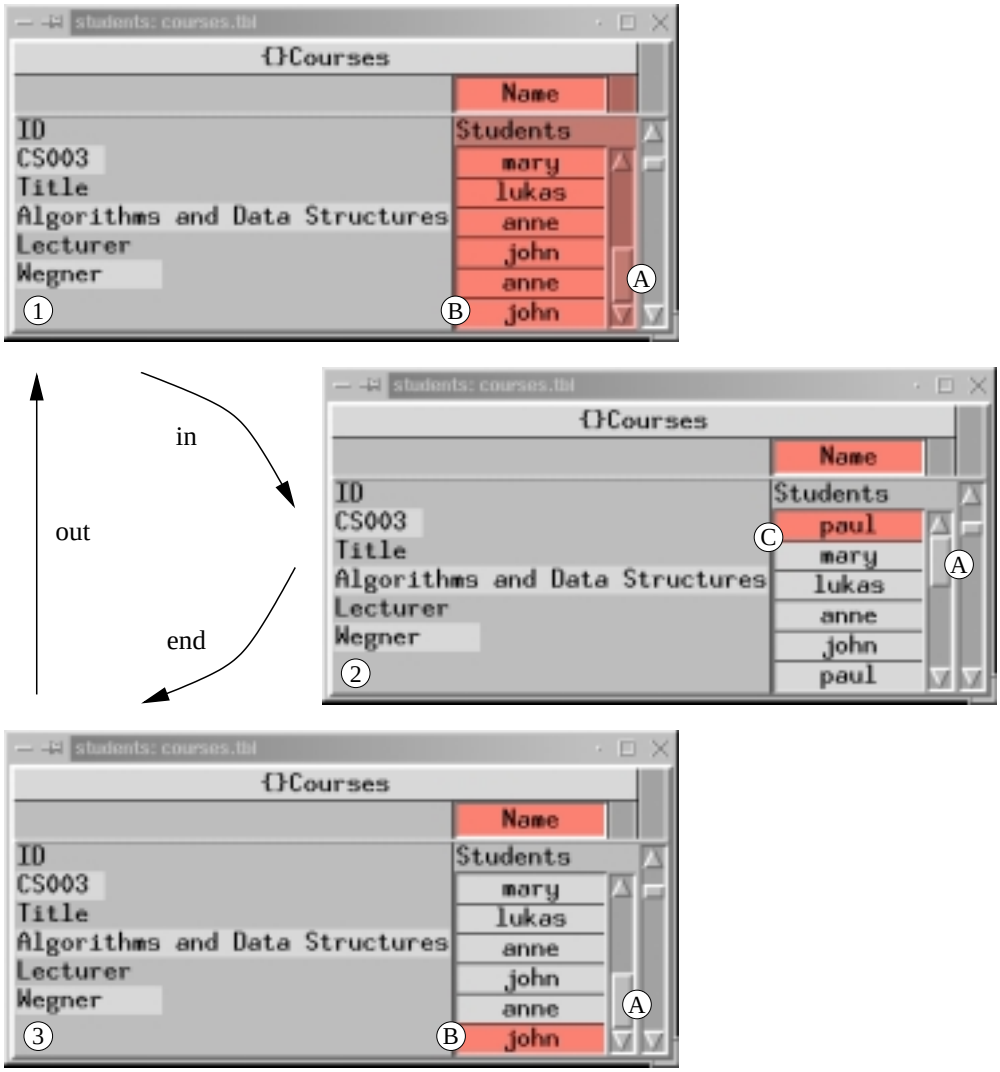
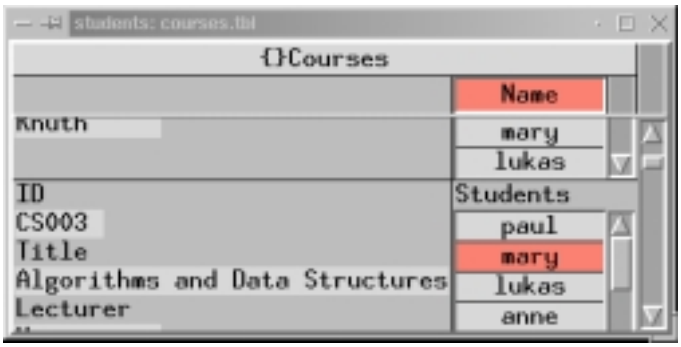


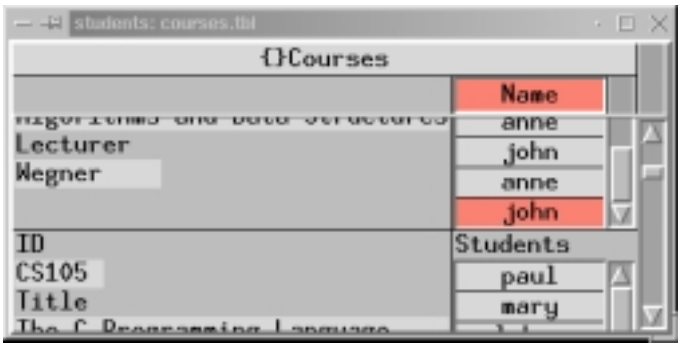
Abbildung 4.5: Viewport-Anpassung nach *in*- und *out*-Operationen

Beispiel 4.5 Viewport-Anpassung nach *in*- und *out*-Operationen

In der Abbildung zeigt der Finger vor der *in*-Operation ① auf die Menge Students, der Viewport ist so eingestellt, daß das Ende der Menge sichtbar ist (siehe Rollbalken Ⓐ). Nach der *in*-Operation ② zeigt der Finger auf das erste Mengenelement Ⓒ (Student „paul“). Der Viewport wurde so angepaßt, daß der Anfang der Menge und damit ihr erstes Element sichtbar ist. Nach der *last*-Operation ③ zeigt der Finger auf das letzte Mengenelement Ⓑ (Student „john“), wodurch wieder das Ende der Menge sichtbar wird. Die folgende *out*-Operation stellt wieder Situation ① her und verändert den Viewport nicht. □



(a) Vor der Fingerbewegung (Fingeroperation *last*)



(b) Nach der Fingerbewegung

Abbildung 4.6: Anpassung geschachtelter Viewports

Beispiel 4.6 Anpassung geschachtelter Viewports

Die Abbildung zeigt eine Repräsentation der Visualisierung „teilnehmende Studenten“ (siehe Beispiel 3.10 auf Seite 87) in zwei Situationen, jeweils vor und nach der Bewegung eines Fingers (Fingeroperation *last*). Die beide Mengen Courses und Students sind von Viewports umgeben, die vertikal adjustierbar sind. In Abb. 4.6(a) (vor der Bewegung) zeigt der Finger zunächst auf die Studentin „mary“, die an der Vorlesung „Algorithms and Data Structures“ teilnimmt. Der direkt umgebende Students-Viewport so eingestellt, daß der Anfang der Menge Students sichtbar ist. Der äußere Courses-Viewport ist so eingestellt, daß noch ein Teil der vorhergehenden Vorlesung sichtbar ist. In Abb. 4.6(b) (nach der Bewegung) zeigt der Finger auf den Studenten „john“ derselben Vorlesung, dem letzten Element der Menge Students. Der Students-Viewport ist jetzt so eingestellt, daß das Ende der Menge Students sichtbar ist. Da der Students-Viewport in Abb. 4.6(a) im Courses-Viewport nicht ganz sichtbar ist, wäre der Finger ohne Anpassung des Courses-Viewports selbst nicht sichtbar. Durch dessen Anpassung ist jetzt das Ende des Students-Viewports sichtbar, und damit auch der Finger. □

Das Verfahren der Viewport-Anpassung wird also durch die Struktur der Visualisierung gesteuert. Daher wird es als Methode Adjust der Visualisierungsknoten entwickelt, die entsprechende Adjust-Methoden der Visualisierungsfunktionen aufrufen.

Viewports können unabhängig vom Finger verändert werden, z. B. indem ihr Rollbalken mit der Maus direkt manipuliert wird. Zeigen mehrere Finger in eine Repräsentation mit Viewports, kann ein Wechsel des aktiven Fingers (die Aktivierung eines anderen Fingers, siehe Abschnitt 3.1.3) eine Viewport-Anpassung notwendig machen, damit der neue aktive Finger sichtbar ist.

Zur Realisierung der Methode muß jede komplexe Visualisierungsfunktion, die einen adjustierbaren Viewport realisiert, eine Adjust-Methode implementieren[15]. Außerdem muß jede komplexe Visualisierungsfunktion eine Position-Methode zur Verfügung stellen, die die Größe der durch sie generierten Repräsentation, sowie die Größe der Repräsentationen von Sub-Objekten und deren relative Lage, d. h. ihren Offset in der Repräsentation des komplexen Objektes berechnet. Diese Methoden werden weiter unten beschrieben.

Die Adjust-Methode eines Visualisierungsknotens v durchläuft die Visualisierung $T = (V, E, r)$, für die $v \in V$ gilt, von v ausgehend, aufsteigend bis zur Wurzel r. Dabei muß der Finger-Offset, der zunächst gleich dem Offset der v entsprechenden Repräsentation in der umgebenden Repräsentation ist, bei jeder Viewport-Anpassung ebenfalls korrigiert und für jeden Vater-Visualisierungsknoten v' an dessen Offset angepaßt werden. Die Offset-Korrektur wird in Abb. 4.7 auf der gegenüberliegenden Seite demonstriert.

Methode Adjust

Klasse:
visNode
Objekt:
Visualisierungsknoten v
Parameter:
Datenobjekt o
Berechnung:

1. Wenn v Wurzel der Visualisierung ist, dann Ende.
2. Bestimme den Vater-Visualisierungsknoten v' von v und das o umgebende Datenobjekt o'.
3. Bestimme die Visualisierungsfunktion f' von v'.
4. Bestimme die Identifikationen id von (o, v) und id' von (o', v').
5. Führe die Position-Methode von f' mit den Parametern o', v', id', o, v und id aus und bestimme die Größen $\vec{s}'$ und $\vec{s}$ von id' bzw. id sowie die relative Lage $\vec{p}$ von id in id'.

[15] Komplexe Visualisierungsfunktionen, die keinen adjustierbaren Viewport realisieren, müssen auch eine Adjust-Methode zur Verfügung stellen. Deren Implementierung kann aber leer sein.

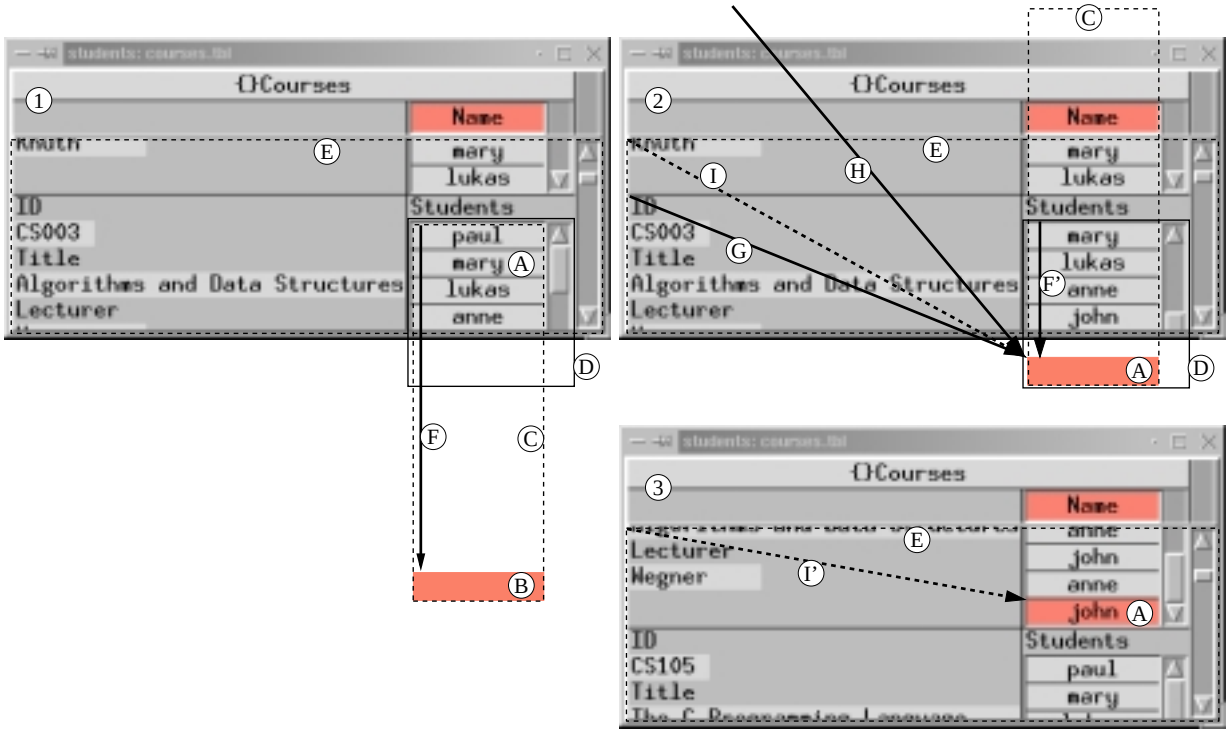


Abbildung 4.7: Offset-Korrektur bei Viewport-Anpassung

Beispiel 4.7 Offset-Korrektur bei Viewport-Anpassung

Die Abbildung zeigt neben dem Ergebnis der Viewport-Anpassung ③ schematisch zwei Zwischenschritte ① und ②, die dem Benutzer nicht präsentiert werden.

Nach dem Bewegen des Fingers, aber vor der Viewport-Anpassung ① ist der Finger von seiner Ausgangsposition Ⓐ an das Ende Ⓑ der Students-Menge Ⓒ gewandert, er ist jetzt außerhalb des Students-Viewports Ⓓ. Der Offset von Ⓑ in Ⓒ ist durch den Vektor Ⓕ angedeutet. Er ist gleich dem Offset von Ⓑ in Ⓓ, da der Offset von Ⓒ in Ⓓ Null ist. Die Adjust-Methode für Ⓓ erkennt am Offset Ⓕ von Ⓑ, daß der Finger momentan nicht sichtbar ist, und stellt den Viewport so ein, daß er das Ende der Students-Menge zeigt.

In dieser Situation ② ist der Finger Ⓐ mit der Students-Menge nach oben gerollt, wodurch sich dessen Offset Ⓕ′ in Ⓓ „verkürzt“. Die Offsets von Ⓐ im Course-Tupel, in der Courses-Menge und im Courses-Viewport Ⓔ sind durch die Vektoren Ⓖ, Ⓗ und Ⓘ angedeutet (der Anfang der Courses-Menge und damit der Startpunkt von Ⓗ ist nicht sichtbar). Die Adjust-Methode des Courses-Viewports erkennt am Offset Ⓘ, daß der Finger Ⓐ momentan nicht sichtbar ist, und stellt den Viewport entsprechend so ein, daß Ⓐ in ihm zentriert ist.

Im Ergebnis ③ ist der Finger Ⓐ mit der Courses-Menge nach oben gerollt, wodurch sich wiederum dessen Offset Ⓘ′ in Ⓔ „verkürzt“ hat. □

6. Speichere $\vec{s}$ und $\vec{p}$ als Fingergröße $\vec{s}_f$ bzw. Finger-Offset $\vec{o}_f$
7. Führe die Adjust-Methode von f' mit den Parametern o', v', id', id, $\vec{s}'$, $\vec{s}$, $\vec{p}$, $\vec{o}_f$ und $\vec{s}_f$ aus und bestimme $\vec{o}_f$ neu.
8. Solange v' nicht die Wurzel der Visualisierung ist:
   8.1. Ersetze o, v und id durch o', v' und id'.
   8.2. Bestimme den Vater-Visualisierungsknoten v' von v.
   8.3. Bestimme die Visualisierungsfunktion f' von v'.
   8.4. Wenn f' keine Konstruktions-Visualisierungsfunktion ist, dann bestimme das o umgebende Datenobjekt o' (ansonsten sind o' und o identisch).
   8.5. Bestimme die Identifikation id' von (o', v').
   8.6. Führe die Position-Methode von f' mit den Parametern o', v', id', o, v und id aus und bestimme die Größen $\vec{s}'$ und $\vec{s}$ von id' bzw. id sowie die relative Lage $\vec{p}$ von id in id'.
   8.7. Erhöhe den Finger-Offset $\vec{o}_f$ um $\vec{p}$.
   8.8. Führe die Adjust-Methode von f' mit den Parametern o', v', id', id, $\vec{s}'$, $\vec{s}$, $\vec{p}$, $\vec{o}_f$ und $\vec{s}_f$ aus und bestimme $\vec{o}_f$ neu.

Rückgabe:
—

Visualisierungsfunktionen, die einen adjustierbaren Viewport realisieren, müssen eine Adjust-Methode implementieren. Aufgabe dieser Methode ist es, die Sichtbarkeit eines Fingers in einem Viewport ggf. durch Adjustierung zu gewährleisten. Die Sichtbarkeit eines Fingers ist abhängig von der Größe des Viewports, der Größe der im Viewport enthaltenen Repräsentation, der Lage der Repräsentation im Viewport, der Fingergröße und dem Finger-Offset. Das Adjustieren des Viewports besteht darin, die Lage der Repräsentation im Viewport geeignet zu verändern.

Eine Adjustierung ist nur notwendig, falls der Finger nicht sichtbar ist oder teilweise vom Viewport verdeckt wird. Die Strategie, nach der die Adjustierung erfolgt, ist letztendlich von der Implementierung der Adjust-Methode abhängig. Eine plausible Startegie wird im Folgenden beschrieben. Sie kann auf alle adjustierbaren Dimensionen angewendet werden.

- Die Änderung der Lage des Datenobjektes im Viewport soll möglichst gering sein.
- Sie soll wenn möglich den visuellen Kontext erhalten, d. h. ein vor dem Adjustieren sichtbarer Teil sollte auch nach dem Adjustieren sichtbar sein, wenn dies mit den anderen Bedingungen vereinbar ist.
- Sie soll eine zu häufige Viewport-Anpassung vermeiden. Beim Navigieren in einer Kollektion würde diese Forderung z. B. bedeuten, daß nach dem Verlassen des Viewports mit einer *next*-Operation nicht nur das nächste Element in den Viewport geholt

wird, sondern mehrere Elemente. Dadurch ergibt sich ein Konflikt mit der ersten Forderung, der aber durch die Erfüllung der zweiten Forderung gemildert wird.

Konkret kann die Strategie umgesetzt werden, indem die Größe und Lage des Fingers mit der Größe und Lage des Viewports verglichen werden.

Einfache Strategie: Ist der Finger vollständig sichtbar, ist nichts zu tun. Ist der Finger größer als der Viewport, werden die Anfangskanten von Finger und Viewport in Deckung gebracht. Ansonsten wird nach der Lage des verdeckten Fingerteils, relativ zum Viewport, entschieden: liegt er oberhalb, werden die Anfangskanten in Deckung gebracht, liegt er unterhalb, werden die Endkanten in Deckung gebracht.

Bessere, aufwendigere Strategie: Ist der Finger vollständig sichtbar, ist nichts zu tun. Ansonsten wird nach der Fingergröße relativ zur Größe des Viewports entschieden: Ist der Viewport kleiner, wird das Kantenpaar mit dem geringeren Abstand in Deckung gebracht, so daß der Finger den Viewport ausfüllt; liegt der Finger mehr oberhalb des Viewports, sind dies die unteren Kanten, liegt er mehr unterhalb, sind dies die oberen Kanten. Ist der Finger kleiner als der Viewport, wird zusätzlich nach der Lage des Fingers zum Viewport entschieden: Ist der Finger vollständig verdeckt, wird er im Viewport zentriert. Ansonsten wird die verdeckte Kante des Fingers mit der entsprechenden Kanten des Viewports in Deckung gebracht, so daß der Finger vollständig sichtbar ist; ist der Finger oben verdeckt, sind dies die oberen Kanten, ist der Finger unten verdeckt, sind dies die unteren Kanten.

Die Schnittstelle komplexer Visualisierungsfunktionen wird um die Methoden Adjust und Position erweitert.

Methode Adjust

Klasse:
cplxVF

Objekt:
komplexe Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Identifikation id_{sub}, Visualisierungsgröße $\vec{s}$, Sub-Visualisierungsgröße $\vec{s}_{\text{sub}}$, Lage $\vec{p}$, Fingergröße $\vec{s}_f$, Finger-Offset $\vec{o}_f$

Aufgabe:
Stelle sicher, daß der Finger, dessen Lage in id_{sub} durch $\vec{s}_f$ und $\vec{o}_f$ beschrieben ist, im Viewport von id sichtbar ist. Wenn der Viewport dazu adjustiert wurde, dann berechne den aktualisierten Finger-Offset $\vec{o}'_f$.

Rückgabe:
Finger-Offset $\vec{o}'_f$

Methode Position

Klasse:
cplxVF

Objekt:
komplexe Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Sub-Objekt o_{sub}, Nachfolgerknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub})

Aufgabe:
Bestimme die Größen $\vec{s}$ von id und $\vec{s}_{sub}$ von id_{sub} sowie die relative Lage $\vec{p}$ von id_{sub} in id.

Rückgabe:
Größen $\vec{s}$ und $\vec{s}_{sub}$ und Position $\vec{p}$

4.3.5 Größenänderungen

Bisher wurden Repräsentationen als „statisch“ in dem Sinn betrachtet, daß die Repräsentation eines Datenobjektes generiert und nicht mehr verändert wurde (siehe Abschnitt 4.2). Durch die Manipulation komplexer Datenobjekte (siehe Abschnitt 4.3) können aber auch Größenänderungen auftreten, die zu Veränderungen der Repräsentationen führen müssen. Dieser Aspekt wird nun näher betrachtet.

In Abschnitt 4.3.1 wurden zwei Klassen von Fingeroperationen beschrieben, die durch die Manipulation von Datenobjekten zu Größenänderungen führen können:

- Änderung eines atomaren Wertes und
- die Kardinalität von Kollektionen ändernde Operationen.

Diese Größenänderung der Repräsentation eines (Sub-) Datenobjektes kann sich auf die umgebenden Repräsentationen auswirken. Alle Methoden von Visualisierungsfunktionen, die die Änderung von Repräsentation implementieren, und dabei eine Größenänderung hervorrufen, berechnen ein Resize-Delta, anhand dessen die umgebenden Repräsentationen aktualisiert werden können.

Für atomare Visualisierungsfunktionen muß die Edit-Methode dieses Delta berechnen (siehe Abschnitt 5.3). Methoden, die die Kardinalität von Kollektionen und damit die Größe ihrer Repräsentationen verändern können, sind Insert, Delete, Null_To_Empty und Empty_To_Null sowie Empty_To_Singleton und Singleton_To_Empty. Nachdem eine dieser Methoden ausgeführt wurde, muß ein Verfahren eingeleitet werden, das vom umgebenden Visualisierungsknoten ausgehend die Visualisierung aufsteigend bis zur Wurzel durchläuft. Dabei muß die Größenänderung für jeden besuchten Visualisierungsknoten betrachtet werden.

Wie schon die Viewport-Anpassung (siehe Abschnitt 4.3.4) wird also auch das Verfahren zur Behandlung von Größenänderungen durch die Struktur der Visualisierung gesteuert. Es wird daher ebenfalls als Methode der Visualisierungsknoten entwickelt, hier namens Resize. Auch alle komplexen Visualisierungsfunktionen müssen eine Resize-Methode implementieren (die Traversierung der Visualisierung beginnt beim umgebenden Visualisierungsknoten und besucht daher nur innere Visualisierungsknoten, die immer zu komplexen Visualisierungsfunktionen gehören). Diese werden von der Resize-Methode der Visualisierungsknoten bei der Traversierung aufgerufen.

Die Resize-Methode einer Visualisierungsfunktion verändert falls notwendig die entsprechende Repräsentation und verkleinert das Resize-Delta, falls die Größenänderung, ggf. auch nur teilweise kompensiert werden kann. Beispiele für Größenänderungen und ihre Auswirkungen sind in Abb. 4.8 auf der nächsten Seite und Abb. 4.9 auf Seite 173 dargestellt.

Diese Abhängigkeit der Auswirkung von inneren Höhenänderungen nach außen muß von den Resize-Implementierungen der Visualisierungsfunktionen berücksichtigt und durch die Veränderung des Resize-Deltas realisiert werden.

Methode Resize

Klasse:
visNode

Objekt:
Visualisierungsknoten v

Parameter:
Datenobjekt o, Resize-Delta $\vec{\Delta}_r$

Berechnung:

1. Wenn $\vec{\Delta}_r = \vec{0}$ ist, dann Ende.
2. Wenn v Wurzel der Visualisierung ist, dann Ende.
3. Bestimme die Identifikation id von (o, v)
4. Solange v nicht die Wurzel der Visualisierung ist:
   - 4.1. Bestimme den Vater-Visualisierungsknoten v' von v.
   - 4.2. Bestimme die Visualisierungsfunktion f' von v'.
   - 4.3. Bestimme das zu v' gehörige Datenobjekt o'.
   - 4.4. Bestimme die Identifikation id' von (o', v').
   - 4.5. Führe die Resize-Methode von f' mit den Parametern o', v', id', o, v, id und $\vec{\Delta}_r$, aus und bestimme das aktualisierte Resize-Delta $\vec{\Delta}_r$.
   - 4.6. Wenn $\vec{\Delta}_r = \vec{0}$ ist, dann Ende.
   - 4.7. Ersetze o, v und id durch o', v' und id'.

Rückgabe:
—

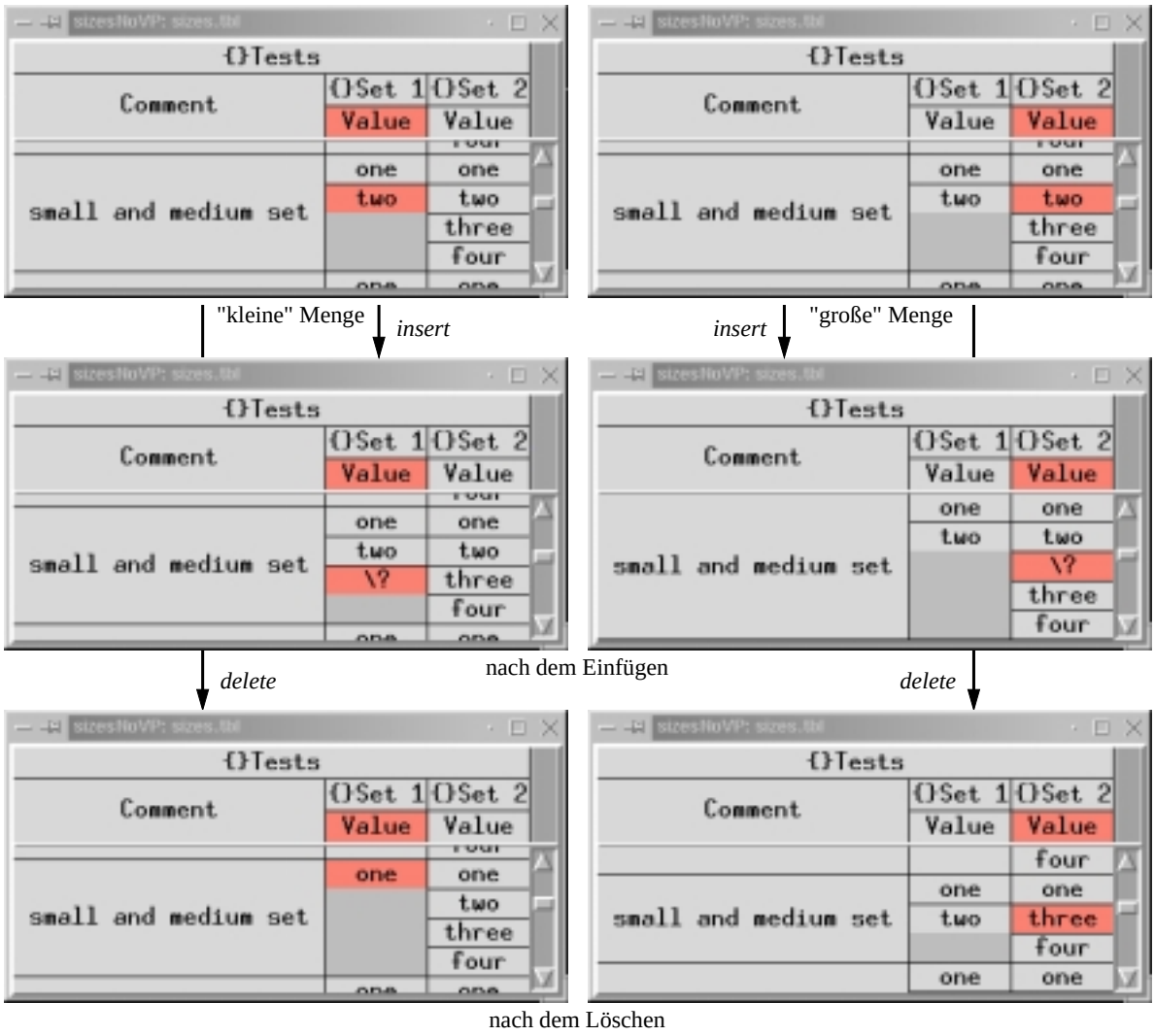


Abbildung 4.8: Auswirkung von Höhenänderungen (1)

Beispiel 4.8 Auswirkung von Höhenänderungen (1)

In der Abbildung haben die Elemente der Tests-Mengen selber zwei Mengen als Komponenten (Set 1 und Set 2). Die Darstellungen entsprechen der Default-Visualisierung.

Ist diese Menge die höchste Komponente des Tupels (rechte Seite), bestimmt sie dessen Höhe. Eine Änderung der Höhe durch das Löschen oder Einfügen eines Mengenelementes wirkt sich direkt auf die Höhe des Tupels aus. Ist diese Menge nicht die höchste Komponente des Tupels (linke Seite), bestimmt sie dessen Höhe nicht. Eine Änderung der Höhe durch das Löschen eines Mengenelementes wirkt sich nicht auf die Höhe des Tupels aus. Nur wenn die Menge durch das Einfügen eines Elementes zur höchsten Komponente des Tupels würde, würde sich die Höhe des Tupels vergrößern. □

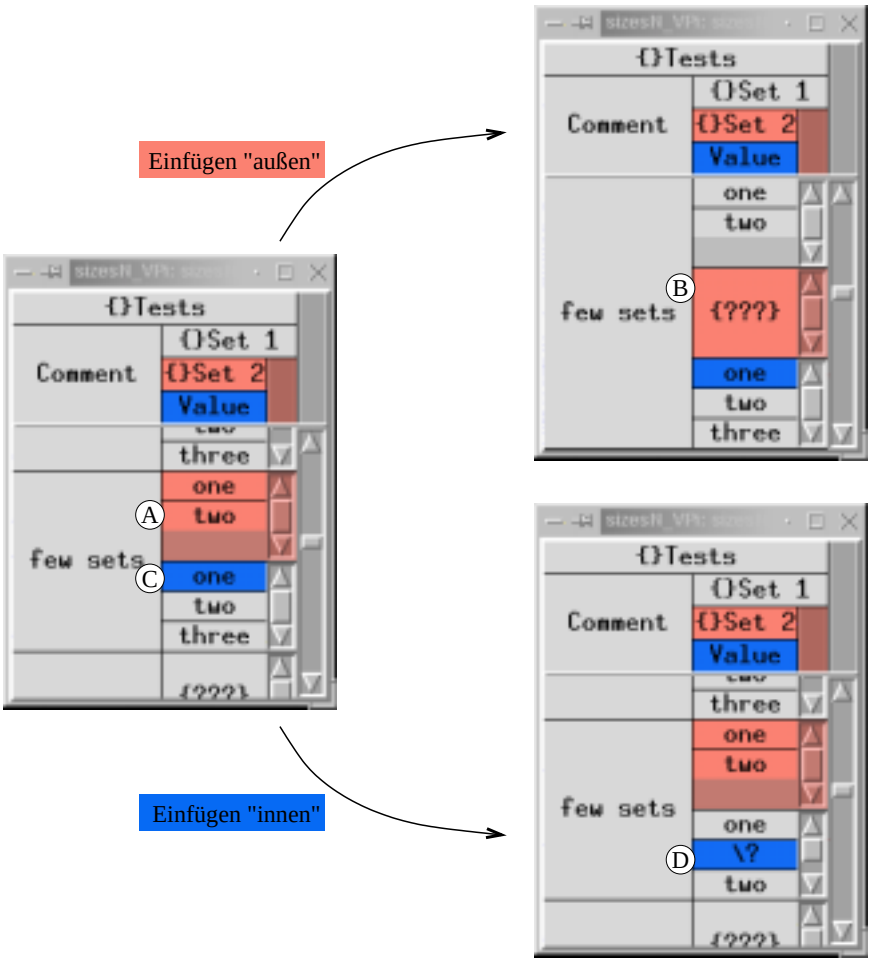


Abbildung 4.9: Auswirkung von Höhenänderungen (2)

Beispiel 4.9 Auswirkung von Höhenänderungen (2)

In der Abbildung ist der Elementtyp der Menge Set 1 selbst eine Menge (Set 2). Die inneren Mengen sind mit vertikal rollbaren Viewports dargestellt.

Wird in die äußere Menge eingefügt, wirkt sich die Höhenänderung der Menge direkt auf die Höhe des Tupels aus. Rechts oben ① wurde nach Element Ⓐ der äußeren Menge die nullwertige Menge Ⓑ eingefügt. Wird in die innere Menge eingefügt, wird die Höhenänderung der Menge von der Viewport-Visualisierungsfunktion kompensiert. Rechts unten ② wurde nach dem Element Ⓒ der inneren Menge der nullwertige String Ⓓ eingefügt. □

Methode Resize

Klasse:
cplxVF

Objekt:
komplexe Visualisierungsfunktion f

Parameter:
Datenobjekt o, Visualisierungsknoten v, Identifikation id von (o, v), Sub-Objekt o_{sub}, Nachfolgerknoten v_{sub}, Identifikation id_{sub} von (o_{sub}, v_{sub}), Resize-Delta $\vec{\Delta}_r$

Aufgabe:
Ändere id so, daß die Repräsentation die Größenänderung $\vec{\Delta}_r$ von id_{sub} reflektiert, und aktualisiere ggf. $\vec{\Delta}_r$.

Rückgabe:
Resize-Delta $\vec{\Delta}_r$

Kapitel 5

Visualisierungsfunktionen

In diesem Kapitel werden, als letzter Baustein des Visualisierungsverfahrens, die Visualisierungsfunktionen genauer betrachtet. Bisher wurden die vier Visualisierungsklassen durch eine grobe Beschreibung ihrer Methoden konkretisiert, die jeweils die zugehörige Signatur und die zu erfüllende Aufgabe umfaßte. Hier soll zunächst beschrieben werden, wie für eine Visualisierungsklasse eine Instanz angelegt werden kann, d. h. welche Aufgaben bei der Implementierung einer Visualisierungsfunktion bearbeitet werden müssen (Abschnitt 5.1). In den folgenden Abschnitten werden dann exemplarisch einige Realisierungen von Visualisierungsfunktionen genauer beschrieben. Dabei wird gemäß der in Abschnitt 3.3.1 definierten Klassenhierarchie zwischen atomaren, Konstruktions-, Tupel- und Kollektions-Visualisierungsfunktionen unterschieden.

Insbesondere wird für jede datenbezogene Visualisierungsklasse eine Default-Visualisierungsfunktion (siehe Abschnitt 3.6.1) beschrieben. Weiterhin wird in Abschnitt 5.5.1 eine Konstruktions-Visualisierungsfunktion beschrieben, die einen vertikal rollbaren Viewport realisiert. Zusammen mit dem in Abschnitt 3.6.1 beschriebenen Algorithmus zur Erstellung einer Default-Visualisierung ist damit die Anforderung der Repräsentation beliebiger Datenobjekte ohne Benutzerkonfiguration erfüllt.

5.1 Implementierung einer Visualisierungsfunktion

Das Implementieren einer Visualisierungsfunktion ist gleichbedeutend mit der Erweiterung des Visualisierungsverfahrens. Eine neue Visualisierungsfunktion ist nur notwendig, wenn eine gewünschte Darstellung mit den bisher vorhandenen Mitteln, d. h. Visualisierungsfunktionen nicht oder nur unzureichend bzw. ineffizient erreicht werden kann.

Sollte sich bei der Implementierung einer Visualisierungsfunktion herausstellen, daß die gewünschte Darstellung nicht erreicht werden kann, muß ggf. das Visualisierungsverfahren durch eine Änderung der Klassenhierarchie erweitert werden (siehe Abschnitt 3.9.3). Diese

Erweiterung ist jedoch, verglichen mit der Implementierung einer Visualisierungsfunktion, erheblich aufwendiger, da auch die Methoden der Visualisierungsknoten angepaßt werden müssen.

Soll ein konkreter Darstellungsaspekt durch die Implementierung einer Visualisierungsfunktion erreicht werden, muß zunächst die Visualisierungsklasse bestimmt werden, aus der die Visualisierungsfunktion instantiiert wird. Diese Aufgabe ist unproblematisch, da die Unzulänglichkeit einer bereits existierenden Visualisierungsfunktion meistens zu einer neuen Visualisierungsfunktion in der gleichen Visualisierungsklasse führt.

Oft wird eine neue Visualisierungsfunktion durch Verfeinerung der Implementierung einer existierenden Visualisierungsfunktion erstellt werden können. Abhängig von der Umsetzung des Visualisierungsverfahrens kann dies durch Vererbungsmechanismen in objekt-orientierten Umgebungen oder muß durch Wiederverwendung von Codefragmenten geschehen.

Zentraler Punkt ist dann die Implementierung des Interface, das durch die Visualisierungsklasse definiert ist. Dabei müssen die Aufgaben, die für die einzelnen Methoden in den Kapiteln 3 und 4 beschrieben wurden, soweit sie anwendbar sind, erfüllt werden. Methoden, die nicht anwendbar sind, können eine leere Implementierung behalten. Das ist z. B. für die Adjust-Methoden von Visualisierungsfunktion, die keinen adjustierbaren Viewport realisieren, der Fall.

Für komplexe, datenbezogene Visualisierungsfunktionen können weitere Methoden zum Zweck der Interaktion hinzukommen, wie das z. B. mit den gruppenbezogenen Move-Methoden von tplLabel der Fall ist (siehe Abschnitt 5.7.2).

Wie die Implementierung des Interface software-technisch realisiert wird, bleibt hier bewußt offen. Denkbar ist z. B. das Kopieren und Modifizieren bereits existierender Implementierungen, was einen eher konservativen Ansatz darstellen würde. Eine objekt-orientierte Methodik würde die Implementierungen von Methoden bereits existierender Visualisierungsfunktionen, ggf. nur teilweise, durch Redefinition bzw. Verfeinerung wiederverwenden.

Durch die Aufteilung der Funktionalität von Visualisierungsfunktionen auf kleine, überschaubare Module (die Methoden), deren Ausführung durch das generische Visualisierungsverfahren wie in Kapitel 4 beschrieben gesteuert wird, ist die Implementierung selbst einfach und überschaubar. Sie kann auch in mehreren Phasen vorgenommen werden, z. B. indem die Datenobjekt- und Fingerdarstellung sowie Navigations- und Änderungsmethoden jeweils einzeln entwickelt und getestet werden.

Als letzte Aufgabe müssen die Meta-Informationen über die neue Visualisierungsfunktion bestimmt und in die Visualisierungsfunktionen-Tabelle (siehe Abschnitt 3.8.2) eingetragen werden. Die Werte der Attribute Name, Class, Parameters, Bindings, SizeCap und SizeReq können dabei direkt aus der Implementierung bestimmt werden.

Für das Attribut Schema muß überlegt werden, welche existierende Schemavisualisierungsfunktion passend ist. Schlimmstenfalls muß eine weitere Visualisierungsfunktion als passen-

de Schemavisualisierungsfunktion implementiert werden[1]. Im Folgenden wird beispielhaft die Bestimmung von Meta-Informationen demonstriert:

Beispiel 5.1 Meta-Informationen für polygon

Die Attributwerte einer speziellen Visualisierungsfunktion polygon (siehe Abschnitt 5.3.3), die Polygone graphisch in einem Rechteck fester Größe darstellt , sind:

- Name: polygon.
- Class: Referenz auf atomVF, da polygon atomar ist.
- Parameters, Bindings: Siehe Abb. 5.7 auf Seite 191.
- SizeCap: „fix“ in beiden Dimensionen, da die Polygone in einem Rechteck mit festen Abmessungen dargestellt werden.
- SizeReq: $\perp$, da polygon atomar ist.
- Schema: Referenz auf atomAttr, da die Schemavisualisierung, analog zu anderen atomaren Visualisierungsfunktionen, durch den Attributnamen erfolgt. □

Beispiel 5.2 Schemareferenz für tplLabel

In Abschnitt 5.7.2 wird die Visualisierungsfunktion tplLabel beschrieben. Für tplLabel ist keine der in dieser Arbeit vorgestellten Visualisierungsfunktionen als Schemavisualisierungsfunktion geeignet. Vor dem Eintragen der Meta-Informationen für tplLabel in die Visualisierungsfunktionen-Tabelle muß also zunächst eine weitere Visualisierungsfunktion, z. B. namens tplLabelScm, definiert werden, die als Schemavisualisierungsfunktion für tplLabel passend ist. Da tplLabelScm eine Schemavisualisierungsfunktion ist, wird beim Eintragen der Meta-Informationen für tplLabelScm für das Attribut Schema ein Nullwert eingetragen. □

Sind diese Aufgaben bearbeitet, kann die neue Visualisierungsfunktion bereits benutzt werden, d. h. in Visualisierungen eingetragen und zur Generierung von Repräsentationen verwendet werden.

Ein Vorteil dieser Methodik ist sicherlich, daß ein Darstellungsproblem strukturiert und komponentenweise gelöst werden kann. Auch die Lösung der Teilprobleme ist durch die vorgegebenen Schnittstellen bereits strukturiert und kann wiederum in Unterprobleme zergliedert werden. Aspekte wie z. B. Größenänderungen und anpaßbare Viewports, die im Visualisierungsverfahren berücksichtigt werden, können durch diese Vorgehensweise auch bei der Implementierung von neuen Visualisierungsfunktionen nicht vergessen werden. Dadurch ist quasi eine „methodische Qualitätskontrolle“ realisiert worden.

[1] Für diese muß dann aber kein Wert für das Attribut Schema eingetragen werden.

5.2 Die Klasse visFunc (alle Visualisierungsfunktionen)

Für alle im Folgenden beschriebenen Visualisierungsfunktionen wird auf ihre Parameter (siehe Abschnitt 3.3.2) und Default-Bindungen (siehe Abschnitt 3.3.3) sowie die wichtigsten Aspekte zur Implementierung der Methoden eingegangen. Parameter werden tabellarisch beschrieben, wobei jeweils der Name, eine kurze Beschreibung, der Default-Wert und der Wertebereich angeben werden. Die Typen, die dabei verwendet werden, und ihre Wertebereiche sind in Tabelle 5.1 zusammengefaßt (Standardtypen, wie z. B. String, Boolean und Integer sind nicht aufgeführt).

Typ	Wertebereich
Alignment	`left`, `center`, `right`
Font	*plattformabhängig*
Wrap	`word`, `char`
Color	`white`, `black`, `gray`, ...
Relief	`flat`, `raised`, `sunken`
Horizontal	`left`, `right`
Vertical	`top`, `bottom`
SizeCap	`fix`, `dep`, `var`
SizeReq	`fix`, `none`

Tabelle 5.1: Typen

Bei Bindungen, die in diesem Kapitel angeben werden, wird für Ereignisse die Notation von Tcl/Tk [Ous94, S. 202 f.] benutzt. Für die Spezifikation der Operationen wird implizit davon ausgegangen, daß der aktive Finger durch F gegeben ist. Die Bindung des Ereignisses `<Enter>` an die Operation „F Push“ bedeutet also, daß die Push-Methode des aktiven Fingers aufgerufen wird, wenn die ↵-Taste (Enter, Return) gedrückt wird. Die Bindung selbst wird von der Post-Methode eines Visualisierungsknotens installiert (siehe Abschnitte 4.2.2 und 4.3).

Für Methoden werden, wenn es angemessen erscheint, kurze Codefragmente in der Syntax von Tcl/Tk bzw. Tcl/DB (siehe Abschnitt 2.3.4) angegeben.

In den Abbildungen, die die Einträge für die einzelnen Visualisierungsfunktionen in der Visualisierungsfunktionen-Tabelle (siehe Abschnitt 3.8.2) zeigen[2], wird für die Default-Werte von Parametern und Default-Bindungen von Tastaturereignissen ebenfalls die Syntax von Tcl/Tk verwendet. Dabei wird die Substitution von `%`-Sequenzen des Kommandos `bind` [Ous94, S. 204 f.] um die in Tabelle 5.2 auf der gegenüberliegenden Seite dargestellten Sequenzen erweitert. Werte bzw. Bindungen, die mit dem Zeichen „`[`“ beginnen, werden vor ihrer Interpretation mit `eval` [Ous94, S. 77 f.] als Tcl-Skript ausgeführt.

[2]Die Visualisierungsfunktionen-Tabelle ist in diesem Kapitel mit einer vereinfachten Schemavisualisierung dargestellt.

Sequenz	Substitution
%V	Visualisierungsknoten
%F	aktiver Finger
%D	Tabellen-Finger zum aktiven Finger

Tabelle 5.2: Substitution von %-Sequenzen

Zunächst werden die Methoden, die von allen Visualisierungsfunktionen implementiert werden müssen, besprochen. Sie bilden das Interface der virtuellen Klasse visFunc. Die Übersicht in Tabelle 5.3 enthält zusätzlich noch die Methode Pop, die nicht von allen, sondern nur von den datenbezogenen Visualisierungsfunktionen der Klasse dataVF implementiert wird.

Klasse	Methoden
visFunc	Width, Height
	Pre, Post
	Finger
dataVF	Pop

Tabelle 5.3: Interface der virtuellen Klassen visFunc und dataVF

Die Methoden Width und Height werden datenunabhängig bei der Generierung bzw. Initialisierung von Visualisierungen aufgerufen (siehe Abschnitt 3.5.2). Sie bekommen als Parameter einen Visualisierungsknoten übergeben, anhand dessen sie die Berechnung der Visualisierungsgrößen vornehmen, ggf. abhängig von der Parametrisierung des Visualisierungsknotens und bereits berechneten Größen von Nachfolgerknoten.

Die Methode Pre hat für alle Visualisierungsfunktionen die gleichen Parameter. Ihr werden ein Datenobjekt, ein Visualisierungsknoten, eine Identifikation, sowie Viewport und Offset übergeben. Sie gibt die im Voraus bestimmbaren Eigenschaften der Repräsentation des Datenobjektes zurück und signalisiert durch einen Boole'schen Wert, ob alle Eigenschaften bestimmt werden konnten.

Die Methode Post bekommt Datenobjekt, Visualisierungsknoten und Identifikation als Parameter übergeben und gibt die Eigenschaften der generierten Repräsentation zurück. Für komplexe Visualisierungsfunktionen werden zusätzlich Angaben über die Eigenschaften bereits integrierter Repräsentationen von Sub-Objekten übergeben. Die Methode Finger hat die Parameter Datenobjekt, Visualisierungsknoten, Identifikation und Fingerart. Sie aktualisiert die Fingervisualisierung in der entsprechenden Repräsentation gemäß der übergebenen Fingerart und hat keinen Rückgabewert.

Die Methode Pop hat die Parameter Datenobjekt, Visualisierungsknoten und Identifikation. Wie in Abschnitt 4.3.2 erläutert wurde, hat sie keine Aufgabe, die sie nach außen hin erfüllen muß und gibt daher auch keinen Wert zurück. Sie wird nur aufgerufen, um ggf.

interne Aufgaben einer Visualisierungsfunktion zu bearbeiten und wird daher im Folgenden nur erwähnt, falls ihre Implementierung tatsächlich nicht leer ist.

Für alle datenbezogenen Visualisierungsfunktionen werden von dataVF Default-Bindungen definiert. Sie sind in Tabelle 5.4 dargestellt. Die Spalte namens „IH“ bezeichnet das Vererbungs-Flag, das in Abschnitt 3.8.1 durch das Attribut Inherit modelliert wird. Dieses gibt an, ob die entsprechende Bindung an Sub-Visualisierungsfunktionen vererbt wird.

Klasse	Ereignis	IH	Operation
dataVF	<Enter>	Nein	F Push
	<Escape>	Nein	F Pop

Tabelle 5.4: Default-Bindungen der Klasse dataVF

5.3 Atomare Visualisierungsfunktionen

Zu den atomaren Visualisierungsfunktionen zählen alle Visualisierungsfunktionen, die keine durch Nachfolgerknoten definierten Repräsentationen von Sub-Objekten in ihre Repräsentation integrieren. Diese Klasse zerfällt wiederum in

- die generischen Visualisierungsfunktionen für atomare Typen und
- die speziellen Visualisierungsfunktionen, insbesondere für Objekttypen.

Zusätzlich könnten noch „spezielle Visualisierungsfunktionen für atomare Typen“ aufgezählt werden, mit denen dann die generischen atomaren Visualisierungsfunktionen gemeint wären, die nicht für alle atomaren Typen definiert sind. Diese werden hier jedoch auch als „generisch“ betrachtet.

Das Interface der Visualisierungsklasse atomVF erweitert das von visFunc um die Methode Edit. Eine Übersicht ist in Tabelle 5.5 dargestellt.

Klasse	Methoden
visFunc	Width, Height
	Pre, Post
	Finger
dataVF	Pop
atomVF	Edit

Tabelle 5.5: Interface der Visualisierungsklasse atomVF

Die Pre-Methoden von atomaren Visualisierungsfunktionen verändern Viewport und Offset nicht. Die Methode Edit realisiert das Ändern eines atomaren Wertes. Dabei können

Tastatur- und Mausfokus von der atomaren Repräsentation übernommen werden. Das Standardverhalten des Visualisierungsverfahrens, das dadurch temporär ersetzt wird, wird nach der Ausführung der Edit-Methode von der Fingermethode Push wieder hergestellt (siehe Abschnitt 4.3.1). Edit bekommt Datenobjekt, Visualisierungsknoten und Identifikation als Parameter und gibt eine Größenänderung der Repräsentation als Resize-Delta zurück.

Neben den Bindungen der Klasse visFunc werden von atomVF keine weiteren Default-Bindungen definiert. Atomare Typen (Integer, String, usw.) werden per Default von der Visualisierungsfunktion atom dargestellt (vgl. Abschnitt 5.3.1).

Als Beispiele generischer Visualisierungsfunktionen für atomare Typen werden die folgenden Visualisierungsfunktionen beschrieben:

- atom für beliebige atomare Typen und
- text für den Typ String.

Als Beispiele spezieller Visualisierungsfunktionen für Objekttypen werden die folgenden Visualisierungsfunktionen beschrieben:

- polygon und image für einen konstruierten Tupel-Typ,
- color für Zeichenketten, die Farben beschreiben und
- atomAttr für die Schemadarstellung von atomaren Attributen (siehe Abschnitt 3.7).

5.3.1 Die Visualisierungsfunktion atom

Die Visualisierungsfunktion atom ist die Default-Visualisierungsfunktion für atomare Typen. Jeder atomare Wert hat eine textuelle Repräsentation. Diese Eigenschaft wird von atom ausgenutzt, um einen atomaren Wert darzustellen: Seine textuelle Repräsentation wird in einem Rechteck mit fester Breite und Höhe dargestellt.

Parameter: Die für atom definierten Parameter und ihre Default-Werte sind in Tabelle 5.6 auf der nächsten Seite dargestellt. Ist die durch `font` angegebene Schriftart keine Proportionalschrift, sondern eine mit fester Breite (*fixed-width*, *monospaced*; z. B. die Default-Schriftart), können Breite und Höhe der Darstellung relativ zur Größe dieser Schrift, d. h. als Vielfaches der Zeichenbreite bzw. -höhe, angegeben werden, indem der numerische Wert des Parameters `width` bzw. `height` mit dem Buchstaben „`x`“ konkateniert wird. Die Parameter `enum` und `path` sind in der Tabelle etwas abgesetzt, da sie nur für bestimmte atomaren Typen anwendbar sind.

Tastaturereignisse: Von atom werden zusätzlich zu den Default-Bindungen der Klassen dataVF und atomVF für `<Enter>` und `<Escape>` keine weiteren Bindungen vorgenommen. Die temporären, lokalen Bindungen, die während des Editierens atomarer Werte gelten, werden von den durch `script` spezifizierten Skripten vorgenommen.

Name	Beschreibung	Default	Typ
`font`	Schriftart	`10x20`[a]	Font
`color`	Textfarbe	`black`	Color
`bgColor`	Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`width`	Breite der Darstellung[b]	`7x`	numerisch[c]
`height`	Höhe der Darstellung[d]	`1x`	numerisch[c]
`align`	Ausrichtung des Textes	`center`	Alignment
`script`	Basisname für Skriptsammlungen	`default`	String
`fmt`	Formatierung des Wertes	$\perp$	String
`enum`[e]	Name des Aufzählungstyps	$\perp$	String
`path`[f]	Attributpfad	$\perp$	String

Tabelle 5.6: Parameter der Visualisierungsfunktion atom

[a] Plattformabhängig, aber keine Proportionalschrift.
[b] Ggf. relativ zur Breite einer *monospaced* Schriftart.
[c] Ggf. konkateniert mit einem „`x`“.
[d] Ggf. relativ zur Höhe der Schriftart.
[e] Für Attribute vom Typ Integer, die Aufzählungstypen simulieren
[f] Für Attribute vom Typ Link

Methoden Width und Height: Durch atom generierte Repräsentation haben immer feste Breiten und Höhen, so daß beide Methoden immer definierte Werte liefern. Diese sind durch die Parameter `width`, `height` und ggf. `font` bestimmt. Die Berechnung der Darstellungsgröße relativ zu den Graden einer *monospaced* Schriftart kann in einer Tcl/DB-Implementierung wie folgt geschehen:

```
if [string match {*x} $width] {
    set fontWidth [font measure $font x]
    set width [expr $fontWidth*[string trimright $width "x"]]
}

if [string match {*x} $height] {
    set fontHeight [font metrics $font -linespace]
    set height [expr $fontHeight*[string trimright $height "x"]]
}
```

Methoden Pre und Post: Die feste Größe der Repräsentation ist stets das Ergebnis der Pre-Methode. Da das Lesen und Generieren eines atomaren Wertes durchaus mit Aufwand verbunden sein kann, wird das `get`-Skript und damit die eigentliche Darstellung

des atomaren Wertes erst in der Post-Methode ausgeführt. In der Pre-Methode muß daher, ähnlich wie bei komplexen Visualisierungsfunktionen, ein Behälter-Widget generiert werden. Für eine Tcl/DB-basierte Implementierung hat ein zusätzliches frame Widget [Ous94, S. 158 ff.] als Behälter für ein `label` Widget [Ous94, S. 161 ff.] weiterhin den Vorteil, daß es die Konfiguration von Höhe und Breite auf Basis beliebiger Längen erlaubt. Dieses ist also sogar notwendig, da in Tcl/Tk für `label` Widgets die Konfiguration von Höhe und Breite nur relativ zur Größe der Schriftart möglich ist.

Methode Finger: Finger sollen durch veränderte Hintergrund- bzw. Randfarben dargestellt werden. Die Fingerart wird als Index in eine globale Farbtabelle interpretiert (siehe Abschnitt 4.3.3). Der Eintrag für den Index 0 definiert den Zustand „kein Finger“, also die „normale“ Hintergrundfarbe.

Methode Edit, Skriptsammlungen Der Parameter `script` gibt den Basisnamen von Skripten an, die zum Lesen und Editieren des atomaren Wertes verwendet werden. Das `edit`-Skript wird von der gleichnamigen Methode aufgerufen und implementiert das Editieren eines atomaren Wertes. Die Methode Edit dient also nur als Hülle, damit das Grundgerüst der Visualisierungsfunktion atom möglichst universell verwendet werden kann. Die Edit-Methode liefert immer ein Resize-Delta von Null, da die Visualisierungsfunktion atom feste Breiten und Höhen hat. Das `get`-Skript wird von der Methode Post verwendet, um die textuelle Repräsentation des atomaren Wertes zu generieren. Vordefiniert sind drei Skriptsammlungen, deren Basisnamen `default`, `link` und `enum` sind. Diese Basisnamen werden von atom mit den Suffixen `get` und `edit` versehen.

- `default_get` liefert den atomaren Wert als String. In einer Tcl/DB-Implementierung wird mit dem Finger F, der auf dem atomaren Wert positioniert ist, das Tcl/DB-Kommando `get` ausgeführt. Falls der Parameter `fmt` definiert ist, wird dessen Ergebnis mit dem Tcl-Kommando `format` entsprechend `fmt` formatiert:

```
set val [$f get]
if [info exists fmt] {
    set val [format $fmt $val]
}
```

- `default_edit` wandelt das Rechteck temporär in einen Eingabebereich um, der dann editiert werden kann. Der Tastaturfokus wird auf diesen Eingabebereich gesetzt. Für ihn werden an die Tastaturereignis `<Enter>` und `<Escape>` spezielle Skriptaufrufe gebunden, die das Beenden bzw. den Abbruch der Eingabe behandeln. Zusätzlich wird das Tastaturereignis `<Control-0>` so gebunden, daß der atomare Wert durch einen Nullwert ersetzt wird. Das folgende Beispiel zeigt den Ausschnitt einer Repräsentation beim Browsen und während des Editierens:

(a) Beim Browsen (b) Beim Editieren

Abbildung 5.1: Editieren atomarer Werte mit `default_edit`

Beispiel 5.3 Editieren mit `default_edit`

Abb. 5.1 zeigt den Ausschnitt einer Repräsentation in zwei Situationen. In Abb. 5.1(a) zeigt ein Finger auf den Wert „lukas@cs“, in Abb. 5.1(b) ist er während des Editierens mit `default_edit` dargestellt. □

- `enum_get` liefert das Label, das dem Integer-Wert entspricht.
- `enum_edit` generiert ein Pop-Up-Menü, das die Selektion eines zulässigen Labels für den durch den Parameter `enum` spezifizierten Aufzählungstyp erlaubt. Das Pop-Up-Menü bindet die Aktivität des Fenstersystems während seiner Sichtbarkeit vollständig und installiert „eigene“ Bindungen für Eingabeereignisse. Das Skript `enum_edit` wird erst beendet, wenn das Pop-Up-Menü verschwindet. Falls durch Menüselektion eine Änderung vorgenommen wurde, wird die Änderung des atomaren Wertes sofort ausgeführt. Das folgende Beispiel zeigt den Ausschnitt einer Repräsentation beim Browsen und während des Editierens:

(a) Beim Browsen (b) Beim Editieren

Abbildung 5.2: Editieren atomarer Werte mit `enum_edit`

Beispiel 5.4 Editieren mit `enum_edit`
Abb. 5.2 zeigt den Ausschnitt einer Repräsentation, die mehrere Aufzählungstypen (Ja/Nein, Geschlecht, Orientierung) enthält, in zwei Situationen. Der Finger zeigt in Abb. 5.2(a) auf das Attribut Sex (dt. Geschlecht), das den gleichnamigen Aufzählungstyp und momentan den Wert „male“ (dt. männlich) hat. In Abb. 5.2(b) ist das Pop-Up-Menü dargestellt, das während des Editierens mit `enum_edit` () erscheint und die zulässigen Werte des Aufzählungstyps enthält: „unknown“, „male“ und „female“. □

- `link_get` liefert die textuelle Repräsentation des atomaren Wertes, auf den der Link, ggf. in Verbindung mit dem Parameter `path`, verweist. In einer Tcl/DB-Implementierung wird dazu die Option `link` des *fid*-Kommandos benutzt (siehe Abschnitt 2.3.4, insbesondere Tabelle 2.2 auf Seite 56):

```
if [info exists path] {
    set lf [$f link -newfinger]
    set val [$lf get $path]
    finger free $lf
} else {
    set val [$f link -atomic]
}
```

- `link_edit` leitet einen modalen Dialog ein, der die Selektion eines geeigneten Link-Ziels erlaubt. Der Tastaturfokus wird lokal übernommen bzw. an den modalen Dialog abgegeben. Die „Eignung“ von Link-Zielen ist von der Typdefinition des Link-Attributes abhängig. Der modale Dialog kann die Komponenten Tabellenauswahl und Tabellennavigation enthalten.

Meta-Daten: Abb. 5.3 zeigt den Eintrag für atom in der Visualisierungsfunktionen-Tabelle.

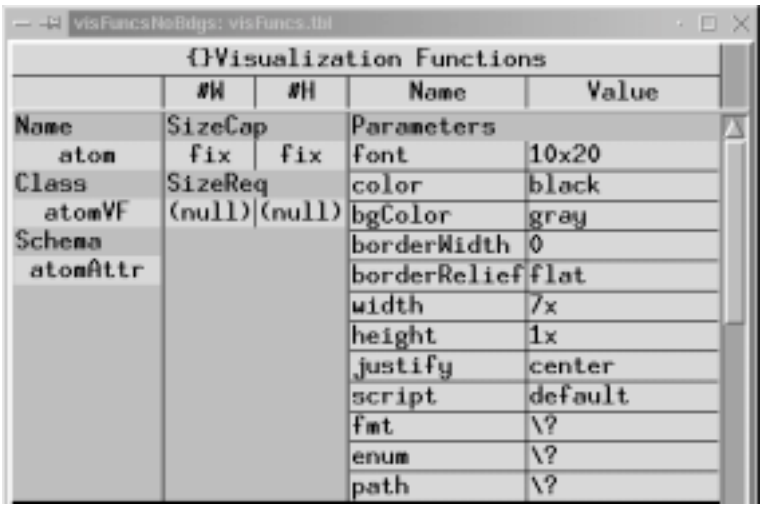


Abbildung 5.3: Eintrag für atom in der Visualisierungsfunktionen-Tabelle

5.3.2 Die Visualisierungsfunktion text

Die Darstellung von langen Zeichenketten (Strings) mit der Visualisierungsfunktion atom ist ungenügend, da die Breite der Repräsentation fest ist. Strings können in ihrer Länge

erheblich variieren; wird die Breite der Repräsentation zu klein gewählt, kann der String nicht vollständig dargestellt werden, wird sie zu groß gewählt, bleibt viel Leerraum in der Repräsentation. Die Visualisierungsfunktion text stellt Strings als Text mit einer festen Breite dar. Die Länge der Strings bestimmt die Höhe der Textdarstellung, die Höhe ist also variabel.

Parameter und Tastaturereignisse: Für text können die Parameter von atom (vgl. Abschnitt 5.3.1), die auch für text sinnvoll sind, mit der gleichen Bedeutung angegeben werden: `font`, `color`, `bgColor`, `borderWidth`, `borderRelief`, `width`, (`height` entfällt,) `align` und `script`. Zusätzlich kann mit dem Parameter `wrap` der Zeilenumbruch gesteuert werden. Für ihn sind die Werte `word` (wortweiser Umbruch, Default) und `char` (zeichenweiser Umbruch) zulässig. Die für text definierten Parameter und ihre Default-Werte sind in Tabelle 5.6 auf Seite 182 dargestellt. Bzgl. Tastaturereignissen gelten die gleichen Aussagen wie bei atom.

Name	Beschreibung	Default	Typ
`font`	Schriftart	`10x20`[a]	Font
`color`	Textfarbe	`black`	Color
`bgColor`	Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`width`	Breite der Darstellung[b]	`7x`	numerisch[c]
`align`	Ausrichtung des Textes	`center`	Alignment
`script`	Basisname für Skriptsammlungen	`default`	String
`fmt`	Formatierung des Wertes	$\perp$	String
`wrap`	Zeilenumbruch	`word`	Wrap

Tabelle 5.7: Parameter der Visualisierungsfunktion text

[a] Plattformabhängig, aber keine Proportionalschrift.
[b] Ggf. relativ zur Breite einer *monospaced* Schriftart.
[c] Ggf. konkateniert mit einem „`x`“.

Methoden Width und Height: Die Breite der Visualisierungsfunktion ist fest und wird genau wie bei atom berechnet. Die Höhe ist variabel, die Height-Methode gibt also Null zurück. Die Höhe einer konkreten Repräsentation kann erst in der Pre- bzw. Post-Methode berechnet werden.

Methoden Pre und Post: Die berechenbare Größe der Repräsentation ist stets das Ergebnis der Pre-Methode. Wie bei atom wird in der Pre-Methode ein Behälter generiert. Der Textwert muß jedoch gelesen werden, um die Höhe der Repräsentation berechnen zu können. In einer Tcl/DB-Implementierung kann die Höhe wie folgt berechnet werden:

```
set fontHeight [font metrics $font -linespace]
set text [${script}_get $f]
set lines [llength [split $text "\n"]]
set height [expr $fontHeight*$lines]
```

Methode Edit, Skriptsammlungen: Der Parameter `script` gibt wie oben bei atom den Basisnamen von Skripten an, die zum Lesen und Editieren verwendet werden. Das Resize-Delta wird aus der Zeilen-Differenz zwischen alter und neuer Textrepräsentation berechnet. Vordefiniert sind zwei Skriptsammlungen, deren Basisnamen `default` und `extern` sind. `default` realisiert eine einfache „in-place“ Texteingabe, `extern` verwendet ein externes Tool als modalen Dialog zur Texteingabe.

- `default_get` liefert den Textwert als String, der gemäß `wrap` in Zeilen umgebrochen ist.
- `default_edit` wandelt das Rechteck temporär in einen Eingabebereich der gleichen Größe um, der editiert werden kann und bei Bedarf mit einem vertikalen Rollbalken versehen wird. Da das Tastaturereignis `<Enter>` zu den Bindings für das Editieren gehört, wird an `<Control-Enter>` der Skriptaufruf für das Beenden der Eingabe gebunden.
- `extern_edit` ruft ein externes Tool als modalen Dialog auf, mit dem der Wert editiert werden kann. Der Tastaturfokus wird lokal übernommen bzw. an das externe Tool abgegeben. Das Skript `extern_edit` wird erst beendet, wenn das externe Tool beendet wird.

Methode Finger: Fingervisualisierungen werden analog atom dargestellt.

Meta-Daten: Abb. 5.4 zeigt den Eintrag für text in der Visualisierungsfunktionen-Tabelle.

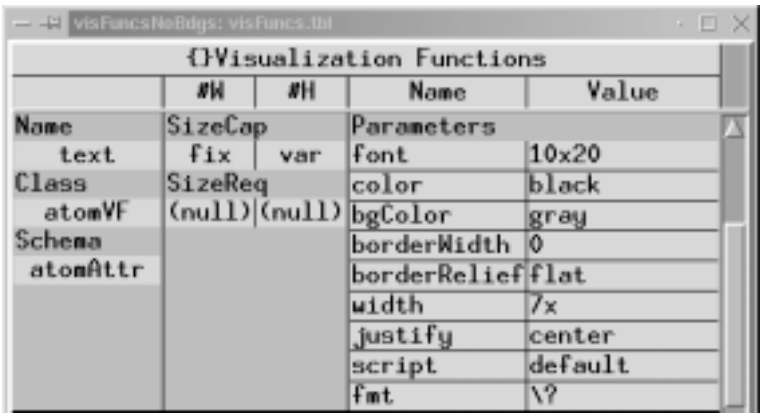
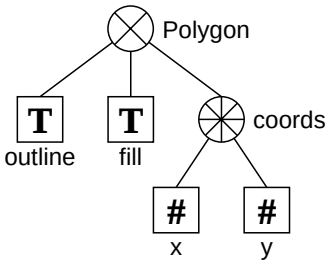
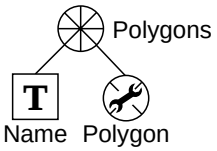


Abbildung 5.4: Eintrag für text in der Visualisierungsfunktionen-Tabelle

5.3.3 Die Visualisierungsfunktion polygon



(a) Objekttyp Polygon



(b) Schema mit Polygon

Abbildung 5.5: Schemabäume für Polygone

Der Objekttyp Polygon (siehe Abb. 5.5(a)) definiert Tupel, die als Komponenten die Rand- und Füllfarbe des Polygons (outline bzw. fill) sowie eine Liste der Eckpunktkoordinaten enthalten (coords).

Die Polygon-Tupel können mit der speziellen Visualisierungsfunktion polygon dargestellt werden, die die Koordinatenliste als Punkte eines geschlossenen Polygonzuges interpretiert und diesen gemäß den Farbangaben graphisch darstellt. Sie bezieht sich auf die oben beschriebene Struktur der Polygon-Tupel nur intern, inegriert also keine durch Nachfolgerknoten der Visualisierung definierten Repräsentationen von Sub-Objekten. Mit und ohne polygon generierte Repräsentationen von Polygon-Tupeln sind in Abb. 5.6 auf der gegenüberliegenden Seite dargestellt.

Methoden Pre und Post, Width und Height, Finger, Tastaturereignisse:
Die Polygone werden graphisch in einem Rechteck fester Breite und Höhe dargestellt. Die Methoden Width und Height geben also stets definierte Größen zurück. Die Leinwand wird in der Pre-Methode erzeugt, ihre Größe berechnet. Das Generieren des Polygons wird erst in der Post-Methode ausgeführt. Zusätzlich zu den Bindungen für `<Enter>` und `<Escape>` werden keine weiteren Bindungen vorgenommen. Während des Editierens eines Polygons werden von der Methode Edit temporäre, lokale Bindungen installiert. Ist ein Polygon in einem Nimbus, wird dies durch die entsprechende Färbung des Hintergrundes dargestellt.

Parameter: Die für polygon definierten Parameter und ihre Default-Werte sind in Tabelle 5.8 auf Seite 190 dargestellt. `color` und `fillColor` definieren die Werte, die für Rand- und Füllfarbe gelten, falls die Komponenten `outline` bzw. `fill` des Polygon-Tupels Null sind.

Generierung des Polygons: Das Polygon wird erst in der Post-Methode auf die Leinwand gemalt. Dazu wird die Koordinaten-Liste durchlaufen, in einer Tcl/DB-Implementierung z. B. wie folgt:

{}Polygons

Name	[]Polygon		<>coords	
	fill	outline	#x	#y
triangle	yellow	blue	50	0
			90	90
			0	90
square	white	red	0	0
			90	0
			90	90
			0	90
star	yellow	red	45	0
			70	90
			0	30
			90	30
			20	90
hexagon	white	blue	20	0
			70	0
			90	45
			70	90
			20	90
			0	45
			30	0

(a) ohne Verwendung von polygon)

(b) mit Verwendung von polygon

Abbildung 5.6: Repräsentationen des Objekttyps Polygon

Beispiel 5.5 Repräsentationen des Objekttyps Polygon
Die Abbildung zeigt zwei Repräsentationen einer Tabelle, die durch das Schema Polygons aus Abb. 5.5(b) auf der gegenüberliegenden Seite strukturiert ist. In Abb. 5.6(a) ist eine durch die Default-Visualisierung ohne Verwendung der speziellen Visualisierungs-Funktion polygon definierte Repräsentation dargestellt, in Abb. 5.6(b) wurde polygon eingesetzt. In beiden Abbildungen zeigt ein Finger Ⓐ auf das sternförmige Polygon. In Abb. 5.6(b) zeigt der Finger Ⓑ auf das Hexagon, das gerade (graphisch) editiert wird. □

Name	Beschreibung	Default	Typ
`color`	Linienfarbe	`black`	Color
`lineWidth`	Dicke der Linie	`2`	numerisch
`bgColor`	Hintergrundfarbe	`gray`	Color
`fillColor`	Füllfarbe	`white`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`width`	Breite der Darstellung	`100`	numerisch
`height`	Höhe der Darstellung	`100`	numerisch

Tabelle 5.8: Parameter der Visualisierungsfunktion polygon

```
set cmd {create polygon}
$f push coords
iterate $f {
    lappend cmd [$f get x] [$f get y]
}
$f pop
eval $w $cmd
```

Da die Größe der Repräsentation fest ist, wird die Polygondarstellung skaliert und auf der Leinwand zentriert. Die Farben des Polygons werden gemäß den Attributen fill und outline dargestellt, die Leinwand (d. h. der Polygonhintergrund) entsprechend dem Parameter `bgColor`. In der Tcl/DB-Implementierung wird das durch Zufügen weiterer Optionen vor dem Ausführen des `eval`-Kommandos erreicht, z. B. für die Füllfarbe:

```
if [$f isnull "fill"] {
    set fill $fillColor
} else {
    set fill [$f get "fill"]
}
# use 'winfo rgb' command to check, if $fill is usable
if ![catch {winfo rgb $w $fill}] {
    lappend cmd -fill $fill
}
```

Methode Edit: Die Methode Edit implementiert das Editieren eines Polygons. Edit installiert lokale Bindungen, die z. B. *drag* & *drop* der Polygonecken mit der Maus erlauben. An die Tastaturereignisse `<Enter>` und `<Escape>` werden Skriptaufrufe gebunden, die das Beenden bzw. den Abbruch der Eingabe behandeln. Ein Eckpunkt des Polygons wird als aktiv markiert, was der Positionierung eines Fingers auf einem

Element der Koordinaten-Liste entspricht. Relativ zum aktiven Eckpunkt werden die Tastaturereignisse `<Left>`, `<Right>`, `<Up>` und `<Down>`, für das Verschieben des Eckpunktes, `<Insert>` und `<Delete>`, für das Einfügen und Löschen von Eckpunkten, sowie `<Tab>` und `<Shift-Tab>` für die Auswahl des aktiven Eckpunktes, interpretiert. In Abb. 5.6(b) wird gerade das Polygon namens „hexagon“ editiert.

Meta-Daten: Abb. 5.7 zeigt den Eintrag für polygon in der Visualisierungsfunktionen-Tabelle.

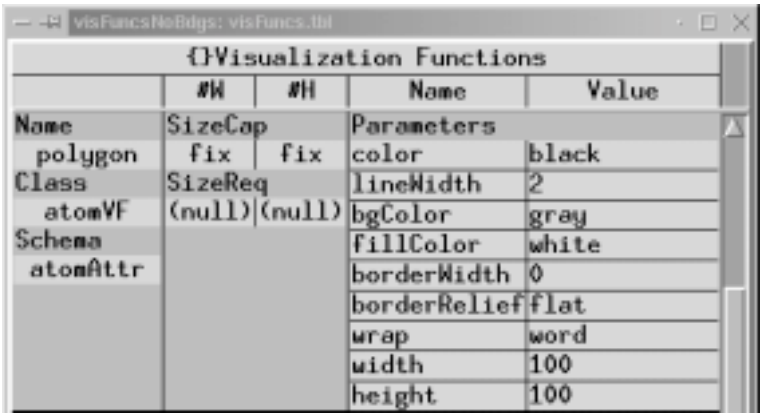


Abbildung 5.7: Eintrag für polygon in der Visualisierungsfunktionen-Tabelle

Diese Realisierung der Darstellung von Polygonen dient haupsächlich der Demonstration einer atomaren, speziellen Visualisierungsfunktion, die für einen komplex strukturierten Objekttyp definiert ist. Eine andere Möglichkeit zur Darstellung von graphischen Objekten, die durch entsprechend strukturierte Datenobjekte definiert sind, wäre die Definition einer Konstruktions-Visualisierungsfunktion, die eine Drawing-Area (Zeichenfläche, *canvas*) implementiert und deren Sub-Visualisierungsfunktionen graphische Objekte mit entsprechenden Identifikationen (siehe Abschnitt 4.1.3) generieren, die dann in der Drawing-Area dargestellt werden. Auf diese Weise könnten die einzelnen graphischen Objekte verschiedenen Typs sein (z. B. auch Kreise und Ellipsen, die durch Polygone nur schlecht angenähert werden können) und in das Interaktionskonzept integriert sein.

5.3.4 Die Visualisierungsfunktion color

Der Typ Color sei ein String, der als Wert textuelle Definitionen von Farben enthalten kann. Die Color-Werte können mit der speziellen Visualisierungsfunktion color dargestellt werden (vgl. Abb. 5.8 auf Seite 193). Sie ist für einen atomaren Typ definiert, kann sich also keine Repräsentationen von Sub-Objekten integrieren. Die Farbwerte werden in einem Rechteck

fester Breite und Höhe dargestellt, wobei die textuelle Repräsentation des Wertes auf einem dem Farbwert entsprechend gefärbten Hintergrund umgeben von einem Rand dargestellt wird. Die Textfarbe wird so gewählt, daß der Text auf dem Hintergrund sichtbar bleibt. Die von color generierten Repräsentationen könnten z. B. für die Farbauswahl in einem elektronischen Warenkatalog eingesetzt werden.

Parameter: Für color können die Parameter von atom (vgl. Abschnitt 5.3.1), die auch für color sinnvoll sind, mit der gleichen Bedeutung angegeben werden: `font`, (`color` entfällt,) `bgColor`, `borderWidth`, `borderRelief`, `width`, `height`, `align` und `script`. Bzgl. Tastaturereignissen gelten die gleichen Aussagen wie bei atom. Der Default-Wert von `borderWidth` ist hier 5, da die Sichtbarkeit des Randes wichtig für die Darstellung des Fingers ist. Hat der Parameter `script` den Wert `enum`, wird auch der Parameter `enum` interpretiert. Dieser hat hier als Default-Wert `color`, den Namen eines Aufzählungstyps, dessen Label Farbnamen sind.

Methode Finger: Ist ein Color-Objekt in einem Nimbus, wird dies durch die entsprechende Färbung des Randes und durch die Invertierung der Schrift dargestellt. Ist die Randbreite Null, kann nicht mehr zwischen verschiedenen Fingern unterschieden werden, da keine verschiedenen Rand-Farben mehr sichtbar sind.

Methode Edit: Mit dem Parameter `script` kann analog zu atom der Editier-Mechanismus von color eingestellt werden. Auch hier sind drei Skriptsammlungen `default`, `extern` und `enum` vordefiniert. Mit den `default`-Skripten kann der Farbwert, wie mit den `default`-Skripten von atom, textuell eingegeben werden. Die `extern`-Skripte ermöglichen die Auswahl eines Farbwertes durch ein externes Tool als modalen Dialog, wie mit den `extern`-Skripten von text. In einer Tcl/DB-Implementierung wäre z. B. die Einbindung des Farbauswahl-Dialoges `colordial` denkbar [HM98, S. 128 ff.]. Die `enum`-Skripte realisieren die Farbauswahl über einen Aufzählungstyp wie die `enum`-Skripte von atom.

Meta-Daten: Abb. 5.8 auf der gegenüberliegenden Seite zeigt den Eintrag für color in der Visualisierungsfunktionen-Tabelle.

5.3.5 Die Visualisierungsfunktion image

Der Objekttyp Image ist gemäß Abb. 2.7 auf Seite 59 definiert. Ein Image-Tupel besteht aus dem Namen, dem Dateinamen, der Kennzeichnung des Graphikformates, der Größe und den eigentlichen, binären Bilddaten[3] (Komponenten Name, File, Format, Size und Data). Die Visualisierungsfunktion image stellt ein Image-Tupel dar, indem die Bilddaten entsprechend des Graphikformates interpretiert und als Rasterbild dargestellt werden. In Abb. 5.9 auf Seite 194 wird image an einem Beispiel erläutert.

[3]Die binären Bilddaten sind in der Datenbank abgelegt, der Dateiname ist rein informativ die Bezeichnung der Datei, aus der diese Daten importiert wurden.

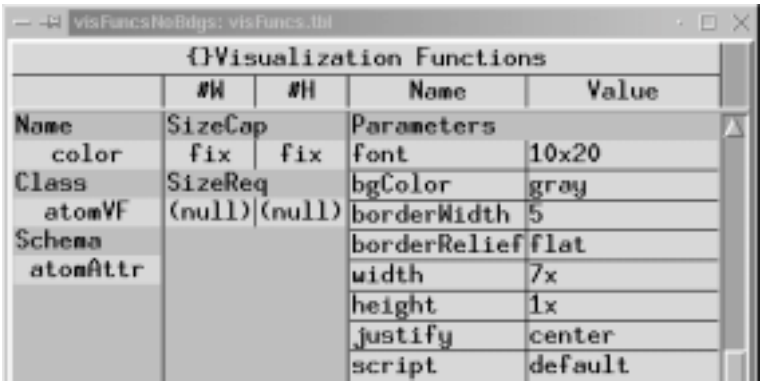


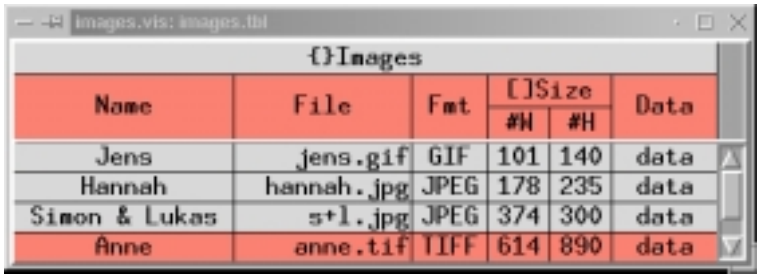
Abbildung 5.8: Eintrag für color in der Visualisierungsfunktionen-Tabelle

Parameter: Die für image definierten Parameter und ihre Default-Werte sind in Tabelle 5.9 dargestellt. Werden Breite `width` oder Höhe `height` als Nullwert angegeben, wird die entsprechende Länge des Bildes durch Skalierung bestimmt und ist daher variabel. Werden Breite und Höhe als Nullwert angegeben, wird das Bild in der im Image-Tupel gegebenen Größe dargestellt und beide Längen sind variabel. Sind Breite und Höhe nicht Null, wird mit dem Parameter `restrainRatio` bestimmt, ob das ursprüngliche Verhältnis der Seiten-Längen beibehalten und ggf. durch einen Rand ausgeglichen werden soll, oder ob das Bild ggf. verzerrt dargestellt werden kann. Mit dem Parameter `script` kann, analog zu atom, der Editier-Mechanismus von image eingestellt werden.

Name	Beschreibung	Default	Typ
`bgColor`	Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`5`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`width`	Breite der Darstellung	`100`	numerisch
`height`	Höhe der Darstellung	`100`	numerisch
`restrainRation`	Verhältnis der Seiten-Längen	`yes`	Boolean
`script`	Basisname für Skriptsammlungen	`default`	String

Tabelle 5.9: Parameter der Visualisierungsfunktion image

Methoden Width und Height: Breite und Höhe hängen von den Parametern `width` und `height` ab. Die Bestimmung einer variablen Länge ist relativ einfach und kann daher bereits in der Pre-Methode vorgenommen werden. Das Generieren der graphischen Information ist relativ aufwendig und wird daher erst in der Post-Methode ausgeführt. Ähnlich wie bei color wird ein Finger durch die Färbung des Randes dargestellt, weswegen dieser die Default-Breite 5 hat.



(a) Repräsentation ohne Verwendung von image

(b) Repräsentation mit Verwendung von image

Abbildung 5.9: Verschiedene Repräsentationen des Objekttyps Image

Beispiel 5.6 Repräsentation von Bildern

Die Abbildung zeigt zwei Repräsentationen einer Tabelle, die eine Menge von Image-Tupeln enthält. In Abb. 5.9(a) ist eine Repräsentation dargestellt, deren Visualisierung die spezielle Visualisierungs-Funktion image nicht verwendet. Der Bytestring Data wird dort von der generischen atomaren Visualisierungs-Funktion text durch den String „(data)“ dargestellt. Für die Repräsentation in Abb. 5.9(b) wurde image eingesetzt, um Data gemäß Format als Graphik zu interpretieren und als Rasterbild darzustellen. Die anderen Komponenten von Image werden von image nicht dargestellt. In beiden Abbildungen zeigt ein Finger auf dasselbe Tupel für „Anne“. □

Beispiel 5.7 Parametrisierung von image

In Abb. 5.9(b) auf der gegenüberliegenden Seite ist image so parametrisiert, daß die Bilder mit fester Höhe und variabler Breite dargestellt werden. Die Bilder sind skaliert, so daß sie, bei horizontaler Konkatenation der Menge, eine homogene Repräsentation erzeugen. Durch den umgebenden, horizontal rollbaren Viewport werden insgesamt feste Längen der Repräsentation erreicht. □

Methode Finger: Ist ein Image-Objekt in einem Nimbus, wird dies durch die entsprechende Färbung des Randes dargestellt. Ist die Randbreite Null, kann der Finger nicht mehr gesehen werden. Eine andere Möglichkeit wäre die Manipulation des Rasterbildes, z. B. durch Veränderung der Helligkeit oder eine Markierung mit einem graphischen Symbol (z. B. einer „zeigenden Hand").

Methode Edit: Der Editier-Mechanismus von image kann mit dem Parameter `script` eingestellt werden. Zwei Skriptsammlungen sind vordefiniert: `default` und `extern`. Mit den `default`-Skripten kann ein Bild mit einem File Browser ausgewählt werden. Bildgröße und -name werden gemäß dem Graphikformat aus der Bilddatei bestimmt. Die `extern`-Skripte ermöglichen die Auswahl eines Bildes durch ein externes Tool als modalen Dialog, wie mit den `extern`-Skripten von text.

Meta-Daten: Abb. 5.10 zeigt den Eintrag für image in der Visualisierungsfunktionen-Tabelle.

visFuncsNoBdgs: visFuncs.tbl

{}Visualization Functions

	#W	#H	Name	Value
Name	SizeCap		Parameters	
image	dep	dep	bgColor	gray
Class	SizeReq		borderWidth	5
atomVF	(null)	(null)	borderRelief	flat
Schema			width	100
atomAttr			height	100
			restrainRati	yes
			script	default

Abbildung 5.10: Eintrag für image in der Visualisierungsfunktionen-Tabelle

5.3.6 Die Visualisierungsfunktion atomAttr

Die Visualisierungsfunktion atomAttr ist für die Schemavisualisierung von Attribute-Tupeln, die keine Sub-Attribute haben, definiert. Sie stellt ein Attribute-Tupel als Rechteck fester Breite und Höhe, in dem die Komponente Name dargestellt ist, dar. Parametrisierung und Realisierung sind ganz analog zu atom für den Typ String, nur daß keine Skripte für das

Editieren vorhanden sein müssen. In einer Tcl/DB-Implementierung ist der Unterschied, daß die `get`-Operation zum Lesen des Wertes mit dem Pfad `Name` ausgeführt wird[4]. Da die Unterschiede zur Realisierung von atom sehr gering sind, wird auf atomAttr hier nicht weiter eingegangen.

5.4 Komplexe Visualisierungsfunktionen

Zu den komplexen Visualisierungsfunktionen zählen die Konstruktions-, die Kollektions- und die Tupel-Visualisierungsfunktionen. Die Funktionen der drei Klassen integrieren in unterschiedlicher Art und Weise eine oder mehrere durch Nachfolgerknoten definierte Repräsentationen. Die Methode Sub wird daher von cplxVF nur als virtuelle Methode in das Interface aufgenommen und von den Visualisierungsklassen für ihre Zwecke redefiniert. Allen gemeinsam ist die Erweiterung des Interfaces um die Methoden Resize, Position und Adjust. In Tabelle 5.10 ist das Interface der virtuellen Klasse cplxVF dargestellt.

Klasse	Methoden
visFunc	Width, Height
	Pre, Post
	Finger
cplxVF	Resize
	Position
	Adjust
	Sub

Tabelle 5.10: Interface der virtuellen Klasse cplxVF

Die Methode Resize wird, wie in Abschnitt 4.3.5 beschrieben, mit den Parametern Datenobjekt, Visualisierungsknoten und Resize-Delta aufgerufen. Eine Adjust-Methode wird nur implementiert, wenn die Visualisierungsfunktion einen adjustierbaren Viewport realisiert. Sie bekommt, wie in Abschnitt 4.3.4 beschrieben, Datenobjekt, Visualisierungsknoten, Identifikationen, Visualisierungsgrößen und -lage sowie Fingergröße und -Offset als Parameter übergeben.

Die Klasse cplxVF definiert keine Bindungen für Visualisierungsfunktionen, da sie auch die nicht datenbezogenen Konstruktions-Visualisierungsfunktionen enthält.

[4]Verwendet man den Parameter `path` von atom, kann man u. U. sogar atom benutzen, statt eine neue Visualisierungsfunktion atomAttr zu definieren.

5.5 Konstruktions-Visualisierungsfunktionen

Konstruktions-Visualisierungsfunktionen integrieren genau eine durch den Nachfolgerknoten definierte Repräsentation. Sie sind nicht datenbezogen, d. h. unabhängig von einem Datenobjekt. Letztendlich ist aber eine direkter oder indirekter Sub-Visualisierungsfunktionen datenbezogen, so daß auch jede durch eine Konstruktions-Visualisierungsfunktion generierte Repräsentation durch ein Datenobjekt–Visualisierungsknoten–Paar identifiziert werden kann. In einer Visualisierung hat ein Visualisierungsknoten mit einer Visualisierungsfunktion der Klasse constrVF genau einen Nachfolger.

Das Interface von cplxVF wird von constrVF nicht erweitert; lediglich die Methode Sub wird redefiniert; ihr werden das Datenobjekt, Visualisierungsknoten und Identifikationen, Viewport und Offset, sowie Eigenschaften übergeben. Da Konstruktions-Visualisierungsfunktionen genau eine Repräsentation integrieren, enthält das Interface von constrVF keine Methode Sub_Finger. Eine Adjust-Methode wird z. B. von den im Folgenden beschriebenen Konstruktions-Visualisierungsfunktionen scrollV, scrollH und scroll implementiert.

Die Klasse constrVF definiert keine Bindungen, da Konstruktions-Visualisierungsfunktionen als nicht datenbezogene Visualisierungsfunktionen den Tastaturfokus nicht erhalten.

5.5.1 Die Visualisierungsfunktion scrollV

Die Konstruktions-Visualisierungsfunktion scrollV realisiert vertikal rollbare Viewports. Sie definiert dazu eine feste, parametrisierbare Höhe, die von der Höhe der Sub-Sub-Visualisierungsfunktion abhängen kann (diese muß dann ihrerseits eine feste Höhe haben). Der Bezug ist hier eine Sub-Sub-Visualisierungsfunktion, da die häufigste Verwendung von scrollV die Darstellung eines Kollektionsausschnitts ist. Soll dieser Ausschnitt eine feste Anzahl von Kollektionselementen enthalten, muß die Höhe des Viewports relativ zur Höhe der Elemente angegeben werden, die folglich fest sein muß. Die Element-Visualisierungsfunktion ist in diesem Fall die Sub-Sub-Visualisierungsfunktion der Konstruktions-Visualisierungsfunktion. Mit einem Rollbalken kann der sichtbare Ausschnitt des Viewports manipuliert werden.

Parameter: Die für scrollV definierten Parameter und ihre Default-Werte sind in Tabelle 5.11 auf der nächsten Seite dargestellt. Durch `height` kann die Höhe der Darstellung als Vielfaches der festen Höhe der Sub-Sub-Visualisierungsfunktion angegeben werden, wenn der numerische Wert mit einem „s“ konkateniert wird.

Methoden Width und Height, Längeneigenschaften und -anforderungen: Die Breite der Visualisierungsfunktion wird aus der Breite der Sub-Visualisierungsfunktion, die fest sein muß, der Randdicke und der Breite des Rollbalkens berechnet, ist also selbst fest. An die Sub-Visualisierungsfunktion werden keine Anforderungen

Name	Beschreibung	Default	Typ
`bgColor`	Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`barWidth`	Dicke des Rollbalkens	`15`	numerisch
`height`	Höhe der Darstellung[a]	`200`[b]	numerisch[c]
`side`	Seite des Rollbalkens	`right`	Horizontal

Tabelle 5.11: Parameter der Visualisierungsfunktion scrollV

[a] Ggf. relativ zur Höhe der Sub-Sub-Visualisierungsfunktion.
[b] `1s`, falls relativ zur Höhe der Sub-Sub-Visualisierungsfunktion.
[c] Ggf. konkateniert mit einem „`s`“.

bzgl. der Höhe gestellt. Die Höhe der Visualisierungsfunktion ist fest. Sie ist entweder absolut, oder, falls der Parameter `height` mit einem „`s`“ endet, relativ zur Höhe der Sub-Sub-Visualisierungsfunktion.

Methoden Pre, Sub und Post: Die Methode Pre gibt immer die von Width und Height berechnete Größe zurück. Sie erzeugt den Behälter, der für die Aufnahme einer Repräsentation als adjustierbarer Viewport ausgelegt ist, und den Rollbalken. Die Länge des Balkens im Rollbereich kann erst festgelegt werden, wenn die Größe der zu integrierenden Repräsentation bekannt ist, also bei der Ausführung der Methoden Sub bzw. Post[5]. Der Offset wird von Pre auf den Ursprung initialisiert und die übergebene Größe des sichtbaren Bereichs (Parameter Viewport) auf den tatsächlich sichtbaren Bereich (generierter Behälter) eingeschränkt. Bei der Berechnung der Größe des Viewports aus der von Width berechneten Breite und der von Height berechneten Höhe muß wieder die Randdicke und die Breite des Rollbalkens beachtet werden. Die Methode Sub sorgt lediglich dafür, daß die zu integrierende Repräsentation im Viewport-Behälter dargestellt wird. Zu beachten ist dabei, daß die Repräsentation außerhalb des Viewports unsichtbar bleibt (*clipping*). Dazu muß in einer Tcl/DB-Implementierung eine Schachtelung von drei frame Widgets ineinander erfolgen: Das äußerste, das gleichzeitig das Behälter-Widget ist, wird zur Darstellung des Randes benötigt, das mittlere definiert den sichtbaren Bereich und realisiert damit das Clipping, das innerste dient zur Aufnahme der Repräsentation, ist also der adjustierbare Bereich. In der Methode Post wird, gemäß den übergebenen Eigenschaften der integrierten Repräsentation, die Länge des Balkens im Rollbereich adjustiert.

Methode Position: Um die Position der integrierten Repräsentation im Viewport relativ zum sichtbaren Ausschnitt berechnen zu können, muß der Methode Position die Lage des sichtbaren Ausschnitts im Viewport bekannt sein. Der horizontale Offset Re-

[5] Die Aufteilung der Berechnung von Konstruktions-Visualisierungsfunktionen durch die Methoden Sub und Post ist eigentlich nur „kosmetisch“, da sie in der Post-Methode eines Visualisierungsknotens direkt nacheinander aufgerufen werden.

präsentation im Viewport ist immer Null. Der vertikale Offset ist immer kleiner oder gleich Null, da der sichtbare Ausschnitt, aus der wörtlich ursprünglichen Lage, logisch nach unten wandert, aber eigentlich die Repräsentation nach oben. Die Distanz, um die der sichtbare Ausschnitt verschoben ist, muß von Null abgezogen werden, da die Bewegung nach oben entgegen der vertikalen Achsenrichtung ist. In einer Tcl/DB-Implementierung kann die vertikale Position wie folgt berechnet werden:

```
# get place info for vertical scroll $sv
set pi [place info $sv]
# extract relative vertical placement
set rel_y [lindex $pi [expr [lsearch -exact $pi -relyx]+1]]
# calculate relative vertical position
set rel_v_pos [expr [winfo height $sv]*$rely]
```

Methode Resize: Da die Anforderung an die Sub-Visualisierungsfunktion eine feste Breite ist, kann der Methode Resize nur ein vertikales Resize-Delta übergeben werden ($\vec{\Delta}_r = (\delta_x, \delta_y)$ mit $\delta_x = 0$). Dieses wird voll kompensiert, da sich die Größe des Viewports nicht ändert. Die Resize-Methode eines Visualisierungsknotens, für den scrollV als Visualisierungsfunktion eingetragen ist, beendet also das Aufsteigen in der Visualisierung (siehe Abschnitt 4.3.5).

Die Methode Resize paßt die Länge des Balkens im Rollbereich gemäß δ_y an. Außerdem stellt sie sicher, daß im sichtbaren Bereich keine Inkonsistenzen durch „in der Luft hängende“ Repräsentationen auftreten, wie in Abb. 5.11 auf der nächsten Seite an einem Beispiel dargestellt wird.

Methode Adjust: Eine der inhärenten Aufgaben der Visualisierungsfunktion scrollV wird von der Methode Adjust implementiert: Sie gewährleistet die Sichtbarkeit eines Fingers im Viewport, indem sie ihn bei Bedarf adjustiert. Sie bekommt alle für die Berechnung notwendigen Größenangaben als Parameter übergeben: Größe der Repräsentation $\vec{s}$, Größe der integrierten Repräsentation $\vec{s}_{sub}$, relative Lage der integrierten Repräsentation im Viewport $\vec{p}$, Fingergröße $\vec{s}_f$ und Finger-Offset $\vec{co}$.

Der Viewport wird von der Methode Adjust nur adjustiert, wenn der Finger ganz oder teilweise verdeckt ist. Die Berechnung der neuen vertikalen relativen Lage $\vec{p}'_y$ ist dann wie folgt:

1. Wenn $\vec{s}_{f,y} > \vec{s}_y$ gilt, dann setze
   $\vec{p}'_y := -\vec{o}_{f,y}$.
2. Sonst wenn $\vec{o}_{f,y} + \vec{s}_{f,y} < -\vec{p}_y$ oder $\vec{o}_{f,y} > -\vec{p}_y + \vec{s}_y$ gilt, dann setze
   $\vec{p}'_y := -(\vec{o}_{f,y} + \vec{s}_{f,y}/2 + \vec{s}_y/2)$.
3. Sonst wenn $\vec{o}_{f,y} < -\vec{p}_y + \vec{s}_y$ gilt, dann setze
   $\vec{p}'_y := -\vec{o}_{f,y}$.
4. Sonst wenn $\vec{o}_{f,y} + \vec{s}_{f,y} > -\vec{p}_y + \vec{s}_y$ gilt, dann setze
   $\vec{p}'_y := -(\vec{o}_{f,y} + \vec{s}_{f,y} - \vec{s}_y)$.

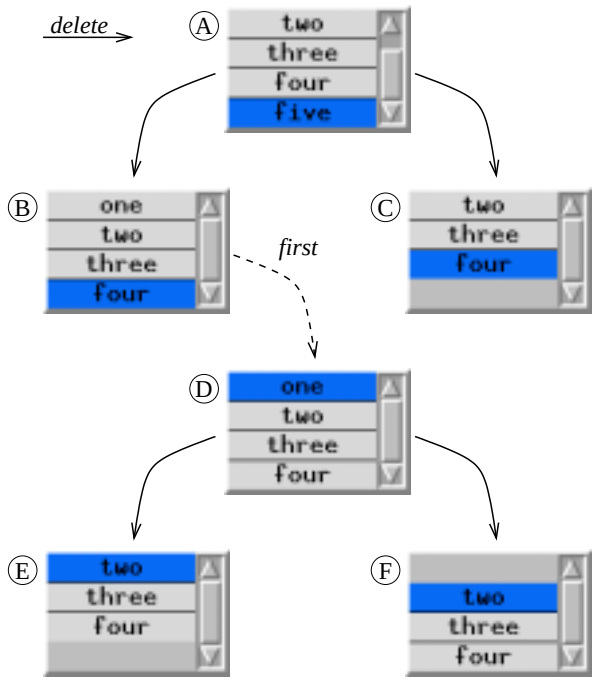


Abbildung 5.11: Mögliche Inkonsistenzen für scrollV durch Größenänderungen

Beispiel 5.8 Mögliche Inkonsistenzen für scrollV durch Größenänderungen

In der Abbildung zeigt der Finger in der Ausgangssituation Ⓐ auf das letzte Element einer Menge mit fünf Elementen, deren erstes Element nicht sichtbar ist. Nach dem Löschen paßt die Menge komplett in den Viewport, der Balken im Rollbereich ist entsprechend länger geworden. Links Ⓑ wurde der sichtbare Bereich korrekt so angepaßt, daß das vierte Element („four“) am unteren Rand des Viewports ist. Rechts Ⓒ wurde der sichtbare Bereich nicht angepaßt, so daß durch den freien Bereich, der am unteren Rand des Viewports sichtbar geworden ist, der Eindruck entsteht, daß die Menge nur drei Elemente hat.

Wird ausgehend von Situation Ⓑ der Finger an den Anfang der Menge bewegt (Operation *first*), entsteht Situation Ⓓ. Nach dem Löschen des ersten Mengenelementes ist die Menge kleiner als der Viewport. Links Ⓔ wurde der sichtbare Bereich korrekt so angepaßt, daß das neue erste Element („two“) am oberen Rand des Viewports ist. Rechts Ⓕ wurde der sichtbare Bereich nicht angepaßt, so daß die Menge „in der Luft hängt“. □

Wenn der Finger höher ist als der Viewport, wird die obere Kante des Fingers mit der oberen Kante des Viewports in Deckung gebracht (1). Ansonsten wird der Finger im Viewport zentriert, wenn er momentan überhaupt nicht sichtbar ist (2). Ist der Fingers am unteren Rand des Viewports teilweise sichtbar, wird die untere Kante des Fingers mit der unteren Kante des Viewports in Deckung gebracht (3). Ist der Fingers am oberen Rand des Viewports teilweise sichtbar, wird die obere Kante des Fingers mit der oberen Kante des Viewports in Deckung gebracht (4).

Der neue vertikale Finger-Offset $\vec{co}'_y$ wird wie in der Methode Position berechnet.

Methode Finger: Ein Finger wird analog atom durch veränderte Hintergrund- bzw. Randfarben dargestellt. Zu beachten ist, daß auch die Färbung des Rollbalkens angepaßt wird, insbesondere auch die Farben, die eine Aktivierung des Rollbalkens signalisieren. In einer Tcl/DB-Implementierung sind dazu die Optionen `activebackground`, `highlightbackground`, `troughcolor` und `background` eines `scrollbar` Widgets zu verändern [Ous94, S. 169 ff.].

Meta-Daten: Abb. 5.12 zeigt den Eintrag für scrollV in der Visualisierungsfunktionen-Tabelle. Der Default-Wert für den Parameter `height` reflektiert die Fußnote b in Tabelle 5.11 auf Seite 198: Er wird durch die Ausführung der Methode `p_height` des Konstruktions-Visualisierungsknotens (Substitution für „`%V`“, siehe Tabelle 5.2 auf Seite 179) bestimmt, an dem scrollV als Visualisierungsfunktion eingetragen ist.

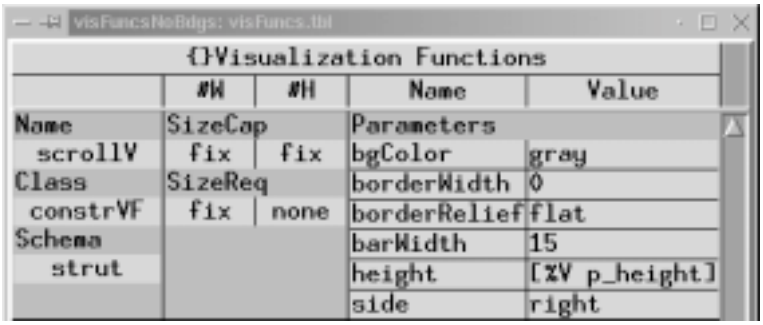


Abbildung 5.12: Eintrag für scrollV in der Visualisierungsfunktionen-Tabelle

5.5.2 Die Visualisierungsfunktionen scrollH und scroll

Die Aussagen zu der Konstruktions-Visualisierungsfunktion scrollV lassen sich analog auf scrollH übertragen. Zusammengefaßt ergeben sie die Funktionalität von scroll. scroll kann auch so parametrisiert werden, daß eine generelle Konstruktions-Visualisierungsfunktion entsteht, die die Funktionalität aller drei genannten umfaßt. Wegen der Übersichtlichkeit ist hier nur die spezielle Variante scrollV beschrieben worden; für scrollH und scroll werden

in Abb. 5.13 lediglich deren Einträge in der Visualisierungsfunktionen-Tabelle dargestellt. Die Anmerkung zum Parameter `height` bei scrollV auf der vorherigen Seite gilt auch für den Parameter `width` von scrollH bzw. die Parameter `height` und `width` von scroll.

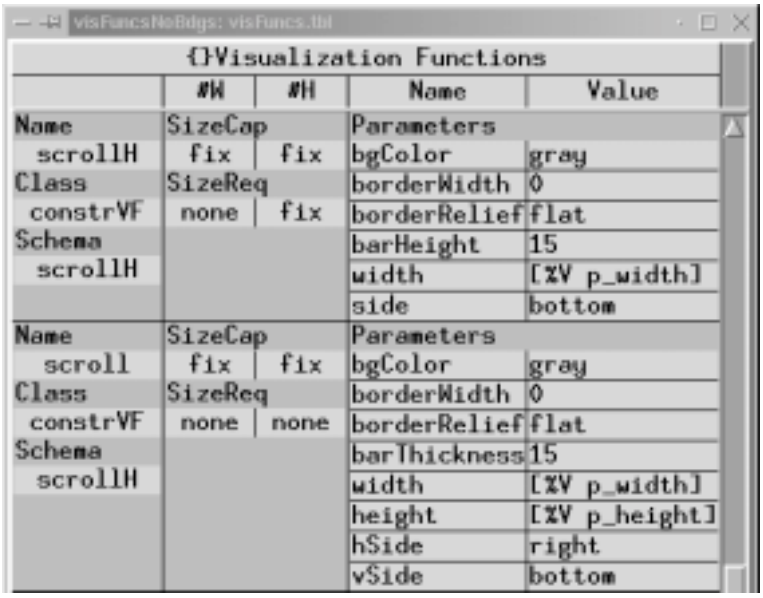


Abbildung 5.13: Einträge für scrollH und scroll in der Visualisierungsfunktionen-Tabelle

5.6 Komplexe datenbezogene Visualisierungsfunktionen

Die virtuelle Klasse cplxDataVF erweitert das Interface für komplexe, datenbezogene Visualisierungsfunktionen um die generischen Navigationsmethoden für die Interaktion Push, Move_Next, Move_Back, Move_First und Move_Last, die datenbezogenen Navigationsmethoden First und Iterate, sowie die Methode Sub_Finger zur Unterstützung der Fingervisualisierung. Außerdem redefiniert sie die Methode Sub. Eine Übersicht des Interface ist in Tabelle 5.12 auf der gegenüberliegenden Seite dargestellt.

Für alle komplexen datenbezogenen Visualisierungsfunktionen werden von der Klasse cplxDataVF weitere Default-Bindungen definiert. Eine Übersicht der Default-Bindungen für cplxDataVF ist in Tabelle 5.13 auf der gegenüberliegenden Seite dargestellt.

5.7 Tupel-Visualisierungsfunktionen

Tupel-Visualisierungsfunktionen integrieren Repräsentationen der darzustellenden Komponenten, die jeweils durch eigene Visualisierungsknoten (bzw. -Teilbäume) definiert werden.

Klasse	Methoden
visFunc	Width, Height
	Pre, Post
	Finger
dataVF	Pop
cplxVF	Resize
	Position
	Adjust
	Sub
cplxDataVF	First, Iterate
	Sub_Finger
	Push
	Move_Next, Move_Back
	Move_First, Move_Last

Tabelle 5.12: Interface der virtuellen Klasse cplxDataVF

Klasse	Ereignis	IH	Operation
dataVF	`<Enter>`	Nein	F Push
	`<Escape>`	Nein	F Pop
cplxDataVF	`<Home>`	Ja	F Move `first`
	`<End>`	Ja	F Move `last`
	`<Control-n>`	Ja	F Move `next`
	`<Control-p>`	Ja	F Move `back`

Tabelle 5.13: Default-Bindungen der Klasse cplxDataVF

Mit dem Tupel-Konstruktor definierte Typen werden per Default von der Visualisierungsfunktion tpl dargestellt (vgl. Abschnitt 5.7.1).

Das Interface der Klasse cplxDataVF wird nicht erweitert, aber die Signaturen der Methoden First und Iterate werden gemäß Abschnitt 4.2.3 redefiniert. Die First- und Push-Methoden bestimmen die im Sinne der jeweiligen Visualisierungsfunktion erste Komponente, d. h. den entsprechenden Nachfolgerknoten. Die Move-Methoden werden aus dem Kontext einer Komponente aufgerufen; sie sind immer relativ zur Position dieser „aktiven“ Komponente auszuführen. Ihre Signaturen bestehen aus den Informationen über das Tupel und die aktive Komponente. Die Move-Methoden geben als Ausführungsstatus zurück, ob die Bewegung erfolgreich war, d. h. ob tatsächlich eine neuer Komponentenknoten bestimmt wurde.

Zusätzlich zu den Bindungen der Klasse cplxDataVF werden keine weiteren Bindungen definiert, insbesondere auch nicht für die relativen *move*-Operationen *next* und *back*, da die Zuordnung zu Tastaturereignissen von den Realisierungen der einzelnen Tupel-Visualisierungsfunktionen und/oder von Parametern abhängig ist.

5.7.1 Die Visualisierungsfunktion tpl

Die Visualisierungsfunktion tpl ist die Default-Visualisierungsfunktion für Tupel, d. h. mit dem Tupel-Konstruktor definierte Typen werden per Default mit dieser Visualisierungsfunktion dargestellt. Sie muß daher in der Lage sein, auch variabel breite und hohe Komponententypen darzustellen. tpl implementiert keine Methode Adjust, da sie keinen adjustierbaren Viewport realisiert.

Parameter: Die Komponentenrepräsentationen werden von tpl horizontal oder vertikal konkateniert, d. h. sie werden nebeneinander bzw. übereinander, getrennt durch senkrechte bzw. waagerechte Linien (Streben, engl. *struts*) dargestellt. Dieses Verhalten ist über den Parameter `orientation` einstellbar, der Default ist horizontal (`orientation=horizontal`). Im Folgenden werden die Eigenschaften von tpl für die horizontale Darstellung beschrieben, sie gelten aber für die vertikale Darstellung analog. Die für tpl definierten Parameter und ihre Default-Werte sind in Tabelle 5.14 auf der gegenüberliegenden Seite dargestellt.

Tastaturereignisse: Für tpl werden, entsprechend der Parametrisierung gemäß `orientation`, zusätzliche Default-Bindungen für die relativen *move*-Operationen *next* und *back* definiert. In Tabelle 5.15 auf der gegenüberliegenden Seite sind alle Default-Bindungen dargestellt.

Methoden Width und Height, Längeneigenschaften und -anforderungen: Die äußere Eigenschaft bzgl. der Breite eines Tupels ist von den zur Verfügung gestellten Eigenschaften der Sub-Visualisierungsfunktionen abhängig: Haben alle Sub-Visualisierungsfunktionen feste Breite, kann das Tupel ebenfalls eine feste Breite (die Summe der Sub-Visualisierungsbreiten zzgl. der Stützen und Ränder) zur Verfügung

Name	Beschreibung	Default	Typ
`orientation`	Richtung der Konkatenation	`horizontal`	Orientation
`strutWidth`	Dicke der Streben	`1`	numerisch
`strutRelief`	Relief der Streben	`flat`	Relief
`color`	Farbe der Streben	`black`	Color
`bgColor`	Rand- und Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief

Tabelle 5.14: Parameter der Visualisierungsfunktion tpl

Klasse	Ereignis	IH	Operation
dataVF	`<Enter>`	Nein	F Push
	`<Escape>`	Nein	F Pop
cplxDataVF	`<Home>`	Ja	F Move `first`
	`<End>`	Ja	F Move `last`
	`<Control-n>`	Ja	F Move `next`
	`<Control-p>`	Ja	F Move `back`
tpl[a]	`<Right>`	Ja	F Move `next`
	`<Left>`	Ja	F Move `back`
tpl[b]	`<Down>`	Ja	F Move `next`
	`<Up>`	Ja	F Move `back`

Tabelle 5.15: Default-Bindungen der Visualisierungsfunktion tpl

[a] Bei horizontaler Konkatenation, d. h. `orientation=horizontal`.
[b] Bei vertikaler Konkatenation, d. h. `orientation=vertical`.

stellen. Die Berechnung für einen Tupel-Visualisierungsknoten v ist wie folgt rekursiv definiert:

$$\mathrm{w}(v) := 2 \cdot \texttt{borderWith} + (|\mathrm{sub}(v)| - 1) \cdot \texttt{strutWith} + \sum_{v_{sub} \in \mathrm{sub}(v)} \mathrm{w}(v_{sub})$$

Hat eine Sub-Visualisierungsfunktion variable Breite, gilt dies auch für das Tupel. Bei horizontaler Konkatenation geht durch variable Komponentenbreiten jedoch die vertikale visuelle Korrelation verloren.

Die gleichen Aussagen gelten auch für die äußere Eigenschaft bzgl. der Höhe eines Tupels, bis auf die Berechnung einer festen Höhe. Diese ergibt sich aus dem Maximum der Sub-Visualisierungsfunktionen zzgl. der Ränder:

$$\mathrm{h}(v) := 2 \cdot \texttt{borderWith} + \max_{v_{sub} \in \mathrm{sub}(v)} \mathrm{h}(v_{sub})$$

Da sich die Eigenschaften variabler Längen von innen nach außen fortpflanzen, stellt die Visualisierungsfunktion keine Anforderungen an die Längen ihrer Sub-Visualisierungsfunktionen.

Methoden Pre und Post: Sind Höhe und/oder Breite des Tupels variabel, können die Längen auch nicht in der Pre-Methode berechnet werden. Nachdem die Komponenten pre-visualisiert wurden und daher ihre Längen bekannt sind, werden die Längen des Tupels in der Post-Methode analog zu den obigen Formeln für Width und Height berechnet, hier für ein Tupel o[6]:

$$\mathrm{w}(o) := 2 \cdot \texttt{borderWith} + (|\mathrm{comps}(o)| - 1) \cdot \texttt{strutWith} + \sum_{o_{sub} \in \mathrm{comps}(o)} \mathrm{w}(o_{sub})$$

$$\mathrm{h}(o) := 2 \cdot \texttt{borderWith} + \max_{o_{sub} \in \mathrm{comps}(o)} \mathrm{h}(o_{sub})$$

Da die Summe der Komponenten- und Strebenbreiten durch die Iterate- und Sub-Methoden während der Post-Visualisierung gebildet wird, kann die Visualisierungbreite eines Tupels o in der Methode Post aus dem Visualisierungs-Delta $\vec{\Delta} = (\delta_x, \delta_y)$ (siehe Abschnitt 4.2.1) berechnet werden:

$$\mathrm{w}(o) := 2 \cdot \texttt{borderWith} + \delta_x$$

Um das Maximum der Komponentenhöhen iterativ durch die Methode Sub bestimmen zu können, muß zwischen den einzelnen Aufrufen (Schritt 5.2.3 auf Seite 131 der Post-Methode eines Visualisierungsknotens) die temporäre Speicherung objektspezifischen Informationen möglich sein.

[6] Die Funktion comps() liefert die Menge der Komponenten eines Tupels.

Der Viewport kann immer links oben eingeschränkt werden. Beinhaltet die Einschränkung den Rand des Tupels, wird der Offset als Ursprung initialisiert, ansonsten entspricht er der Randbreite. Sind Höhe und Breite fest, kann der Viewport auch rechts unten eingeschränkt werden. In Abschnitt 5.8.1 wird in Beispiel 5.10 auf Seite 218 die Einschränkung des Viewports und die entsprechenden Initialisierung des Offsets erläutert.

Methode Position: Die Position einer Komponentenrepräsentation läßt sich, wiederum abhängig von den Visualisierungsbreiten der Komponenten, analog zu den Formeln aus dem vorigen Abschnitt berechnen.

Angenommen, in einer Tcl/DB-Implementierung werden die einzelnen Komponentenrepräsentationen sowie die Streben mit dem Geometrie-Manager `pack` [Ous94, S. 183 ff.] arrangiert. Dann kann die horizontale Position der Komponentenrepräsentation *cw* in der Tupelrepräsentation *tw* wie folgt berechnet werden:

```
set oX $borderWidth
foreach w [pack slaves $tw] {
    if {$w == $cw} break
    incr oX [winfo width $w]
}
```

Methoden First, Sub und Iterate: Die Methode First liefert die erste Komponente eines Tupels, die für tpl durch den ersten Nachfolgerknoten in der Visualisierung bestimmt ist. In einer Tcl/DB-Implementierung der Methode First wird dazu die Option `push` des *fid*-Kommandos mit einem Pfad benutzt:

```
set res [$vf push "sub-atts.1"]
```

Komponentenrepräsentationen werden von der Methode Sub in die Tupelrepräsentation integriert. Wenn die Strebendicke nicht Null ist, wird außer für die erste Komponentenrepräsentation vor der Komponentenrepräsentation eine Strebe erzeugt. Resultat ist die Größe der Komponentenrepräsentation ggf. zzgl. der Strebendicke als Visualisierungs-Delta.

Die nächste Komponente eines Tupels wird von tpl in der Methode Iterate durch den nächsten Nachfolgerknoten in der Visualisierung bestimmt. In einer Tcl/DB-Implementierung wird die Option `go` des *fid*-Kommandos mit dem Parameter `-next` verwendet:

```
set res [$vf go -next]
```

Außerdem korrigiert sie, gemäß dem Parameter `orientation`, den Offset der aktuellen Visualisierungsposition: Ist `orientation`=`horizontal`, wird der X-Offset um

das von der Methode Sub berechnete X-Delta erhöht, ist `orientation=vertical`, wird der Y-Offset um das von der Methode Sub berechnete Y-Delta erhöht.

Die Methoden First und Iterate realisieren also eine durch die Visualisierung gesteuerte Reihenfolge der Komponenten in der Repräsentation des Tupels.

Methode Resize: Ein Resize-Delta in der durch `orientation` vorgegebenen Dimension wird voll nach außen weitergegeben. Das kann nur für Tupel auftreten, deren Länge in dieser Dimension variabel ist, da auch die auslösende Komponente in der Dimension variabel sein muß. Ein Delta in der anderen Dimension kann, zumindest teilweise, kompensiert werden, falls die Komponentenrepräsentation, die den Aufruf der Resize-Methode ausgelöst hat, kleiner als die Tupelrepräsentation ist. In Abschnitt 5.8.1 wird in Beispiel 5.11 auf Seite 221 die teilweise Kompensation eines Resize-Deltas erläutert.

Methoden Finger und Sub_Finger: Wenn das Tupel in einem Nimbus ist, soll dies durch die Farbe des Hintergrundes und des Randes dargestellt werden. Die Methode Sub_Finger ist leer, da tpl die Repräsentation der einzelnen Komponenten nicht ergänzt.

Methode Push und Move-Methoden: Die Methode Push ist identisch mit der Methode First. Die Move-Methoden bestimmen die jeweiligen neuen aktuellen Elemente. In einer Tcl/DB-Implementierung wird die Option go des *fid*-Kommandos benutzt, z. B. mit dem Parameter `-first` für Move_First:

```
set res [$f go -first]
```

Meta-Daten: In Abb. 5.14 auf der gegenüberliegenden Seite ist der Eintrag für tpl in der Visualisierungsfunktionen-Tabelle dargestellt. Die parameterabhängigen Bindungen der relativen *move*-Operationen *next* und *back* werden dort für eine Tcl/DB-Implementierung realisiert: Die Tastaturereignisse, an die die jeweiligen Operationen zu binden sind, werden durch den Aufruf von Methoden `e_back` bzw. `e_next` des Tupel-Visualisierungsknotens (Substitution für „`%V`“, siehe Tabelle 5.2 auf Seite 179) bestimmt. `e_back` und `e_next` haben als Methoden des Tupel-Visualisierungsknotens Zugriff auf den Parameter `orientation`.

5.7.2 Die Visualisierungsfunktion tplLabel

Die Visualisierungsfunktion tplLabel stellt alle Komponenten eines Tupels mit ihren Attributnamen als Label versehen dar. Alle Komponenten, deren Knoten in der Visualisierung als Nachfolgerknoten des Tupel-Knotens benachbart sind und eine feste Höhe haben, werden jeweils als **Gruppen** zusammengefaßt; Komponenten, deren Visualisierungsfunktionen variable Höhe haben, bilden also „Einergruppen“. Die Komponentenrepräsentationen werden innerhalb der Gruppen vertikal konkateniert. Die Label werden über den zugehörigen

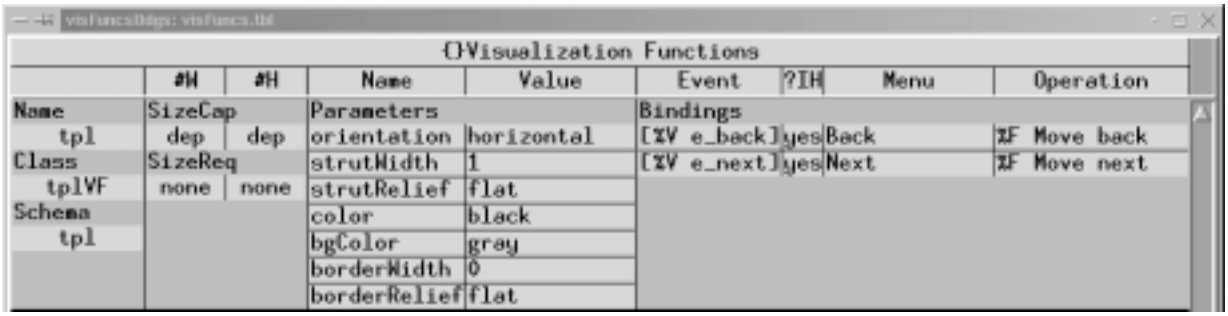


Abbildung 5.14: Eintrag für tpl in der Visualisierungsfunktionen-Tabelle

Komponenten in der Breite der jeweiligen Gruppe dargestellt. Die Gruppen selbst werden horizontal konkateniert. Im folgenden Beispiel wird tplLabel an einer Repräsentation von Polygon-Tupeln erläutert. tplLabel wird auch in den beiden Beispielanwendungen der Abschnitte 6.1.1 und 6.1.2 intensiv eingesetzt, siehe z. B. Abb. 6.6 auf Seite 236.

Beispiel 5.9 Repräsentation von Polygon-Tupeln mit tplLabel

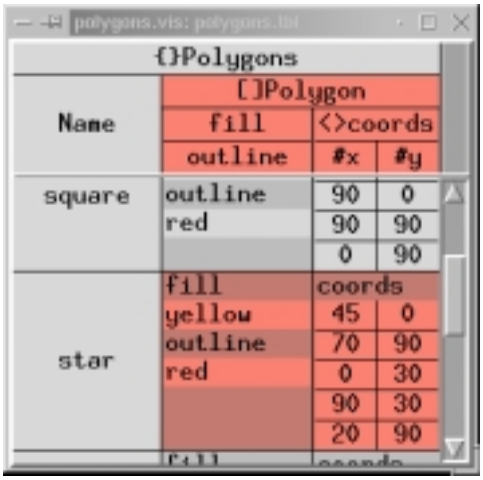


Abbildung 5.15: Repräsentation von Polygon-Tupeln mit tplLabel

In Abb. 5.15 sind die Polygon-Tupel aus Beispiel 5.5 auf Seite 189 mit tplLabel dargestellt; ein Finger zeigt auch hier auf das Polygon-Tupel namens „star“. In der mittleren Spalte der Repräsentation (linke Spalte der Polygon-Tupel) bilden die atomaren Komponenten fill und outline eine Gruppe, in der rechten Spalte ist die komplexe Komponente coords eine Gruppe für sich. Die Komponentenattributnamen fill, outline und coords sind sowohl im Schema als auch in der Tabelle dargestellt. □

Parameter: Die für tplLabel definierten Parameter und ihre Default-Werte sind in Tabelle 5.16 auf der nächsten Seite dargestellt. Die oben beschriebene Default-Gruppierung

kann durch den Parameter `groupAtts` beeinflußt werden, dessen Wertebereich eine Liste von Attributnamen ist. Ist `groupAtts` nicht nullwertig, beginnen nur die genannten Attribute eine neue Gruppe. Zusammen mit der Reihenfolge der Nachfolgerknoten in der Visualisierung können so alle möglichen Gruppierungen spezifiziert werden. Im Gegensatz zu tpl werden von tplLabel per Default keine Streben dargestellt, da die Abgrenzung der einzelnen Komponenten durch ihre Attributnamen ausreichend ist. Durch die Parameter `hStrutWidth` und `vStrutWidth` kann die Dicke der horizontalen Streben innerhalb der Gruppen und/oder vertikalen Streben zwischen den Gruppen definiert werden (ein Wert von 0 bedeutet, daß keine Strebe dargestellt wird).

Name	Beschreibung	Default	Typ
`font`	Schriftart für Label	`10x20`[a]	Font
`align`	Ausrichtung der Label	`left`	Alignment
`color`	Textfarbe	`black`	Color
`bgColor`	Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`strutColor`	Farbe der Streben	`black`	Color
`hStrutWidth`	Dicke der horizontalen Streben	0	numerisch
`vStrutWidth`	Dicke der vertikalen Streben	0	numerisch
`strutRelief`	Relief der Streben	`flat`	Relief
`groupAtts`	Gruppierung	$\perp$	< String >[b]

Tabelle 5.16: Parameter der Visualisierungsfunktion tplLabel

[a] Plattformabhängig, aber keine Proportionalschrift.
[b] Eine Liste von Attributnamen.

Tastaturereignisse: Für tplLabel werden zusätzliche Default-Bindungen definiert. Zu beachten ist, daß die richtungsorientierten Tastaturereignisse, wie z. B. `<Right>`, relativ zu den von tplLabel definierten Gruppen interpretiert werden. Die Bindungen für die relativen *move*-Operationen *next* und *back* werden von der Klasse cplxDataVF übernommen. In Tabelle 5.17 auf der gegenüberliegenden Seite sind alle Default-Bindungen dargestellt.

Methoden Width und Height, Längeneigenschaften und -anforderungen: Die Visualisierungsfunktion stellt keine Anforderungen an ihre Sub-Visualisierungsfunktionen. Hat eine ihrer Sub-Visualisierungsfunktionen variable Breite oder Höhe, so gilt das Gleiche für die entsprechende äußere Längeneigenschaft. Hat eine Sub-Visualisierungsfunktion variable Breite, kann die Darstellung mehrerer Tupel übereinander, trotz der enthaltenen Attributnamen, unübersichtlich werden. Sie sollte dann als letzte Gruppe im Tupel dargestellt werden.

Klasse	Ereignis	IH	Operation
dataVF	<Enter>	Nein	F Push
	<Escape>	Nein	F Pop
cp|xDataVF	<Home>	Ja	F Move first
	<End>	Ja	F Move last
	<Control-n>	Ja	F Move next
	<Control-p>	Ja	F Move back
tp|Label|	<Right>	Ja	F Move nextGrp
	<Left>	Ja	F Move backGrp
	<Down>	Ja	F Move grpNext
	<Up>	Ja	F Move grpBack

Tabelle 5.17: Default-Bindungen der Visualisierungsfunktion tp|Label|

Stellen alle Sub-Visualisierungsfunktionen feste Höhe bzw. Breite zur Verfügung, kann auch das Tupel feste Höhe bzw. Breite zur Verfügung stellen. Diese errechnen sich dann aus dem Maximum der Höhen bzw. der Summe der Breiten aller Gruppierungen. Für die Höhe einer Gruppierung ist die Summe der Höhen der Gruppenmitglieder, zzgl. der Höhen der Label und horizontalen Streben, zu berechnen. Für einen Tupel-Visualisierungsknoten v, an dem tp|Label| eingetragen ist, gilt[7]:

$$\mathrm{h}(v) := 2 \cdot \texttt{borderWith} + \max_{G \in \mathrm{grps}(v)} \mathrm{h}(G)$$

$$\mathrm{h}(G) := (|G| - 1) \cdot \texttt{hStrutWith} + \sum_{v_{sub} \in G} \mathrm{h}(v_{sub}) + \mathrm{textH}(\texttt{font})$$

Die Breite einer Gruppe ergibt sich aus dem Maximum der Breiten der Gruppenmitglieder, wobei die Breite eines Gruppenmitglieds der größere Wert von Breite der Sub-Visualisierungsfunktion und Breite des Labels ist[8,9]:

$$\mathrm{w}(v) := 2 \cdot \texttt{borderWith} + (|\mathrm{grps}(v)| - 1) \cdot \texttt{vStrutWith} + \sum_{G \in \mathrm{grps}(v)} \mathrm{w}(G)$$

$$\mathrm{w}(G) := \max_{v_{sub} \in G} \max(\mathrm{w}(v_{sub}), \mathrm{textW}(\mathrm{attr}(v_{sub}, \texttt{font}))$$

[7] Die Funktion textH() liefert die Darstellungshöhe eines Strings, die aber nicht vom String, sondern nur von der verwendeten Schriftart abhängig ist.

[8] Die Funktion textW() liefert die Darstellungsbreite eines Strings, die vom String und von der verwendeten Schriftart abhängig ist.

[9] Die Funktion grps() liefert die Menge aller Gruppen, d. h. eine Menge von Mengen von Komponenten-Visualisierungsknoten.

Methoden Pre und Post: Bezüglich variabler Längen gelten die gleichen Aussagen wie im entsprechenden Abschnitt für tpl (siehe Seite 206). Zu beachten ist hier, daß Breite und Höhe wieder gruppenweise berechnet werden. Auch für tplLabel ist die Ausnutzung des horizontalen Visualisierungs-Deltas und die Maximum-Bestimmung durch die Methode Sub vorzuziehen.

Methode Position: Bei der Berechnung der Position einer Komponentenrepräsentation in der Tupelrepräsentation muß die Höhe der Label beachtet werden. Als Offset der Komponente wird das Offset des Labels berechnet, damit beim Adjustieren von Viewports (siehe Abschnitt 4.3.4) das Label sichtbar bleibt.

Wenn in einer Tcl/DB-Implementierung die Komponentenrepräsentationen mit dem Geometrie-Manager `place` angeordnet werden, dann kann der Offset eines Komponenten-Labels *cl* wie folgt berechnet werden:

```
set cli [place info $cl]
set o(x) [lindex $cli [expr [lsearch -exact $cli -x]+1]]
set o(y) [lindex $cli [expr [lsearch -exact $cli -y]+1]]
```

Methoden First, Sub und Iterate: Die Methoden First und Iterate verfahren bei der Navigation im Tupel analog zu tpl. Für Iterate sind jedoch bei der Aktualisierung der Visualisierungsposition die Gruppierungen zu beachten: Ist die aktuelle Komponente die letzte einer Gruppe, wird der vertikale Offset zurückgesetzt und der horizontale Offset um die Breite der Gruppe, sowie ggf. die Breite der vertikalen Strebe, erhöht; ansonsten wird der vertikale Offset um die Höhe des Labels und der Komponentenrepräsentation, sowie ggf. die Breite der horizontalen Strebe, erhöht. Der horizontale Offset bleibt unverändert.

Die Methode Sub muß die Label der Komponentenattributnamen erzeugen, diese zusammen mit den Komponentenrepräsentationen plazieren und außer bei der ersten Komponente einer Gruppe vorher eine horizontale Strebe generieren. Bei den Labeln ist darauf zu achten, daß sie die ganze Breite ihrer Gruppe ausfüllen, damit ihre Färbung durch die Methode Sub_Finger in Verbindung mit der Navigation durch *next*-Operationen keine „springende“ Fingerbreite hervorruft. Für die ersten Komponenten einer Gruppe muß außer für die erste Gruppe vorher eine vertikale Strebe generiert werden.

Methode Resize: Ein positives Resize-Delta in horizontaler Richtung kann, zumindest teilweise, kompensiert werden, wenn die auslösende Komponente nicht die Breite ihrer Gruppe bestimmt. Umgekehrt kann ein negatives Resize-Delta in horizontaler Richtung, zumindest teilweise, kompensiert werden, wenn die auslösende Komponente die Breite ihrer Gruppe bestimmt. Für die vertikale Richtung gelten analoge Aussagen, wobei sich das Resize-Delta direkt auf die Höhe der Gruppe auswirkt und die Höhe der Tupelrepräsentation nur ändert, wenn die Gruppe deren Höhe bestimmt.

Methoden Finger und Sub_Finger: Wieder wird eine Fingervisualisierung durch farbliche Hervorhebung dargestellt. Neben Hintergrund und Rand sollen auch die Label gefärbt werden. Ein Label soll jedoch auch gefärbt sein, wenn nicht das ganze Tupel, sondern nur die entsprechende Komponente des Tupels in einem Nimbus ist. Diese Aufgabe implementiert die Methode Sub_Finger.

Methode Push und Move-Methoden: Die Methoden Push, Move_First, Move_Last, Move_Back und Move_Next sind genau wie die gleichnamigen Methoden von tpl implementiert. Die Methoden Move_GrpNext und Move_GrpBack bestimmen Komponenten nur innerhalb einer Gruppe neu — ist die aktuelle Komponente die letzte bzw. erste einer Gruppe, wird keine neue Komponente bestimmt. Die Methoden Move_NextGrp und Move_BackGrp bestimmen, falls die aktuelle Komponente nicht in der letzten bzw. ersten Gruppe liegt, als neue die erste Komponente der nächsten bzw. vorherigen Gruppe.

Meta-Daten: Abb. 5.16 zeigt den Eintrag für tplLabel in der Visualisierungsfunktionen-Tabelle.

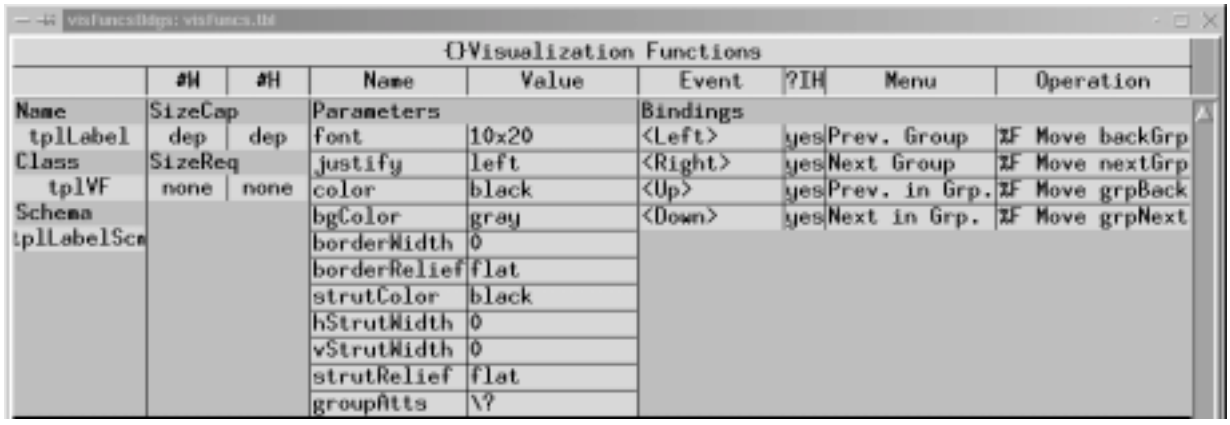


Abbildung 5.16: Eintrag für tplLabel in der Visualisierungsfunktionen-Tabelle

5.7.3 Die Visualisierungsfunktion subAttr

Die Visualisierungsfunktion subAttr ist für die Schemavisualisierung spezialisiert (siehe Abschnitt 3.7). Diese Spezialisierung drückt sich schon dadurch aus, daß sie eine Tupel-Visualisierungsfunktion ist, die zur Darstellung von Kollektionen herangezogen wird. Das bedeutet, daß die Elementrepräsentationen mit verschiedenen Visualisierungsfunktionen generiert werden können. Die Definition eines Visualisierungsknotens und seiner Nachfolgerknoten muß auf die darzustellenden Daten abgestimmt sein. Das impliziert, daß diese

Visualisierungsfunktion nur für statische Daten eingesetzt werden kann. Genau diese Eigenschaft besitzen Schemata als Datenobjekte, speziell für deren Darstellung die Visualisierungsfunktion subAttr definieren ist.

Es zeigt sich, daß die Implementierung von subAttr ganz analog zur Implementierung von tpl in der horizontalen Parametrisierung ist. In einer Tcl/DB-Implementierung kann sogar die Visualisierungsfunktion tpl in der entsprechenden Parametrisierung verwendet werden, da die Behandlung von Tupelkomponenten und Kollektionselementen mittels der Optionen `push` und `move` der *fid*-Kommandos (siehe Abschnitt 2.3.4, insbesondere Tabelle 2.2 auf Seite 56) identisch ist. Hier wird daher nicht weiter auf die Implementierung von subAttr eingegangen.

5.8 Kollektions-Visualisierungsfunktionen

Kollektions-Visualisierungsfunktionen integrieren Elementerepräsentationen, die alle durch einen Nachfolgerknoten definiert und durch dieselbe Visualisierungsfunktion generiert werden. In einer Visualisierung hat ein Kollektions-Visualisierungsknoten also genau einen Nachfolger. Mit Kollektions-Konstruktoren (Menge, Liste, Multimenge) definierte Typen werden per Default von der Visualisierungsfunktion cln dargestellt (vgl. Abschnitt 5.8.1).

Das Interface der Klasse cplxDataVF wird um Methoden erweitert, die der Darstellung von nullwertigen und leeren Kollektionen dienen und die Kardinalität von Kollektionen beeinflussen. Die Signaturen der Methoden First und Iterate werden gemäß Abschnitt 4.2.3 redefiniert. Die Adjust-Methode wird z. B. von der Kollektions-Visualisierungsfunktion clnForm (siehe Abschnitt 5.8.2) implementiert, die einen adjustierbaren Viewport realisiert. Nullwertige und leere Kollektionen werden von den Methoden Null bzw. Empty dargestellt. Die Kardinalität von Kollektionen wird durch die Methoden Null_To_Empty, Empty_To_Null, Empty_To_Singleton, Singleton_To_Empty, Insert und Delete beeinflußt. Die First- und Push-Methoden bestimmen das, im Sinne der jeweiligen Visualisierungsfunktion, erste visualisierte Element (Sub-Objekt) der Kollektion. Die Move-Methoden werden aus dem Kontext eines Elementes aufgerufen; sie sind immer relativ zur Position dieses „aktiven" Elementes auszuführen. Ihre Signaturen bestehen aus den Informationen über die Kollektion und das aktive Element. Die Move-Methoden geben als Ausführungsstatus zurück, ob die Bewegung erfolgreich war, d. h. ob tatsächlich ein neues zu aktivierendes Element bestimmt wurde. Eine Übersicht des Interface ist in Tabelle 5.18 auf der gegenüberliegenden Seite dargestellt.

Für alle Kollektions-Visualisierungsfunktionen werden von der Klasse clnVF Default-Bindungen für die kardinalitätsverändernden Operationen definiert. Eine Übersicht aller für clnVF definierten Default-Bindungen ist in Tabelle 5.19 auf Seite 216 dargestellt. Zu beachten ist hierbei, daß die Umwandlung einer nullwertigen Kollektion in eine leere und einer leeren Kollektion in eine einelementige über die Bindung der *in*-Operation erfolgt (siehe

Klasse	Methoden
visFunc	Width, Height Pre, Post Finger
dataVF	Pop
cplxVF	Resize Position Adjust Sub
cplxDataVF	First, Iterate Sub_Finger Push Move_Next, Move_Back Move_First, Move_Last
clnVF	Null, Empty Null_To_Empty, Empty_To_Null Empty_To_Singleton, Singleton_To_Empty Insert, Delete

Tabelle 5.18: Interface der virtuellen Klasse clnVF

auch Abschnitt 4.3.1, Seite 145). Wie bereits in der Klasse tplVF werden keine Bindungen für die relativen *move*-Operationen definiert.

5.8.1 Die Visualisierungsfunktion cln

Die Visualisierungsfunktion cln ist die Default-Visualisierungsfunktion für Kollektionen, d. h. mit Kollektions-Konstruktoren (Menge, Liste, Multimenge) definierte Typen werden per Default mit dieser Visualisierungsfunktion dargestellt. Sie muß daher in der Lage sein, auch variabel breite und hohe Elementtypen darzustellen. cln implementiert keine Methode Adjust, da sie keinen adjustierbaren Viewport realisiert.

Parameter: Die Elementrepräsentationen werden von cln vertikal oder horizontal konkateniert, d. h. sie werden übereinander bzw. nebeneinander, getrennt durch waagerechte bzw. senkrechte Linien (Streben, engl. *struts*), dargestellt. Dieses Verhalten ist über den Parameter `orientation` einstellbar, der Default ist vertikal (`orientation=vertical`). Im Folgenden werden die Eigenschaften von cln für die vertikale Darstellung beschrieben, sie gelten aber für die horizontale Darstellung analog. Die für cln definierten Parameter und ihre Default-Werte sind in Tabelle 5.20 auf der nächsten Seite dargestellt. Textattribute (z. B. `font`) werden für die textuelle Darstellung von nullwertigen und leeren Mengen verwendet.

Klasse	Ereignis	IH	Operation
dataVF	<Enter>	Nein	*F* Push
	<Escape>	Nein	*F* Pop
cplxDataVF	<Home>	Ja	*F* Move first
	<End>	Ja	*F* Move last
	<Control-n>	Ja	*F* Move next
	<Control-p>	Ja	*F* Move back
clnVF	<Insert>	Ja	*F* Insert after
	<Shift-Insert>	Ja	*F* Insert before
	<Delete>	Ja	*F* Delete current
	<BackSpace>	Ja	*F* Delete previous
	<Control-0>	Nein	*F* Null_Empty to_null

Tabelle 5.19: Default-Bindungen der Klasse clnVF

Name	Beschreibung	Default	Typ
orientation	Richtung der Konkatenation	vertical	Orientation
strutWidth	Dicke der Streben	1	numerisch
strutRelief	Relief der Streben	flat	Relief
color	Textfarbe und Farbe der Streben	black	Color
bgColor	Rand- und Hintergrundfarbe	gray	Color
borderWidth	Dicke des Randes	0	numerisch
borderRelief	Relief des Randes	flat	Relief
font	Schriftart	10x20[a]	Font

Tabelle 5.20: Parameter der Visualisierungsfunktion cln

[a] Plattformabhängig, aber keine Proportionalschrift.

Tastaturereignisse: Für cln werden, entsprechend der Parametrisierung gemäß `orientation`, zusätzliche Default-Bindungen für die relativen *move*-Operationen *next* und *back* definiert. In Tabelle 5.21 sind alle Default-Bindungen dargestellt.

Klasse	Ereignis	IH	Operation
dataVF	`<Enter>`	Nein	F Push
	`<Escape>`	Nein	F Pop
cplxDataVF	`<Home>`	Ja	F Move `first`
	`<End>`	Ja	F Move `last`
	`<Control-n>`	Ja	F Move `next`
	`<Control-p>`	Ja	F Move `back`
clnVF	`<Insert>`	Ja	F Insert `after`
	`<Shift-Insert>`	Ja	F Insert `before`
	`<Delete>`	Ja	F Delete `current`
	`<BackSpace>`	Ja	F Delete `previous`
	`<Control-0>`	Nein	F Null_Empty `to_null`
cln[a]	`<Down>`	Ja	F Move `next`
	`<Up>`	Ja	F Move `back`
cln[b]	`<Right>`	Ja	F Move `next`
	`<Left>`	Ja	F Move `back`

Tabelle 5.21: Default-Bindungen der Visualisierungsfunktion cln

[a] Bei vertikaler Konkatenation, d. h. `orientation=vertical`.
[b] Bei horizontaler Konkatenation, d. h. `orientation=horizontal`.

Methoden Width und Height, Längeneigenschaften und -anforderungen:
Die äußere Eigenschaft bzgl. der Breite einer Kollektion ist von der zur Verfügung gestellten Eigenschaften der Sub-Visualisierungsfunktion abhängig: Hat sie feste Breite, kann die Kollektion ebenfalls eine feste Breite (die gleiche wie die Sub-Visualisierungsfunktion zzgl. der Ränder) zur Verfügung stellen. Die Berechnung für einen Kollektions-Visualisierungsknoten v ist dann wie folgt:

$$\mathrm{w}(v) := 2 \cdot \texttt{borderWith} + \mathrm{w}(v_{sub})$$

Hat die Sub-Visualisierungsfunktion variable Breite, gilt dies auch für die Kollektion.

Die äußere Eigenschaft bzgl. der Höhe einer Kollektion ist immer variabel, auch wenn die Sub-Visualisierungsfunktion eine feste Höhe hat (dann kann jedoch die Höhe einer Kollektion bei der Pre-Visualisierung berechnet werden). Da die Höhe der Kollektion variabel ist und sich die Eigenschaft einer variablen Breite von innen nach außen fortpflanzt, stellt die Visualisierungsfunktion keine Anforderungen an die Längen ihrer Sub-Visualisierungsfunktion.

Methoden Pre und Post: Die Höhe einer Kollektion kann in der Pre-Methode nur berechnet werden, wenn sie nullwertig oder leer ist oder die Höhen der Elemente fest sind. Die ersten beiden Fälle werden weiter unten im Zusammenhang mit den Methoden Null bzw. Empty besprochen. Im letzten Fall resultiert die Höhe einer Kollektion o aus der folgenden Formel:

$$\mathrm{h}(o) := 2 \cdot \texttt{borderWith} + (|o| - 1) \cdot \texttt{strutWith} + |o| \cdot \mathrm{h}(v_{sub})$$

Ist außerdem die Breite der Sub-Visualisierungsfunktion und damit die der Kollektion fest, können ihre Eigenschaften als Ergebnis der Pre-Visualisierung berechnet werden.

Der Viewport kann immer links oben eingeschränkt werden. Beinhaltet die Einschränkung den Rand der Kollektion, wird der Offset als Ursprung initialisiert, ansonsten entspricht er der Randbreite. Wurden die Eigenschaften berechnet, kann der Viewport auch rechts unten eingeschränkt werden. Die Veränderung von Viewport und Offset wird im Folgenden an einem Beispiel erläutert:

(a) Feste Elementhöhen

(b) Variable Elementhöhen

Abbildung 5.17: Veränderung von Viewport und Offset durch die Pre-Methode von cln

Beispiel 5.10 Veränderung von Viewport und Offset durch die Pre-Methode von cln

In Abb 5.17 sind schematisch zwei Repräsentationen von Kollektionen mit Rändern in einem umgebenden Viewport dargestellt. Der Viewport ist jeweils durch den dunkel hinterlegten Bereich, horizontale und vertikale Abstände sind durch $x_0, x_1, \ldots$ bzw. $y_0, y_1, \ldots$ markiert. Vor Aufruf der Pre-Methode hat der Viewport die Abmessung $(x_5 - x_0, y_5 - y_0)$, der Offset ist (x_1, y_1). Bei festen Elementhöhen (Abb. 5.17(a)) kann der Viewport an beiden definierenden Ecken Ⓐ und Ⓑ, hier mit Rand, auf $(x_3 - x_2, y_3 - y_2)$ eingeschränkt werden. Der Offset ist dann $(0, 0)$. Bei variable Elementhöhen (Abb. 5.17(b)) kann der Viewport nur an der Ecke Ⓐ, hier ohne Rand, auf $(x_5 - x_1, y_5 - y_1)$ eingeschränkt werden. Der Offset ist in diesem Fall $(x_2 - x_1, y_2 - y_1)$. □

Sind die Höhen der Elemente variabel, also nicht im Voraus für jede Kollektion bekannt, müssen der Kollektions-Visualisierungsfunktion zur Berechnung der Höhe einer

Kollektion alle Elementhöhen bekannt sein. Dies ist erst in der Post-Methode der Fall, wenn alle Elemente pre-visualisiert wurden. Dann resultiert die Höhe der Kollektion aus der Summe der Elementhöhen, also für eine Kollektion o aus folgender Formel:

$$\mathrm{h}(o) := 2 \cdot \texttt{borderWith} + (|o| - 1) \cdot \texttt{strutWith} + \sum_{o_{sub} \in o} \mathrm{h}(o_{sub})$$

Da die Summe der Element- und Strebenhöhen durch die Iterate- und Sub-Methoden während der Post-Visualisierung gebildet wird, kann die Höhe der Kollektion in der Methode Post aus dem Visualisierungs-Delta $\vec{\Delta} = (\delta_x, \delta_y)$ berechnet werden:

$$\mathrm{h}(o) := 2 \cdot \texttt{borderWith} + \delta_y$$

Ist die Breite der Sub-Visualisierungsfunktion variabel, d. h. nicht im Voraus bekannt, resultiert die Breite der Kollektion aus dem Maximum der Elementbreiten, also folgender Formel:

$$\mathrm{w}(o) := 2 \cdot \texttt{borderWith} + \max_{o_{sub} \in o} \mathrm{h}(o_{sub})$$

Die Breite kann auch iterativ durch die Methode Sub bestimmt werden. Dazu muß zwischen den einzelnen Methodenaufrufen von Sub (Schritt 6.3.3.2 auf Seite 132 der Post-Methode eines Visualisierungsknotens) die temporäre Speicherung objektspezifischer Informationen möglich ist.

Methode Position: Die Position einer Elementrepräsentation in der Kollektionsrepräsentation läßt sich, wiederum abhängig von den Höhen der Elementrepräsentationen, analog zu den Formeln aus dem vorigen Abschnitt berechnen.

Angenommen, in einer Tcl/DB-Implementierung werden die einzelnen Elementrepräsentationen sowie die Streben mit dem Geometrie-Manager `pack` arrangiert. Dann kann die vertikale Position der Elementrepräsentation ew in der Kollektionsrepräsentation cw mit variabel hohen Elementen wie folgt berechnet werden:

```
set oY $borderWidth
foreach w [pack slaves $cw] {
    if {$w == $ew} break
    incr oY [winfo height $w]
}
```

Sind die Elementhöhen fest, kann die vertikale Position aus dem Elementindex ei berechnet werden:

```
set oY [expr $borderWidth + ($ei-1)*($strutWidth + $subHeight)]
```

Methoden First, Sub und Iterate: Die Methode First legt das erste Element einer Kollektion fest. In einer Tcl/DB-Implementierung wird die Option `push` des *fid*-Kommandos mit dem Parameter `-first` benutzt:

```
set res [$f push -first]
```

Elementrepräsentationen werden von der Methode Sub in die Kollektionsrepräsentation integriert. Wenn die Strebendicke nicht Null ist, wird, außer für die erste Elementrepräsentation, oberhalb der Elementrepräsentation eine Strebe erzeugt. Resultat ist die Größe der Elementrepräsentation, ggf. zzgl. der Strebendicke, als Visualisierungs-Delta.

Die Methode Iterate bestimmt, falls es existiert, das „nächste“ Element einer Kollektion. In einer Tcl/DB-Implementierung wird die Option `go` des *fid*-Kommandos mit dem Parameter `-next` benutzt:

```
set res [$f go -next]
```

Außerdem korrigiert sie, gemäß dem Parameter `orientation`, den Offset der aktuellen Visualisierungsposition: Ist `orientation=vertical`, wird der Y-Offset um das von der Methode Sub berechnete Y-Delta erhöht, ist `orientation=horizontal`, wird der X-Offset um das von der Methode Sub berechnete X-Delta erhöht.

Methoden für leere und nullwertige Kollektionen: Ist eine Kollektion leer oder nullwertig, wird Sie visuell durch die textuelle Repräsentation des entsprechenden Wertes dargestellt. In diesem Fall hängt die Höhe der Repräsentation von der verwendeten Schriftart ab. Ist die Breite der Element-Visualisierungsfunktion fest, wird der String zentriert dargestellt, da auch die Breite der Kollektions-Visualisierungsfunktion fest ist; ansonsten ist die Breite der Kollektions-Visualisierungsfunktion vom String und der verwendeten Schriftart abhängig.

Bei der Umwandlung zwischen nullwertigen, leeren und einelementigen Mengen muß lediglich darauf geachtet werden, daß als Visualisierungs-Delta jeweils die Größendifferenz zwischen den Repräsentationen berechnet wird.

Methoden Insert und Delete: Die Methode Insert muß berücksichtigen, daß vor jedem Element, bis auf dem ersten, eine Strebe dargestellt ist. Falls vor dem ersten Element eingefügt wird bedeutet das, daß zwischen dem neuen ersten und alten ersten Element eine Strebe generiert werden muß. Umgekehrt muß die Methode Delete beim Löschen des ersten Elementes die Strebe zwischen dem alten ersten und dem neuen ersten Element entfernen.

Methode Resize: Ein Resize-Delta in der durch `orientation` vorgegebenen Dimension wird voll nach außen weitergegeben. Ein Delta in der anderen Dimension kann, zumindest teilweise, nur dann kompensiert werden, wenn die Element-Visualisierungsfunktion in dieser Dimension variabel ist und die Repräsentation des Elementes, das

den Aufruf der Resize-Methode ausgelöst hat, kleiner als die Repräsentation der Kollektion ist. Die Teilweise Kompensation eines Resize-Deltas wird im Folgenden an einem Beispiel erläutert:

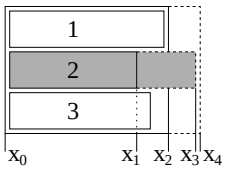


Abbildung 5.18: Teilweise Kompensation eines Resize-Deltas

Beispiel 5.11 Teilweise Kompensation eines Resize-Deltas

In Abb. 5.18 verbreitert sich das Element 2 von seiner ursprünglichen Breite x_1 auf x_3 (grau hinterlegtes Rechteck). Von der Kollektion wird das Resize-Delta $\vec{\Delta}_r = x_3 - x_1$ des breiter gewordenen Elementes 2 auf $\vec{\Delta}'_r = x_4 - x_2$ kompensiert. □

Methoden Finger und Sub_Finger: Wenn die Kollektion in einem Nimbus ist, soll dies durch die Farbe des Hintergrundes und des Randes dargestellt werden. Nullwertige und leere Mengen müssen besonders beachtet werden, da für diese auch der Hintergrund der textuellen Darstellung angepaßt werden muß. Die Methode Sub_Finger ist leer, da cIn die einzelnen Elementrepräsentationen nicht ergänzt.

Methode Push und Move-Methoden: Die Methode Push ist identisch mit der Methode First. Die Move-Methoden bestimmen die jeweiligen neuen aktuellen Elemente. In einer Tcl/DB-Implementierung wird die Option go des *fid*-Kommandos benutzt, z. B. mit dem Parameter `-first` für Move_First:

```
set res [$f go -first]
```

Meta-Daten: In Abb. 5.19 auf der nächsten Seite ist der Eintrag für cIn in der Visualisierungsfunktionen-Tabelle dargestellt. Die parameterabhängigen Bindungen werden dort wie für tpl durch Skriptaufrufe dargestellt (siehe Abschnitt 5.7.1, Seite 208).

5.8.2 Die Visualisierungsfunktion clnForm

Die Visualisierungsfunktion clnForm[10] realisiert eine elementweise Darstellung von Kollektionen, d. h. zu einem Zeitpunkt wird genau ein Element der Kollektion dargestellt.

[10] Der Name clnForm ist aus der Analogie zu formularartigen Darstellungen entstanden, die in formularbasierten Benutzerschnittstellen von RDBMS weit Verbreitet sind. Typischerweise werden formularartige Darstellungen daher im Kontext von Tupeln betrachtet, hier wird aber speziell deren Konzept der *one-tuple-at-a-time*-Darstellung von Relationen (Mengen von Tupeln) adaptiert.

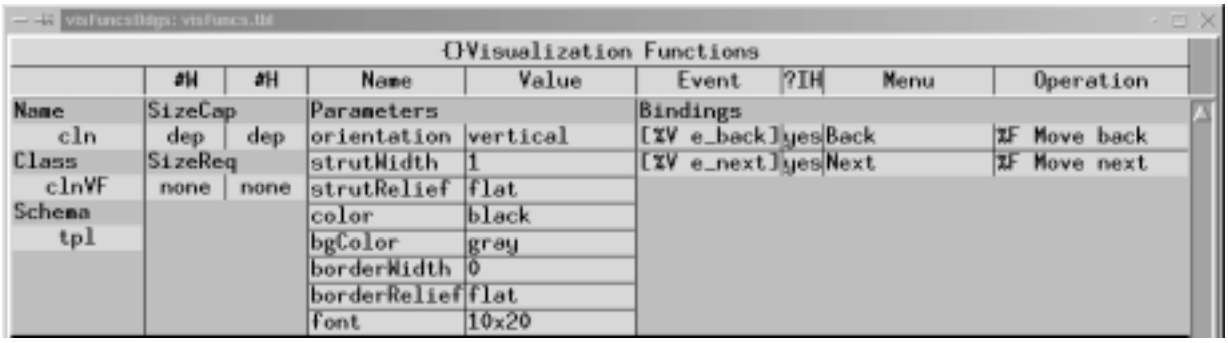
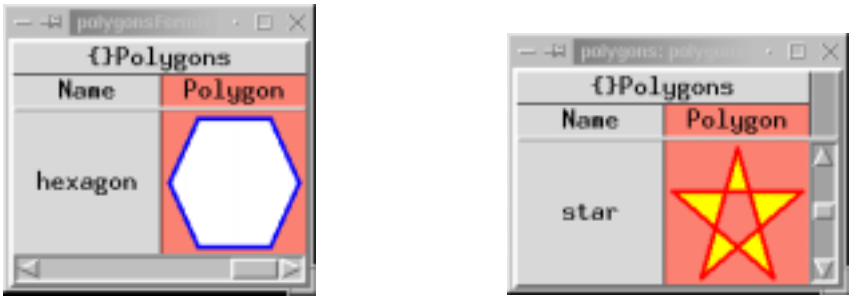


Abbildung 5.19: Eintrag für cln in der Visualisierungsfunktionen-Tabelle

Die Repräsentation enthält eine Positionierungshilfe, d. h. einen Vor- und einen Zurück-Knopf sowie eine Anzeige der relativen Position des aktuell angezeigten Elementes in der Kollektion. Wie in Abb. 5.20 dargestellt ist, kann die Positionierungshilfe kann eine vertikale oder horizontale Navigation suggerieren, wenn die Knöpfe als Hoch/Runter- oder Links/Rechts-Paar ausgelegt sind.



(a) horizontal (b) vertikal

Abbildung 5.20: Suggerierte Navigationsrichtungen von clnForm

Die Repräsentation ist dadurch einer Kollektion mit umgebendem Viewport sehr ähnlich. Ein wesentlicher Unterschied zu dieser ist, daß der sichtbare Ausschnitt nicht zwischen zwei Elementen positioniert werden kann, sondern immer genau ein Element der Kollektion sichtbar ist. Dieses Verhalten kann hier nur schlecht an einem Beispiel demonstriert werden; es ist z. B. für die Darstellung von Bildserien sinnvoll, wie Abb. 5.21 auf der gegenüberliegenden Seite zeigt. Zusammen mit einem Tcl-Skript könnten so z. B. auch Animationen realisiert werden.

Der zweite wesentliche Unterschied zu Viewport-Visualisierungsfunktionen ist, daß clnForm als Kollektions-Visualisierungsfunktion Zugriff auf die Elemente der Kollektion und die Element-Visualisierungsfunktion hat. Diese Eigenschaft ermöglicht es clnForm, für die Elemen-

Abbildung 5.21: Elementweise Positionierung von clnForm

trepräsentationen eine vom Mechanismus der späten Visualisierung (siehe Abschnitt 4.1.1) unabhängige Cache-Funktionalität zu implementieren.

Parameter: Über den Parameter `cache` kann die Größe des Caches für Elementrepräsentationen, d. h. die Anzahl der Elementrepräsentationen, die neben der Repräsentation des im Viewport sichtbaren Elementes verfügbar sind, eingestellt werden. Die weiteren für clnForm definierten Parameter und ihre Default-Werte sind in Tabelle 5.22 dargestellt. Textattribute (z. B. `font`) werden für die textuelle Darstellung von nullwertigen und leeren Mengen verwendet. Als Erweiterung von clnForm könnte über

Name	Beschreibung	Default	Typ
`orientation`	Navigationsrichtung	`horizontal`	Orientation
`cache`	Cache-Größe	`5`	numerisch
`color`	Textfarbe	`black`	Color
`bgColor`	Rand- und Hintergrundfarbe	`gray`	Color
`borderWidth`	Dicke des Randes	`0`	numerisch
`borderRelief`	Relief des Randes	`flat`	Relief
`font`	Schriftart	`10x20`[a]	Font

Tabelle 5.22: Parameter der Visualisierungsfunktion clnForm

[a] Plattformabhängig, aber keine Proportionalschrift.

zusätzliche Parameter die Anzahl gleichzeitig dargestellter Kollektionselemente, sowie deren visuelle Anordnung innerhalb der Repräsentation, variiert werden.

Tastaturereignisse: Für clnForm werden die gleichen Default-Bindungen wie für cln definiert (siehe Tabelle 5.21 auf Seite 217, wobei `orientation` nicht die Konkatenations- sondern die Navigationsrichtung parametrisiert).

Methoden Width und Height, Längeneigenschaften und -anforderungen: Ein generierte Repräsentation hat feste Abmessungen, was als Anforderung an die Sub-Visualisierungsfunktion ausgedrückt und als Eigenschaft nach außen zur Verfügung gestellt wird. Die Methoden Width und Height geben also stets definierte Werte zurück, bei deren Berechnung die Längen der Sub-Visualisierungsfunktion, die Randbreiten und je nach Orientierung der Platz für die Positionierungshilfe berücksichtigt werden müssen (siehe den entsprechenden Abschnitt für scrollV auf Seite 197).

Methoden Pre und Post: Da die Abmessungen fest sind, können die Eigenschaften als Ergebnis der Pre-Visualisierung zurückgegeben werden. Der Viewport wird auf den tatsächlich sichtbaren Bereich eingeschränkt (ohne Rand und Positionierungshilfe), der Offset auf den Ursprung initialisiert.

Methoden First, Sub und Iterate: Die Methode First legt das erste Element einer Kollektion wie die entsprechende Methode der Visualisierungsfunktion cln fest. Die Repräsentation des ersten Elementes wird von der Methode Sub integriert. Ihr Ergebnis ist unrelevant, da die Größen fest und daher in der Methode Iterate bereits bekannt sind. Letztere, die Methode Iterate, bestimmt kein nächstes Element, so daß nur ein Element pre-visualisiert wird (clnForm implementiert eine bedarfsgesteuerte Generierung von Elementrepräsentationen).

Generierung von Elementrepräsentationen: Da zunächst nur eine Elementrepräsentation generiert und angezeigt wird, müssen bei Bedarf weitere Repräsentationen von Elementen explizit generiert (siehe Abschnitt 4.2.4) und angezeigt werden. Der Bedarf entsteht durch eine Viewport-Veränderung, die explizit durch die Positionierungshilfen oder implizit durch eine Fingerbewegung ausgelöst wird. Das Generieren geschieht durch Aufruf der Methode Pre des Visualisierungsknotens, an dem clnForm eingetragen ist. Die Parameter Viewport und Offset werden dazu wie in der Pre-Methode der Visualisierungsfunktion clnForm auf dieser Seite festgelegt; das Datenobjekt wird durch die Fingernavigation oder die Bindungen der Positionierungshilfe bestimmt.

Beim Anzeigen der generierten Elementrepräsentation kommt die Cache-Funktionalität ins Spiel. Hat der Parameter `cache` einen Wert $n > 0$, wird die alte, d. h. vorher angezeigte Elementrepräsentation nicht zerstört und für den späteren Gebrauch in einem Cache der Größe n gespeichert. Die Repräsentation eines Elementes muß daher nur dann generiert werden, wenn sie nicht im Cache verfügbar ist, d. h. aus dem Cache gelöscht oder noch nie angezeigt wurde. Als Verdrängungsstrategie für den Cache könnte z. B. LRU angewandt werden (lange nicht angezeigte Elementrepräsentationen werden verdrängt). Aus dem Cache verdrängte Visualisierungen werden zerstört, um System-Ressourcen freizugeben. Hat `cache` den Wert 0, werden alte Elementrepräsentationen sofort zerstört und bei jeder Anzeige neu generiert.

Methode Position: Die Position einer Elementrepräsentation $id_{sub} = (o_{sub}, v_{sub})$ relativ zum sichtbaren Ausschnitt ist der Nullvektor ($\vec{p} = (h_{pos}, v_{pos})$ mit $h_{pos} = v_{pos} = 0$), wenn o_{sub} das momentan sichtbaren Elementes der Kollektion ist. Soll die Position

eines nicht sichtbaren Elementes bestimmt werden, werden dazu die Elementindizes und die (feste) Länge der Element-Visualisierungsfunktion verwendet. Die Position wird entsprechend der Parametrisierung der Navigationsrichtung in horizontaler oder vertikaler Richtung berechnet. Wenn i_{sub} und i_{vis} die Indizes von o_{sub} und dem sichtbaren Element sind und w_{sub} und h_{sub} Breite und Höhe der Element-Visualisierungsfunktion sind, dann sind

$$h_{pos} = (i_{sub} - i_{vis}) \cdot w_{sub} \quad \text{und} \quad v_{pos} = (i_{sub} - i_{vis}) \cdot h_{sub}$$

die horizontalen und vertikalen Positionen von o_{sub} in der Repräsentation bei horizontaler bzw. vertikaler Navigationsrichtung.

Methode Resize: Die Implementierung der Methode Resize ist leer. Da die Anforderung an die Sub-Visualisierungsfunktion feste Längen sind, kann der Methode Resize nur ein Resize-Delta von Null ($\vec{\Delta}_r = (\delta_x, \delta_y)$ mit $\delta_x = \delta_y = 0$) übergeben werden.

Methode Adjust: Die Methode Adjust stellt sicher, daß die Repräsentation eines Elementes angezeigt wird. Wurde vorher ein anderes Element angezeigt, wird der Finger-Offset wie bei der Methode Position über die beiden Elementindizes aktualisiert und die neue Elementrepräsentation angezeigt, die ggf. vorher wie oben beschrieben generiert wird.

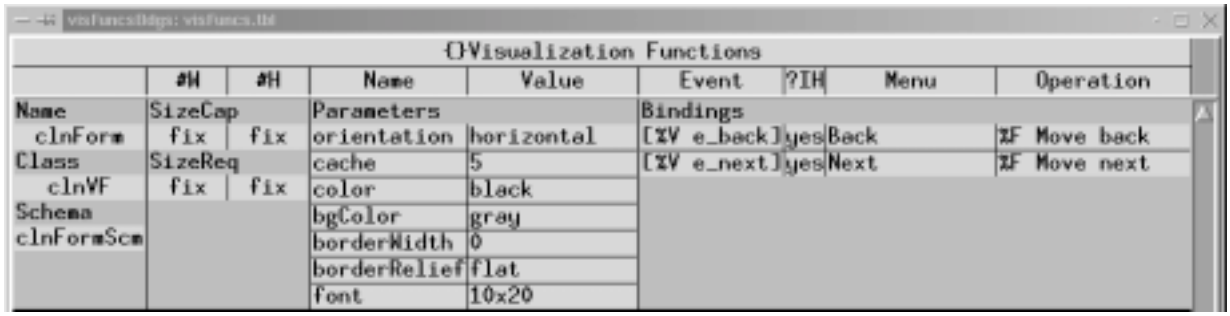
Methoden für nullwertige und leere Kollektionen: Nullwertige und leere Kollektionen werden wie bei cln durch die textuelle Darstellung des entsprechenden Wertes repräsentiert, wobei hier die Längen der Repräsentation fest sind. Daher muß auch bei der Umwandlung zwischen nullwertigen, leeren und einelementigen Mengen kein Visualisierungs-Delta berechnet werden, es ist immer Null.

Methoden Insert und Delete: Beim Einfügen neuer Elemente wird mit deren Repräsentation wie bei der Generierung von Elementrepräsentationen verfahren. Beim Löschen wird die Elementrepräsentation zerstört und nicht in den Cache aufgenommen.

Methode Push und Move-Methoden: Diese Methoden sind genau wie bei cln implementiert.

Methode Finger: Ein Finger wird wieder durch veränderte Hintergrund- bzw. Randfarben dargestellt. Zu beachten ist, daß auch die Färbung der Positionierungshilfe angepaßt wird.

Meta-Daten: In Abb. 5.22 auf der nächsten Seite ist der Eintrag für clnForm in der Visualisierungsfunktionen-Tabelle dargestellt.



{}Visualization Functions

	#W	#H	Name	Value	Event	?IH	Menu	Operation
Name	SizeCap		Parameters		Bindings			
clnForm	fix	fix	orientation	horizontal	[%V e_back]	yes	Back	%F Move back
Class	SizeReq		cache	5	[%V e_next]	yes	Next	%F Move next
clnVF	fix	fix	color	black				
Schema			bgColor	gray				
clnFormScm			borderWidth	0				
			borderRelief	flat				
			font	10x20				

Abbildung 5.22: Eintrag für clnForm in der Visualisierungsfunktionen-Tabelle

Kapitel 6

Anwendungen und Ausblick

Abschließend soll in diesem Kapitel zunächst an zwei Anwendungen exemplarisch der Einsatz des entwickelten Visualisierungsverfahrens demonstriert werden. Danach werden die Ergebnisse der Kapitel 3 bis 5 zusammengefaßt. Aspekte, die in dieser Arbeit nicht weiter behandelt werden konnten, kommen im letzten Abschnitt zur Sprache.

6.1 Anwendungen

Die beiden hier vorgestellten Anwendungen sind eng mit ESCHER verbunden und wurden auf der Basis von Tcl/Tk und Tcl/DB (siehe Abschnitt 2.3.4) in prototypischen Implementierungen bereits realisiert. Das erste Beispiel für den Einsatz des vorgestellten Visualisierungsverfahrens ist eine graphische Benutzerschnittstelle für ESCHER selbst; sie wird hier Tk-ESCHER genannt. Das zweite Beispiel umfaßt Visualisierungen der in Abschnitt 3.8 strukturierten Meta-Daten des Visualisierungsverfahrens, nämlich der Visualisierungen selber und der Visualisierungsfunktionen- und Visualisierungsklassen-Tabelle.

6.1.1 Tk-ESCHER

Zentraler Bestandteil von Tk-ESCHER ist die Definition einer geeigneten Visualisierung für die Meta-Tabelle Applications, deren Struktur bereits in Beispiel 2.2 auf Seite 47 diskutiert wurde. Die Anwendungen (en. *applications*) sind eine Klassifizierung aller im System definierten Tabellen und bilden eine einstufige Hierarchie. Sie formen einen Namensraum (*name space*), so daß in verschiedenen Anwendungen gleichnamige Tabellen enthalten sein können (in etwa vergleichbar mit den Table Spaces in ORACLE).

Eine besondere Anwendung ist die Systemanwendung, in der die (Meta-) Tabellen des Systems gespeichert werden; sie bildet also das Data Dictionary von ESCHER. In der Sy-

stemanwendung sind z. B. das Boot-Schema und die Meta-Tabelle Applications selbst sowie das zugehörige Schema gespeichert[1].

Die Visualisierung der Meta-Tabelle Applications wird im Folgenden „MetaSelect“ genannt, weil die generierte Repräsentation der Auswahl von Anwendungen und Tabellen dient. MetaSelect soll so aufgebaut sein, daß sie für eine Anwendung deren Name, die Beschreibung und eine Auswahl der in der Anwendung definierten Tabellen bzw. Schemata präsentiert. Für jede Tabelle soll deren Name und der Name des Schemas angezeigt werden. Der Schemaname soll dabei durch den String „(schema)“ ersetzt werden, falls die Tabelle ein Schema repräsentiert.

Die Navigation in der Tabelle soll so eingeschränkt sein, daß das übliche Hineingehen in das eine Anwendung repräsentierende Tupel (Attribut Application) den Finger direkt auf die erste Tabelle der Anwendung zeigen läßt, anstatt auf den Namen der Anwendung. Genauso soll die *out*-Operation auf einem eine Tabelle repräsentierenden Tupel (Attribut Table), die normalerweise den Finger auf die Menge der Tabellen zeigen lassen würde, diesen wieder auf das gesamte Application-Tupel zeigen lassen. Das Hineingehen in ein Table-Tupel soll die Repräsentation dieser Tabelle auslösen; dazu muß eine Zuordnung zwischen Schemata und Visualisierungen existieren, die z. B. in einer weiteren Meta-Tabelle gespeichert sein kann. Das Einfügen und Löschen von Anwendungen oder Tabellen soll durch entsprechende Bindungen ermöglicht werden.

Die Visualisierung MetaSelect wurde als Beispiel bereits in Kapitel 3 verwendet und ist in Abb. 3.13(b) auf Seite 90 dargestellt. Die Menge Tables wurde dort mit einem vertikal rollbaren Viewport versehen, der so hoch ist, daß in ihm acht Tabellen gleichzeitig sichtbar sind. Dieser Viewport fixiert die variable Höhe der Menge Tables, so daß auch die Application-Tupel eine feste Höhe haben und die Applications-Menge mit clnForm dargestellt werden kann. Die Application-Tupel werden mit `tplLabel` dargestellt, wobei die Gruppierung explizit so gewählt ist, daß Name und Beschreibung der Anwendung die erste Gruppe bilden und die Tabellen als Einergruppe daneben dargestellt werden[2].

Eine mit MetaSelect generierte Repräsentation zeigt Abb. 3.14(b) auf Seite 91. Dort ist die Systemanwendung namens „.system“ sichtbar; das Boot-Schema, die Meta-Tabelle Applications und ihr Schema sind unter den Namen „.boot.scm“, „.meta.tbl“ und „.meta.scm“ aufgeführt. Alle anderen sichtbaren Tabellen werden durch das Visualisierungsschema strukturiert, das in der Systemanwendung unter dem Namen „visTree“ bekannt ist; bei diesen Tabellen handelt es sich also um Visualisierungen, unter ihnen befindet sich auch MetaSelect (Name „MetaSelect“).

Auf das Schema der Tabellen wird jeweils mit einem Link (Attribut Schema) verwiesen, so daß durch eine entsprechende Parametrisierung von atom (`script=link`, `path=Name`) der

[1] Die Meta-Tabelle Applications wird als Tabelle einer Anwendung also durch ein Table-Tupel in sich selbst repräsentiert, was wiederum ein Beispiel für die Selbstreferenzierung ist.

[2] Da der Viewport die Höhe der Menge Tables fixiert, würden per Default alle drei Komponenten in einer Gruppe untereinander dargestellt.

```
 1  proc schemaLink_get {id tf} {
 2      if [$tf is -null] {
 3          # table without schema?  Can't be, but...
 4          return "(none)"
 5      }
 6      set lf [$tf link -newfinger]
 7      # if this link target's schema link has himself as target,...
 8      if ![string compare [$lf link] [$lf get Schema]] {
 9          # ... it must be the boot schema, and thus the source
10          # must be a schema
11          set res "(schema)"
12      } else {
13          # this link target is an 'ordinary' schema, and thus the
14          # source must be a table - return its schema's (i.e. this
15          # link target's) name
16          set res [$lf get Name]
17      }
18      finger free $lf
19      return $res
20  }
```

Anhand des Schemaverweises kann festgestellt werden, ob eine Tabelle ein Schema repräsentiert: Dies ist dann der Fall, wenn das Schema das Boot-Schema ist. Das Boot-Schema ist die einzige Tabelle, die sich selbst als Schema strukturiert. Das Link-Ziel des Schemaverweises ist für das Boot-Schema also das Boot-Schema selbst. Dieser Test wird in Zeile 8 als Vergleich zweier Strings durchgeführt: Die Variable `lf` enthält die FID des Fingers, der in Zeile 6 generiert und auf dem Link-Ziel positioniert wurde (auf der Tabelle, die das Schema von `$tf` repräsentiert), die Option `link` des *fid*-Kommandos gibt die Position von `$lf` als Link-Ziel zurück, und die Option `get` liefert das Link-Ziel des Schema-Links von `$lf`. Ist der Test erfolgreich[a], zeigt `$tf` auf ein Schema und der String „(schema)“ ist das Ergebnis; ansonsten liefert die Option `get` in Zeile 16 den Namen des Schemas, auf das `$lf` zeigt, als Ergebnis.

[a] Hier tritt die Eigenschaft der „schwachen Typisierung“ von Skriptsprachen wie Tcl positiv in Erscheinung, die auch Ousterhout hervorhebt („[...] emphasis on compilation and strong typing makes it difficult [...] to accommodate the tremendous variety and rapid evolution [...]“ [Ous99]): der Vergleich der beiden Link-Werte ist unabhängig von ihrer Repräsentation.

Skript 6.1: Tcl/DB-Skript `schemaLink_get`

Schemaname anstelle des Links angezeigt werden könnte. Diese Lösung würde aber für jede Tabelle, die ein Schema repräsentiert, den Schemanamen „.boot.scm“ anzeigen und nicht wie beabsichtigt den String „(schema)“. Daher wird für die Darstellung des Schemanamens ein spezielles Skript `schemaLink_get` implementiert (siehe Skript 6.1 auf der vorherigen Seite), das durch die Parametrisierung `script=schemaLink` von atom zur Darstellung des Schemanamens verwendet wird[3].

Die Navigation wird eingeschränkt, indem die Bindungen der Visualisierungsfunktion wie in Tabelle 6.1 auf der gegenüberliegenden Seite dargestellt überschrieben werden. Dort sind auch weitere Bindungen überschrieben bzw. ergänzt worden, in denen spezielle Skripten zum Anlegen und Löschen von Applikationen und Tabellen, zum Generieren von Default-Visualisierungen und zur Repräsentation von Tabellen aufgerufen werden.

Beginnt das Bindungsskript mit einem „+“, wird es in Tcl/Tk-Manier an die vorhandene Default-Bindung angefügt. Dadurch wird z. B. beim Anlegen einer neuen Applikation durch die Default-Bindung für `<Insert>` (`%F Insert after`) zunächst ein neues Application-Tupel angelegt und der aktive Finger auf dieses Tupel bewegt, so daß dann vom Skript `MetaSelect_createAppl`, dem ein auf eben jenes Tupel zeigender Finger durch die Substitution der Sequenz „`%D`“ übergeben wird, die nullwertigen Komponenten des Tupels konkretisiert werden können.

Mit den Bindungen werden z. T. auch von den Default-Werten abweichende Menüeinträge für Pop-Up-Menüs angegeben. Bindungsskripte, deren zugehörige Menüeinträge mit Punkten enden, leiten einen modalen Dialog ein; das sind hier die Skripte `MetaSelect_createAppl` und `MetaSelect_createTbl` zum Anlegen neuer Applikationen bzw. Tabellen, für die noch weitere Informationen abgefragt werden müssen.

Auf die Implementierung der einzelnen Skripte (`MetaSelect_createAppl`, `MetaSelect_deleteAppl`, `MetaSelect_visualize`, `MetaSelect_createDefVis`, `MetaSelect_createTbl` und `MetaSelect_deleteTbl`) wird hier nicht detailliert eingegangen.

Erweiterungen, die zusätzlich in MetaSelect integriert werden könnten, sind z. B.:

- Die Änderung von Anwendungsnamen und -beschreibungen sowie Tabellennamen könnte an spezielle Ereignisse gebunden werden.

- Für die Mengen der Anwendungen und Tabellen könnten Filtermöglichkeiten durch Redefinition der First- und Iterate- sowie der Move-Methoden implementiert werden. Diese könnten z. B. nach Tabellentyp (Tabellen/Schemata) oder Tabellenname (Suchmuster, Systemtabellen[4], usw.) selektieren.

[3] Das Skript `schemaLink_edit` kann unimplementiert bleiben, da der Finger nicht auf das Attribut Schema zeigen und damit auch nicht dort hinein bewegt werden kann.

[4] Systemnamen beginnen mit einem Punkt (analog den „Dot-Files“ im UNIX-Umfeld) und könnten so per Default ausgeblendet werden.

Knoten		
Ereignis	Menü	Tcl-Skript
Tupel Application		
<Enter>	In	{ %F Push; %F Move last; %F Push }
<Escape>	—	{ }
<Insert>	New...	+{ MetaSelect_createAppl %D }
<Shift-Insert>	—	+{ MetaSelect_createAppl %D }
<Delete>	Delete	{ MetaSelect_deleteAppl %D; %F Delete current }
<Backspace>	—	{ %F Move back; MetaSelect_deleteAppl %D; %F Delete current }
Tupel Table		
<Enter>	Visualize	{ MetaSelect_visualize %D }
<Escape>	Out	{ %F Pop; %F Pop }
<Control-v>	Create Def.Vis.	{ MetaSelect_createDefVis %D }
<Insert>	New...	+{ MetaSelect_createTbl %D }
<Shift-Insert>	—	+{ MetaSelect_createTbl %D }
<Delete>	Delete	{ MetaSelect_deleteTbl %D; %F Delete current }
<Backspace>	—	{ %F Move back; MetaSelect_deleteTbl %D; %F Delete current }

Tabelle 6.1: Bindungen für die Visualisierung MetaSelect

6.1.2 Meta-Visualisierung

Zunächst werden zwei Visualisierungen zur Darstellung der Visualisierungsfunktionen- bzw. Visualisierungsklassen-Tabelle definiert. Dann wird eine Visualisierung für das Visualisierungsschema definiert und mit Bindungen versehen, so daß mit ihm generierte Repräsentationen für die Verfeinerung von Visualisierungen gemäß der Anforderungen aus Abschnitt 3.8.1 auf Seite 112 geeignet sind.

Visualisierungsfunktionen

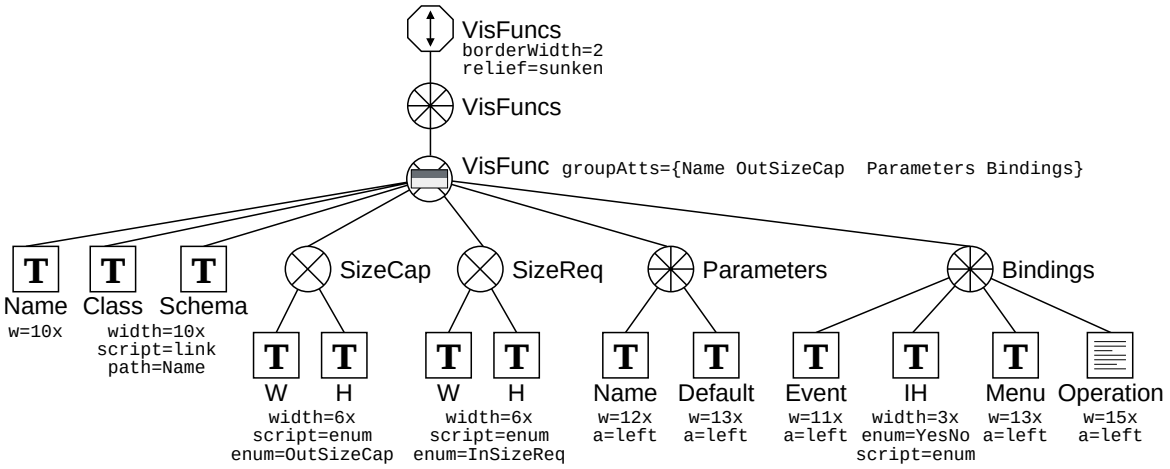


Abbildung 6.1: Visualisierung für die Visualisierungsfunktionen-Tabelle
(Die Attribute `width` und `align` sind z. T. mit `w` bzw. `a` abgekürzt.)

Die Visualisierungsfunktionen-Tabelle wurde bereits in Kapitel 5 bei der Besprechung der einzelnen Visualisierungsfunktionen ausgiebig dargestellt, z. B. in Abb. 5.16 auf Seite 213. Hier wird noch die zugehörige Visualisierung „nachgeliefert“ (siehe Abb. 6.1) und eine Repräsentation gezeigt, bei der der Schemateil mit einer anderen Schemavisualisierung generiert wurde (siehe Abb. 6.2 auf der gegenüberliegenden Seite).

Die Komponenten der VisFunc-Tupel werden in vier Gruppen aufgeteilt von tplLabel dargestellt:

1. Name der Visualisierungsfunktion, Name der zugehörigen Visualisierungsklasse und Name der Schemavisualisierungsfunktion (die Darstellung der Links Class und Schema ist jeweils so parametrisiert, daß relativ zum Link-Ziel das Attribut Name angezeigt wird: `script=link` und `path=Name`),

2. Längeneigenschaften- und -anforderungen,

3. definierte Parameter mit Default-Werten,

4. überschriebene Bindungen der Visualisierungsklasse und zusätzliche Bindungen.

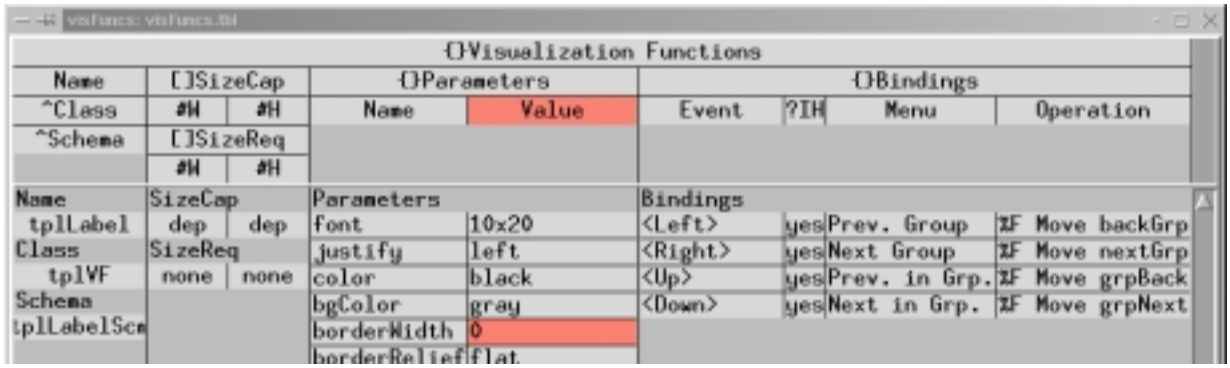


Abbildung 6.2: Repräsentation der Visualisierungsfunktionen-Tabelle (Ausschnitt)

Visualisierungsklassen

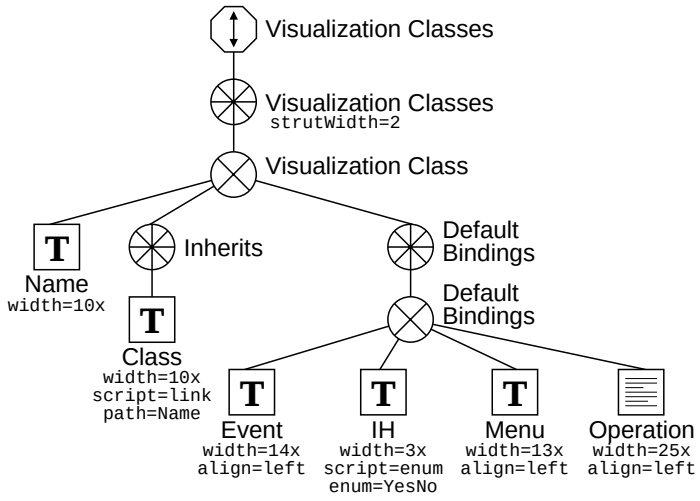


Abbildung 6.3: Visualisierung für die Visualisierungsklassen-Tabelle

Die Visualisierung für die Visualisierungsklassen-Tabelle entspricht weitestgehend der Default-Visualisierung (siehe Abb. 6.3). Folgende Veränderungen wurden vorgenommen:

- angepaßte Attributbreiten und Textausrichtungen,
- für die Links auf vererbende Klassen (Element Class der Menge Inherits) wird mittels der `link`-Skripten mit dem Pfad `Name` jeweils der Klassenname angezeigt,
- das Vererbungs-Flag (Attribut Inherit des Tupels Default Binding) hat in der Darstellung den „Sub“ und wird mit den `enum`-Skripten als Aufzählungstyp YesNo dargestellt,
- die horizontalen Streben zwischen den Elementen der Menge Visualization Classes sind etwas dicker als alle anderen Linien (`strutWidth=2`), und

- die Bindungs-Skripten werden mit `text` dargestellt.

Eine mit dieser Visualisierung generierte Repräsentation der Visualisierungsklassen-Tabelle ist in Abb. 6.4 dargestellt.

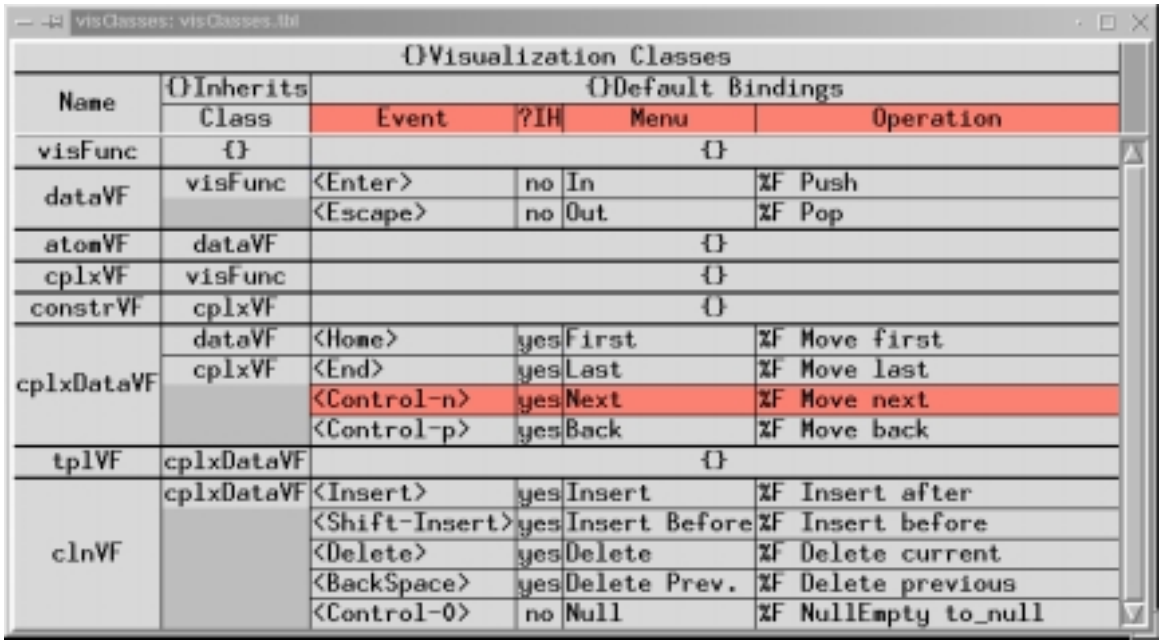


Abbildung 6.4: Repräsentation der Visualisierungsklassen-Tabelle

Visualisierungsschema

Die Visualisierung des Visualisierungsschemas ist in Abb. 6.5 auf der gegenüberliegenden Seite dargestellt. Diese Visualisierung ist die eigentliche „Meta-Visualisierung“, denn sie definiert die Repräsentation von Visualisierungen, inklusive sich selbst (auch hier ein Beispiel der Selbstreferenzierung). Die Visualisierung ist in Abb. 6.6 auf Seite 236 als durch sich selbst definierte Repräsentation dargestellt Ⓐ; in der Abbildung ist außerdem eine duale, graphische Repräsentation der Visualisierung dargestellt Ⓑ.

Die Links des Visualisierungsschemas sind in der Visualisierung jeweils so parametrisiert, daß sie den Namen des Link-Ziels mit den `link`-Skripten der Visualisierungsfunktion atom darstellen. Auf äußerster Ebene bilden die beiden Links Schema und Root sowie das Tupel Size eine Gruppe der tplLabel-Visualisierungsfunktion, werden also untereinander dargestellt. Die zweite Gruppe ist Menge der Visualisierungsknoten einer Visualisierung, von denen jeweils drei in einem vertikal rollbaren Viewport dargestellt werden. Die Gruppenbildung muß hier explizit über den Parameter `groupAtts` erfolgen, da der Visualisierungsknoten Nodes eine feste Höhe hat.

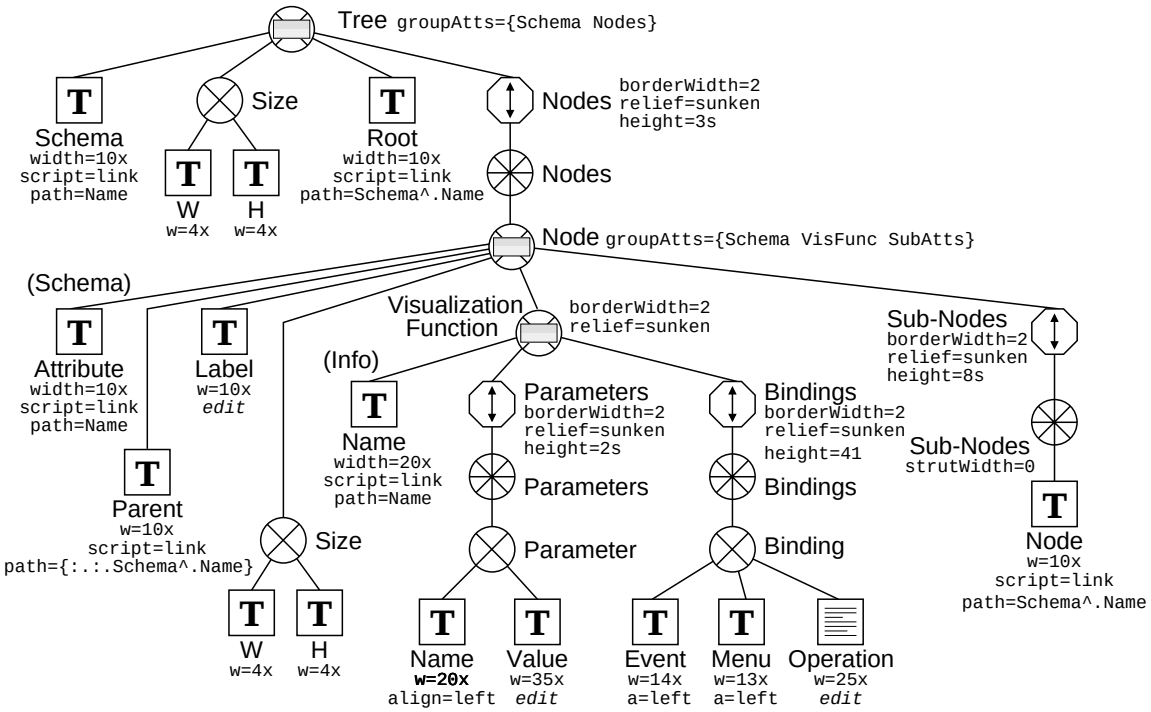


Abbildung 6.5: Visualisierung des Visualisierungsschemas
(Die Attribute `width` und `align` sind z. T. mit `w` bzw. `a` abgekürzt,
`edit` steht für `align=left`, `border=2` und `relief=sunken`.)

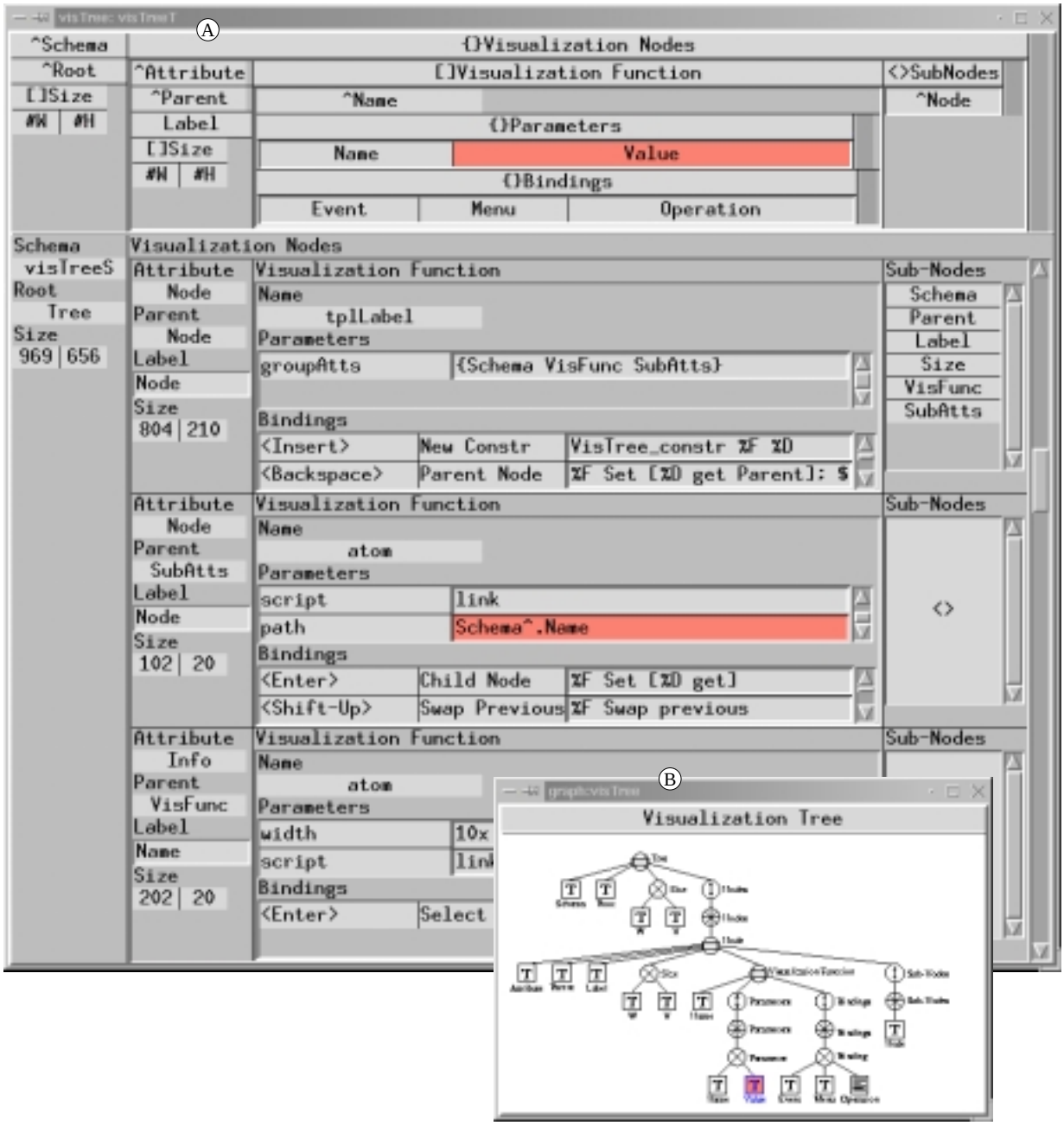


Abbildung 6.6: Repräsentationen der Meta-Visualisierung

Die Komponenten der Node-Tupel werden in drei Gruppen aufgeteilt ebenfalls mit tplLabel dargestellt, wobei die erste Gruppe aus dem (Schema-) Attributnamen, dem Attributnamen des Vorgängerknotens, dem (Visualisierungs-) Attributnamen und den Längeninformationen, die zweite aus den Angaben zur Visualisierungsfunktion und die dritte aus der Liste der Nachfolgerknoten besteht.

Die Angaben zur Visualisierungsfunktion werden als Tupelkomponenten in einer Gruppe der tplLabel-Visualisierungsfunktion dargestellt, damit die beiden recht breiten Mengen der Parameter- und Bindungsspezifikationen untereinander erscheinen. Diese beiden Mengen sind von Viewports umgeben, damit sie feste Höhen haben. Ein Parameter-Tupel hat nur Strings als Komponenten, also feste Höhe; der entsprechende Viewport hat die Höhe von zwei Parameter-Tupeln. Für die Komponente Operation des Binding-Tupels ist die variabel hohe atomare Visualisierungsfunktion `text` eingetragen. Daher kann die Höhe des zugehörigen Viewports nicht relativ zur Höhe der Sub-Sub-Visualisierungsfunktion angegeben werden, sondern ist in Pixeln (`height=41`) spezifiziert[5].

Insgesamt ist die mittlere Gruppe der Knoten ungefähr neun Zeilen hoch: Vier Label, ein atomarer Wert und zwei Viewports zu je zwei Zeilen. Daher wird der Viewport der Nachfolgerknoten-Liste so parametrisiert, daß er so hoch wie acht (einzeilige) Node-Links ist (`height=8s`); zusammen mit dem Label ist also auch die rechte Gruppe ungefähr neun Zeilen hoch.

Die Viewports sind mit einer Randbreite von zwei Pixeln und einem versunken erscheinenden Relief dargestellt, um sie von ihrer Umgebung etwas hervorzuheben. Da die Attribute Value und Operation der Parameter und Bindungen die am häufigsten geänderten Attribute der Visualisierungen sind, werden sie genauso dargestellt.

In Tabelle 6.2 auf Seite 239 sind die Bindungen dargestellt, mit denen die Visualisierung die Anforderungen aus Abschnitt 3.8.1 auf Seite 112 zur Verfeinerung von Visualisierungen wie folgt erfüllt:

Navigation in der Visualisierung: Zeigt der aktive Finger auf einen Node-Link, kann er mit `<Enter>` auf dem Ziel-Tupel positioniert werden. Zeigt er auf einen Parent-Link, kann er mit `<Enter>` auf dem Element einer SubNodes-Liste positioniert werden, das auf das zum Parent-Link gehörende Node-Tupel verweist (repräsentiert das Node-Tupel den Wurzelknoten, wird der Finger auf dem Root-Link positioniert). Zeigt der Finger auf ein Node-Tupel, kann er mit `<Backspace>` auf dem den Vorgängerknoten repräsentierenden Node-Tupel positioniert werden.

Einfügen und Löschen von Konstruktions-Visualisierungsknoten: Zeigt der aktive Finger auf ein Node-Tupel, kann mit `<Insert>` ein Konstruktions-Visualisierungsknoten eingefügt werden. Dieser wird immer an Stelle des aktiven Fingers eingefügt, was dem normalen Verhalten von `<Shift-Insert>` entspricht. Die konkrete Konstruktions-

[5]Da der Default-Zeichensatz `10x20` 20 Pixel hoch ist, reicht eine Viewport-Höhe von 41 Pixeln für die Darstellung von zwei Zeilen.

Visualisierungsfunktion wird über einen modalen Dialog bestimmt. Nur Konstruktions-Visualisierungsknoten können mit <Delete> auch wieder gelöscht werden; andere Knoten werden implizit durch Operationen auf Node-Links gelöscht.

Löschen und Umordnen der Nachfolgerknoten von Tupel-Visualisierungsknoten:
Zeigt der aktive Finger auf einen Node-Link eines Tupel-Visualisierungsknotens[6], können mit <Delete> und <Backspace> Nachfolgerknoten und damit deren Teilbäume gelöscht werden. Das Skript VisTree_delSub verhindert, daß der letzte Nachfolgerknoten gelöscht wird. Mit <Insert> und <Shift-Insert> können Tupelkomponenten auch wieder eingefügt werden, wobei das Skript VisTree_insSub einen modalen Dialog zur Selektion der einzufügende Komponente anzeigt und dann die Knoten der Default-Visualisierung generiert und in die Visualisierung einfügt.

Auswahl von Visualisierungsfunktionen: Zeigt der aktive Finger auf einen Info-Link, wird mit <Enter> ein modaler Dialog zur Bestimmung einer neue Visualisierungsfunktion aufgerufen. In diesem Dialog stehen nur Visualisierungsfunktionen der gleichen Klasse und spezielle Visualisierungsfunktionen zur Wahl, damit z.B. keine Tupel-Visualisierungsfunktion an einem atomaren Visualisierungsknoten eingetragen werden kann. Alle an dem Visualisierungsknoten angegebenen Parameter, die nicht für die neue Visualisierungsfunktion definiert sind, werden aus der Menge Parameters gelöscht. Wird eine Kollektions- oder Tupel-Visualisierungsfunktion durch eine spezielle Visualisierungsfunktion ersetzt, werden alle Knoten der SubNodes-Liste und deren Teilbäume gelöscht. Wird eine spezielle Visualisierungsfunktion ersetzt, werden die entsprechenden Knoten der Default-Visualisierung generiert und in die Visualisierung einfügt

Definition und Veränderung von Parametern und Bindungen: Das Einfügen in die Menge Parameters wird immer von einem modalen Dialog eingeleitet, der die Auswahl eines anwendbaren Parameters gestattet. Diese Auswahl wird aus der Parametermenge der Visualisierungsfunktion gebildet und um die bereits spezifizierten Parameter eingeschränkt. Bindungen werden für <Enter> auf der Menge Parameters sowie <Insert> und <Shift-Insert> auf dem Tupel Parameter definiert.

Ausführung der Längenberechnung: Zeigt der aktive Finger auf ein Node-Tupel, das Tree-Tupel oder ein Size-Tupel, kann mit <Control-s> die Längenberechnung ausgeführt werden. Auf dem Tree-Tupel oder dem Tupel Size, das eine Komponente von Tree ist, wird die Längenberechnung für den Wurzelknoten, also die gesamte Visualisierung, durchgeführt; die resultierenden Längenangaben werden in dem Size-Tupel eingetragen. Auf dem Size-Tupel eines Node-Tupels wird die Längenberechnung für den durch das Tupel repräsentierten Visualisierungsknoten bzw. dessen Teilbaum durchgeführt.

[6]Diese Einschränkung muß in den Bindungs-Skripten implementiert werden.

Knoten		
Ereignis	Menü	Tcl-Skript
Node-Link		
<Enter>	Child Node	{ %F Jump [%D get] }
<Shift-Up>	Swap Previous	{ %F Swap previous }
<Shift-Down>	Swap Next	{ %F Swap }
<Delete>	Delete Sub-Tree	{ VisTree_delSub %T; %F Delete current }
<Backspace>	Del. Prev. Sub-Tree	{ %F Move back; VisTree_delSub %T; %F Delete current }
<Insert>	Insert Sub-Tree	{ %F Insert after; VisTree_insSub %T }
<Shift-Insert>	Ins. Sub-Tree Bef.	{ %F Insert before; VisTree_insSub %T }
Parent-Link		
<Enter>	Source Link	{ %F Jump [%D get] }
Info-Link		
<Enter>	Select Vis.Func.	{ VisTree_selVF %T }
Node-Tupel		
<Backspace>	Parent Node	{ %F Jump [%D get Parent]; $F Pop; $F Pop }
<Insert>	New Constr...	+{ VisTree_constr %F %D }
<Shift-Insert>	—	+{ VisTree_constr %F %D }
<Control-s>	Calc. Sizes	{ VisTree_sizes %D }
Size-Tupel (Komponente des Node-Tupels)		
<Control-s>	Calc. Sizes	{ %D pop; VisTree_sizes %D; %D push Size }
Size-Tupel (Komponente des Tree-Tupels)		
<Control-s>	Calc. Sizes	{ set $f [%D link -newfinger :.Root] val_set %D [VisTree_sizes %f] finger free $f }
Tree-Tupel		
<Control-s>	Calc. Sizes	{ set $f [%D link -newfinger Root] %D push Size val_set %D [VisTree_sizes %f] %D pop; finger free $f }
Width, Height, Name (im Parameter-Tupel)		
<Enter>	—	{}

Tabelle 6.2: Bindungen für Knoten im Visualisierungsschema

Außerdem werden die Default-Bindungen der folgenden atomaren Attribute so verändert, daß das Editieren durch die *in*-Operation nicht mehr möglich ist: Width und Height in beiden Size-Tupeln sowie Name im Parameter-Tupel.

Die in Abb. 6.6 auf Seite 236 dargestellte duale Repräsentation Ⓑ wird durch entsprechende Bindungen in das Interaktionskonzept integriert, so daß in beiden Repräsentationen navigiert werden kann. Diese Bindungen werden hier nicht dargestellt.

Die Fingermethoden Jump und Swap, die oben in einigen Tcl-Skripten verwendet wurden, sind Erweiterung des Interaktionskonzeptes. Jump läßt einen Finger zu einem beliebigen Datenobjekt in einer Tabelle „springen“, Swap vertauscht zwei Elemente einer Kollektion (und deren Repräsentation).

6.2 Zusammenfassung

In der vorliegenden Arbeit wurde ein neues, erweiter- und konfigurierbares Visualisierungsverfahren zur Interaktion mit komplex strukturierten Datenobjekten vorgestellt. Die Erweiterbarkeit bezieht sich dabei auf das Verfahren selber, auf die vom Verfahren einsetzbaren Techniken der Visualisierung, die hier Visualisierungsfunktionen genannt werden, und auf die in das Verfahren integrierte Interaktion. Die mit dem Verfahren generierbaren Repräsentationen sind besonders zum Browsen in den Objekten und zum Editieren der Objekte geeignet, die typischerweise in objekt-relationalen Datenbanken gespeichert werden.

Insbesondere die Erweiterung um neue Techniken der Visualisierung für bestimmte Klassen von Datenobjekten ist anhand einer objekt-orientierten Modellierung der Konzepte klar strukturiert und in übersichtliche Teilaufgaben modularisiert.

Die generierten Repräsentationen können modulartig in vorhandene graphische Benutzerschnittstellen integriert werden oder als vollständige graphische Benutzerschnittstelle einer Anwendung eingesetzt werden; der letzte Fall wurde an zwei Beispielanwendungen exemplarisch demonstriert.

Objekt-relationale Datenbanksysteme sind, verglichen etwa mit den Betriebssystemen der UNIX-Familie, sehr große und weitgehend monolithische Softwaresysteme. Für UNIX sind um einen gemeinsamen „kleinen“ Kern herum eine Vielzahl austauschbarer Komponenten wie Kommandointerpreter (Shells), Werkzeuge und ganze Fenstersysteme verfügbar; demgegenüber sind bei den großen kommerziellen DBMSs Komponenten wie Anfrageverarbeitung, Transaktionsmanagement, Speicher- und Indexverwaltung sehr eng verzahnt und nicht austauschbar. Modularität und Orthogonalität eines Softwaresystems, also die sinnvolle Aufteilung in Funktionseinheiten und die Möglichkeit, Methoden einer Komponente auf andere Komponenten anzuwenden, sind aber die einzige Möglichkeit, der ständig wachsenden Komplexität Herr zu werden.

Wie in den Kapiteln 1 und 2 beschrieben wurde, hat der Übergang zu objekt-orientierten Entwicklungen im Bereich der Benutzerschnittstellen für Datenbanken wenig Abhilfe gebracht. Der Programmieren-Compilieren-Ausprobieren-Zyklus macht die Entwicklung sehr schwerfällig. Starke Typbindung und komplexe Hierarchien von Objektklassen schaffen starke Abhängigkeiten und immer unübersichtlichere und inflexiblere Systeme.

Ziel muß es sein, mehr Funktionalität mit weniger Komponenten zu erreichen. Für den Teilaspekt der Benutzerschnittstelle wurde dies in der vorliegenden Arbeit versucht. Visualisierungsvorschriften für Datenobjekte (Relationen, Tabellen) werden als Baum aus der Strukturdefinition (Schema) abgeleitet und als normales, persistentes Datenobjekt in der Datenbank gespeichert; sie werden kurz „Visualisierungen“ genannt.

Ist der Anwender mit der systemgenerierten Default-Repräsentation eines Datenobjektes zufrieden, muß er keine zusätzlichen Angaben machen. Andere Repräsentationen können durch interaktive Verfeinerung von Visualisierungen definiert werden, wobei veränderte Parameter, Bindungen, usw. automatisch in dem Datenobjekt, das die Visualisierung repräsentiert, gespeichert werden. Die Verfeinerung erfolgt aus Systemsicht mit den bereits vorhandenen Editier-Möglichkeiten.

Die Interaktion basiert auf einem Cursor-Prinzip, bei dem das zu bearbeitende atomare oder komplexe Datenobjekt graphisch hervorgehoben wird und dadurch auch beim Navigieren den Standort eines sog. „Fingers“ signalisiert. Die Rolle der Finger bei der Repräsentation von Datenbankinhalten wurde im Abschnitt 4.3 ausführlich besprochen.

Finger können mittels der Skriptsprache Tcl/DB manipuliert werden, die eine Erweiterung von Ousterhouts Tool Command Language Tcl ist und in Abschnitt 2.3.4 erläutert wurde. Visualisierungsfunktionen und ihre Methoden werden durch Tcl/DB-Skripte im objektorientierten Stil realisiert. Sie werden zur Generierung von Repräsentationen durch das Visualisierungsverfahren oder auch ereignisgesteuert aufgerufen. Werden Methodenaufrufe durch Ereignisse (z. B. Tastatur- oder Mausereignisse) ausgelöst, spricht man in der für graphische Benutzerschnittstellen üblichen, ereignisorientierten und asynchronen Ablaufwelt von Bindungen (vgl. Abschnitt 2.2). Tcl/DB-Skripte stellen demnach das aktive Element im Sinne von Stonebrakers *„procedure as a data type“* dar.

Visualisierungen sind wie bereits angedeutet „normale“ Datenobjekte, die die visuelle Repräsentation anderer Datenobjekte definieren. Eine Visualisierung paßt zu einem Datenobjekt, da sie aus dem Schema abgeleitet wird, das die Struktur des Datenobjektes definiert. Auch Schemata können als „normale“ Datenobjekte angesehen werden, deren Struktur durch ein Meta-Schema definiert ist, das hier „Boot-Schema“ genannt wird. Daher können visuelle Repräsentationen für Schemata definiert werden, indem Visualisierungen aus dem Boot-Schema abgeleitet werden, die dann Schemavisualisierungen heißen. Dieser Zusammenhang zwischen Datenobjekten (Tabellen), Schemata, dem Boot-Schema, Visualisierungen und Schemavisualisierungen ist in Abb. 6.7 auf der nächsten Seite im mittleren Teil dargestellt.

Angedeutet wird dort auch, daß ein Schema mehrere Tabellen definieren kann. Die explizite Trennung von Schema und Tabelle ist eine Besonderheit in ESCHER und an die Trennung

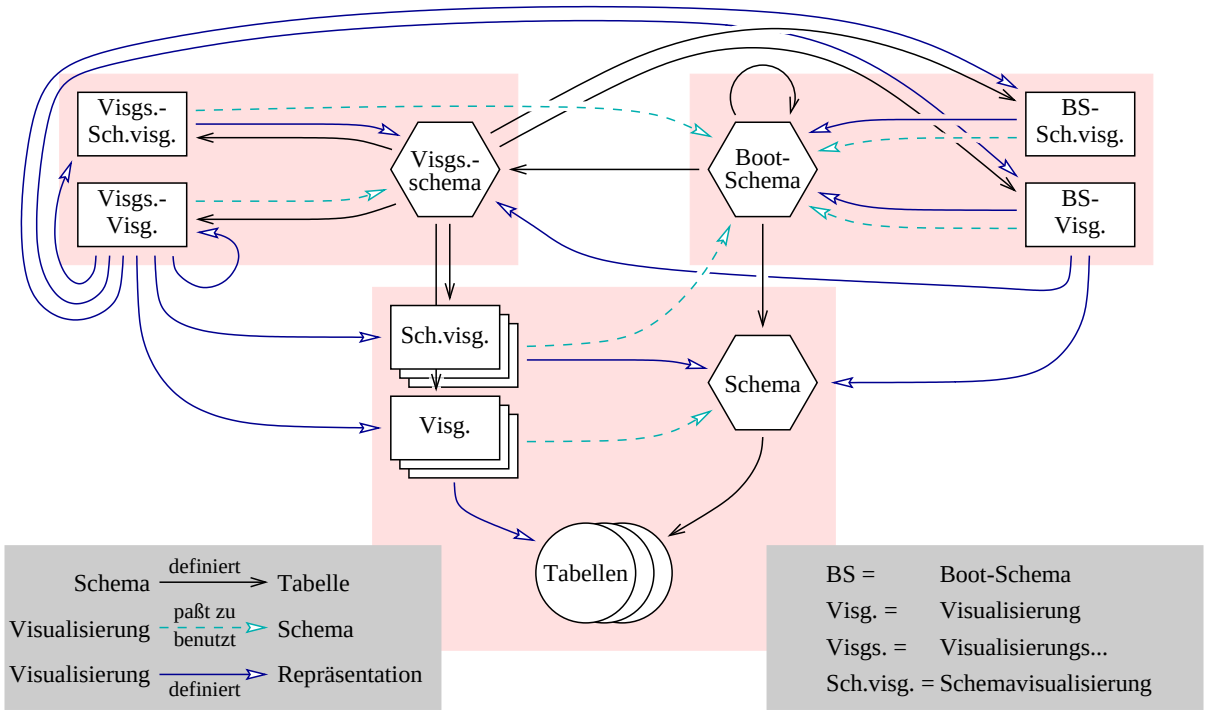


Abbildung 6.7: Schemata, Tabellen, und Visualisierungen als Meta-Objekte

von Inhalt und Layout in Textverarbeitungssystemen angelehnt. Natürlich kann es auch mehrere Visualisierungen zu einer Tabelle geben, die unterschiedliche visuelle Repräsentationen definieren (z. B. eine graphische, eine tabellarische und eine formularorientierte Version).

Weil Visualisierungen Datenobjekte sind, können sie in ESCHER persistent gespeichert werden. Hierfür müssen sie dem System durch strukturelle Definitionen (Schemata) bekannt sein. Ein wesentliches Ergebnis dieser Arbeit ist, daß dazu nur **ein** Schema nötig ist, das sog. „Visualisierungsschema“. Das bedeutet nicht nur, daß das Schema unabhängig von speziellen Formen der visuellen Repräsentation ist, also z. B. formularorientierte Visualisierungen sich damit genauso angeben lassen wie tabellarische. Vielmehr haben damit auch Schemavisualisierungen dieselbe Struktur wie „normale“ Visualisierungen, was den Umgang erheblich vereinfacht und der obigen Forderung nach „mehr Funktionalität mit weniger Komponenten“ entspricht.

Visualisierungen haben aber nicht nur ein Schema, sie haben im Sinne einer möglichst weitgehenden Orthogonalität auch selbst wieder eine Visualisierung. Diese „Visualisierungs-Visualisierung“ ist in Abb. 6.7 links oben dargestellt, zusammen mit der „Visualisierungs-Schemavisualisierung“, der Visualisierung des Visualisierungsschemas. Das bedeutet, daß visuelle Repräsentationen von Visualisierungen als Datenobjekten mit ihren eigenen Mitteln definiert und generiert werde können, und beinhaltet auch die Interaktion mit diesen Datenobjekten, d. h. die bereits angesprochene Verfeinerung von Visualisierungen.

Die Visualisierungs-Visualisierung ist also eine Meta-Visualisierung im gleichen Sinn, wie das Boot-Schema als Schema aller Schemata, inklusive sich selbst, ein Meta-Schema ist, was sich durch die Schleifen in der Abbildung ausdrückt. Abb. 6.6 auf Seite 236 zeigt die Visualisierungs-Visualisierung in der durch sich selbst definierten Repräsentation.

Der letzte noch unerwähnte Teil in Abb. 6.7 ist die rechts oben dargestellte Visualisierung von Schemata als durch das Boot-Schema strukturierte Datenobjekte. Hier sind sowohl Visualisierung als auch Schemavisualisierung aus dem Boot-Schema abgeleitet; die „Boot-Schema-Visualisierung“ definiert die visuelle Repräsentation jedes Schemas, die Boot-Schema-Schemavisualisierung definiert die dazu passende visuelle Repräsentation des Boot-Schemas als Strukturinformation.

Aus dem oben gesagten folgt, daß die sechs beschriebenen Meta-Objekte (Schemata und Visualisierungen) die notwendige und hinreichende Anzahl und Ausprägung zur Definition und visuellen Repräsentation beliebiger Anwendungs-Objekte (Schemata und durch sie definierte Tabellen) darstellt, inklusive ihrer eigenen Schemata und Visualisierungen.

Formal läßt sich dies daraus ableiten, daß in jedes Schema zwei „definierende Visualisierungspfeile“ $\twoheadrightarrow$ hineingehen (Visualisierung und Schemavisualisierung), in jede Visualisierung genau ein Visualisierungspfeil hineingeht und aus jeder Visualisierung genau ein „Schemabenutzungspfeil“ $\dashrightarrow$ herausgeht. Ferner geht in jedes Schema mindestens ein Schemabenutzungspfeil hinein. Somit kann bei einer vollständig orthogonalen Darstellung auf keines der Objekte verzichtet werden und keines ist redundant.

Aus dem „Dreiklang“ von Tabelle, Schema und Meta-Schema ist nun ein „Zehnklang“ geworden. Das klingt zunächst aufwendiger (dissonanter) als es tatsächlich der Fall ist. Trotzdem ist die Frage gerechtfertigt, ob diese Methode der Selbstreferenzierung mit Meta-Objekten die richtige ist, wenn man das am Anfang dieses Abschnitts formuliert Ziel „Reduzierung der Komplexität“ im Auge behalten will. Geht man den Weg weiter, kommt man z. B. zu Meta-Objekten für Transaktionen, d. h. einer Sperrtabelle, die alle Sperren verwaltet, einschließlich der Sperren dieser Sperrtabelle selber, usw.

Interessant ist dieser Aspekt in Verbindung mit ver- bzw. geteilten Repräsentationen, d. h. der Verwendung von Visualisierungen im Mehrbenutzerbetrieb, wo inkonsistente Repräsentationen durch Sperren auf Visualisierungen und die transaktionsgeschützte Verfeinerung von Visualisierungen vermieden werden müssen. Stellt man sich vor, daß Visualisierungen zur Repräsentation von Daten der Flugsicherung benutzt werden, dann kann eine Inkonsistenz aufgrund eines Visualisierungsexperiments fatale Folgen haben[7].

Die Diskussion um Open Source Software hat z. B. im Zusammenhang mit Linux gezeigt, daß die Verfügbarkeit von Quellcode (siehe Abschnitt 6.3) und die damit verbundene Inspizierbarkeit aller Parameter tendenziell zu mehr Sicherheit führt. In diesem Sinn sind auch

[7] Hier drängt sich eine Ähnlichkeit zur Reaktorkatastrophe in Tschernobyl auf, wo eine Störung des Reaktors im laufenden Betrieb simuliert werden sollte und sich durch fehlerhafte Bedienung zu einem tödlichen „Run-away“ entwickelte.

die obigen Selbstreferenzierungen ein Sicherheitsfaktor. Weiterhin kann eine erfolgreiche Selbstanwendung als Test mit Extremcharakter gewertet werden.

Jenseits dieser mehr spekulativen Beurteilung der Komplexität und der Vor- und Nachteile der vorgestellten Lösung kann man objektiv meßbar feststellen, daß sich der Umfang des Quellcodes der Implementierung von Tk-ESCHER gegenüber der von X-ESCHER wesentlich verkleinert hat, selbst wenn man Faktoren wie verwendete Bibliotheken (Motif vs. Tcl/Tk) und unterschiedliche Programmierumgebungen (Systemprogrammier- vs. Skript-Sprache) in betracht zieht.

Auch die Laufzeiten des Systems haben sich nur wenig verändert, obwohl von einer übersetzenden zu einer interpretierenden Umgebung übergegangen wurde. Durch neuere Entwicklungen, z. B. den seit Version 8.0 in Tcl integrierten *on-the-fly bytecode compiler*, sind weitere Laufzeitverbesserungen zu erwarten. Verläßlichere Aussagen ließen sich allerdings erst nach genauen Messungen und nach möglichen Optimierungen machen, womit diese Zusammenfassung in den Ausblick auf zukünftige Aufgaben übergeht.

6.3 Ausblick

In dieser Arbeit wurde verschiedentlich auf Teilaspekte von und Beziehungen zu verwandten Themen hingewiesen, auf die im Rahmen einer einzelnen Dissertation nicht weiter eingegangen werden kann. Die wichtigsten Anknüpfungspunkte sollen hier nochmals kurz angesprochen werden.

Variante Typen (siehe Abschnitt 2.3.1) können als Spezialform der Tupel-Konstruktoren angesehen werden. Ein varianter Typ definiert verschiedene, alternative Interpretationen eines Datenobjektes. Analog muß deren Visualisierung auch verschiedene Repräsentationen definieren, von denen aber nur jeweils eine dargestellt wird. Ihre Integration in eine Repräsentationen ist sicherlich problematisch, da die Strukturen der einzelnen Varianten sehr verschieden sein können und dadurch auch ihre Darstellung sehr heterogen sein kann. In das Visualisierungsverfahren können sie integriert werden, indem für jede Variante ein Nachfolgerknoten (bzw. Teilbaum) in der Visualisierung angelegt wird. Die Variantenauswahl kann über geeignete Visualisierungsfunktionen für variante Typen oder durch einen zusätzlichen Nachfolgerknoten erfolgen, der das Auswahlkriterium repräsentiert. Als Form der Variantenauswahl sind z. B. Karteireiter, Combo-Boxen oder Pop-Up-Menüs denkbar.

Die Auswahl der darzustellenden Variante wird entweder als Erweiterung in den Methoden der Visualisierungsknoten oder in geeigneten Visualisierungsfunktionen implementiert.

Problematisch ist auch die Darstellung von Schemainformationen, denn verschiedene Instanzen eines varianten Typs können unterschiedliche Varianten darstellen —

welche Struktur soll dann im Schema dargestellt werden? Denkbar wäre z. B. die Schemainformation in die Repräsentation des Datenobjektes zu integrieren, so daß jede Instanz eines varianten Typs seine „eigene“ Schemarepräsentation hat. In [WPC92] wurde vorgeschlagen, daß der aktive Finger die dargestellte Struktur bestimmt.

Nest und Unnest-Operationen [TF86] (siehe Abschnitte 3.2.1 und 4.3) entsprechen einer Umstrukturierung des Schemabaumes und erfordern daher eine Umstrukturierung der Visualisierung. Dabei können die Visualisierungsknoten der Attribute, die bei einer Nest-Operation im Elementtyp einer Kollektion zusammengefaßt werden, als Nachfolgerknoten des einzufügenden Kollektions-Visualisierungsknotens verwendet werden. Umgekehrt können bei einer Unnest-Operation die Nachfolgerknoten eines zu entfernenden Kollektions-Visualisierungsknotens in den umgebenden Visualisierungsknoten integriert werden.

Ob Nest und Unnest-Operationen durch Visualisierungen simuliert werden können (z. B. durch „virtuelle“ Kollektions-Visualisierungsfunktionen) oder nur eine tatsächlich veränderte Strukturdefinition reflektieren können, ist noch zu untersuchen.

Weitere Visualisierungsfunktionen müssen implementiert und evaluiert werden, um die Erweiterbarkeit des vorgestellten Visualisierungskonzeptes zu überprüfen und ggf. zu verbessern. Diese könnten z. B. Informationsvisualisierungen im Sinne von [CNJ+98] sein oder formularartige Tupel-Visualisierungsfunktionen mit frei plazierbaren Repräsentationen von Komponenten darstellen. Auch durch die Realisierung von weiteren Anwendungsbeispielen wird sich die Notwendigkeit neuer Visualisierungsfunktionen ergeben.

Dynamische strukturelle Änderungen von Visualisierungen durch Benutzeroperationen zur Laufzeit (siehe Abschnitt 4.3) erfordern neben der Veränderung der Visualisierung (3.6.2) das Generieren und oder Zerstören von Repräsentationen, wobei ggf. vorhandene Repräsentationen von Sub-Objekten integriert bzw. vor der Zerstörung bewahrt werden müssen. Dazu ist eine Erweiterung sowohl des Visualisierungsverfahrens als auch der bestehenden Visualisierungsfunktionen notwendig, auf die hier nicht weiter eingegangen wird.

WWW-Umgebungen schränken die Möglichkeiten zur Darstellung komplexer Datenobjekte ein. Zwar können mit Hilfe des `TABLE`- und verwandter Tags, die für HTML seit der Version 3.0 standardisiert sind, auch geschachtelte Tabellen dargestellt werden, mehr Freiheiten bieten aber Lösungen, die auf speziellen Plug-ins aufsetzen, z. B. für Java oder Tcl/Tk. Insbesondere die letzte Variante ist für das in dieser Arbeit entwickelte Verfahren interessant, da die Konzepte mit Hilfe von Tcl/DB evaluiert und prototypisch realisiert wurden. Zu beachten ist, daß der beim Client-Rechner ausgeführte Tcl-Code keine Tcl/DB-Anweisungen enthalten darf, da diese Erweiterung des Tcl-Interpreters im Tcl-Plug-in nicht zur Verfügung steht. Erfahrungen aus anderen Forschungsaktivitäten des ESCHER-Projektes [TWW99] zeigen, daß trotzdem

Repräsentationen komplexer Objekte für verteilte Anwendungen im WWW realisiert werden können.

CSCW ist ein hochaktuelles und gegenwärtig stark bearbeitetes Forschungsgebiet. Das Fingerkonzept hat sich als geeignete Basis für kooperatives Arbeiten erwiesen [GWZ, ATW]. Auch die Aussage von Nicol et al., daß Benutzerschnittstellen für kooperatives Arbeiten im WWW durch komponentenbasierte Technologien erstellt werden können [NGP+99], läßt zusammen mit dem vorhergehenden Abschnitt das vorgestellten Visualisierungsverfahren hier besonders interessant erscheinen.

Interaktion mit der Maus ist in dieser Arbeit eher „stiefmütterlich" behandelt worden. Konzepte wie die interaktive Selektion von Objekten und Sub-Objekten mit *drag & drop* können durch den Einsatz von Behälter-Widgets (siehe Abschnitt 4.1.1) mittels Bindungen für Ereignisse des Fenstersystems realisiert werden.

Ergonomische Untersuchungen mit dem Ziel einer qualitativen Beurteilung des entwickelten Visualisierungsverfahrens als Schnittstelle zwischen Mensch und Maschine waren nicht das Ziel dieser Arbeit. Das vorgestellte Visualisierungsverfahren ist vielmehr ein Mittel, das die Untersuchung verschiedener Visualisierungsalternativen ermöglicht, und so einer vergleichenden Beurteilung durch Ergonomiefachleute im Hinblick auf Anforderungen der HCI zugänglich macht.

Weitere Themen sind Animationen durch visualisierte Finger in Tcl/DB-Skripten, datenbankspezifische Konzepte wie z. B. Transaktionen und die damit verbundenen Sperrmechanismen, *cut & paste* Operationen, datengesteuerte Beeinflussung von Repräsentationen („stelle alle Repräsentationen von Sub-Objekten mit roter Schrift dar, falls das Attribut Bestand einen Wert kleiner zehn hat"), usw.

Das hier vorgestellte Visualisierungsverfahren ist mittels Tcl/Tk prototypisch realisiert worden. Es ist, wie X-ESCHER und die Erweiterung Tcl/DB für den skriptbasierten Zugang zu ESCHER, in der Gesamtheit im Quellcode öffentlich zugänglich und (im Rahmen der mit dem Quellcode von Tcl selbst verbundenen Lizenz, die dort in der Datei `license.terms` nachlesbar ist) frei verfügbar. Damit verbindet der Autor dieser Arbeit das Angebot an interessierte Leser, die hier vorgestellten, erweiterbaren Methoden für ihre eigenen Aufgaben einzusetzen.

Das in der Einleitung beschworene, angebrochene Zeitalter der interaktiven, digitalen, visuellen Kommunikation wird dabei nicht nur nach Antworten auf die oben gerade aufgezählten offenen Fragen verlangen; es wird auch weitere, innovative Anwendungsfelder eröffnen, die jetzt noch nicht absehbar sind. Die intuitive, anpaßbare visuelle Interaktion mit diesen Datenwelten, gerade auch für den ungeübten Anwender, bleibt eine zentrale Herausforderung. Der Autor hofft, daß die vorliegende Arbeit dazu eine methodische und exemplarische Hilfestellung geliefert hat.

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Beispiele

Literaturverzeichnis

[AB84] Abiteboul, S. and N. Bidoit: *Non first normal form relations to represent hierarchical organized data.* In *Proc. PODS'84* [POD84], pages 191–200.

[AB87] Atkinson, M. P. and P. Buneman: *Types and persistence in database programming languages.* ACM Computing Surveys, 19(2):105–190, June 1987.

[ABC+76] Astrahan, M. M., M. W. Blasgen, D. D. Chamberlin, et al.: *System R: Relational approach to database management.* ACM Transactions on Database Systems, 1(2):97–137, 1976.

[ABJ+90] Atkinson, M. P., F. Bancilhon, D. D. J., et al.: *The object-oriented database system manifesto.* In Kim, W., J.-M. Nicolas, and S. Nishio (editors): *Proc. DOOD'89*, pages 223–240. Elsevier Science, 1990.

[AC75] Astrahan, M. M. and D. D. Chamberlin: *Implementation of a structured english query language.* Communications of the ACM, 18(10):580–588, 1975.

[ACM92] ACM SIGCHI: *Curriculum for Human-Computer Interaction*, 1992. ACM Special Interest Group on Computer-Human Interaction Curriculum Development Group, New York.

[AFS89] Abiteboul, S., P. C. Fischer, and H.-J. Schek (editors): *Nested Relations and Complex Objects in Databases.* Number 361 in *Lecture Notes in Computer Science.* Springer, 1989.

[AG89] Agrawal, R. and N. H. Gehani: *ODE (object database and environment): The language and the data model.* In Clifford, J., B. Lindsay, and D. Maier (editors): *Proc. SIGMOD'89*, ACM SIGMOD Record 18(2), pages 36–45. ACM Press, 1989.

[AH84] Abiteboul, S. and R. Hull: *IFO: A formal semantic database model.* In *Proc. PODS'84* [POD84], pages 119–132.

[AH87] Abiteboul, S. and R. Hull: *IFO: A formal semantic database model.* ACM Transactions on Database Systems, 12(4):525–565, 1987.

[AH90] Andrews, T. and C. Harris: *Combinig language and database advances in an object-oriented development environment.* In Zdonik, S. B. and D. Maier (editors): *Readings in Object-Oriented Database Systems*, pages 186–196. Morgan Kaufmann, 1990.

[Ahm98] Ahmad, M.: *Tcl-DB: Entwurf und Implementierung einer Skriptsprache für den Datenbank-Editor ESCHER.* Diplomarbeit, Universität Gh Kassel, Fachbereich Mathematik/Informatik, 1998.

[AJ94] Andrew Johnson, F. F.: *The sandbox: A virtual reality interface to scientific databases.* In French, J. C. and H. Hinterberger (editors): *Proc. 7th Int. Conf. on Scientific and Statistical Database Management*, pages 12–21. IEEE Computer Science Press, 1994.

[ALPS87] Andersen, F., V. Linnemann, P. Pistor, and N. Südkamp: *Advanced Information Management Prototype — User manual for the online interface of the Heidelberg Data Base Language (HDBL) prototype implementation.* Technical Report TN 86.01, IBM Germany, Heidelberg Scientific Center, March 1987. Release 1.3.

[AMH+93] Ashlund, S., K. Mullet, A. Henderson, et al. (editors): *INTERCHI'93 Conference Proceedings: Human Factors in Computing Systems, INTERACT'93 and CHI'93, Bridges between Worlds, Amsterdam, The Netherlands, 24–29 April 1993.* ACM Press, 1993.

[ATW] Ahmad, M., J. Thamm, and L. Wegner: *Rapid application development for Web-based collaboration.* To be published as: Zhang, Y., M. Rusinkiewicz and Y. Kambayashi: *Proc. 2nd Int. Symp. on Cooperative Database Systems for Advanced Applications (CODAS'99)*, Springer (in print).

[AW89] Apers, P. M. G. and G. Wiederhold (editors): *Proceedings of the 15th International Conference on Very Large Data Bases, August 22–25, 1989, Amsterdam, The Netherlands.* Morgan Kaufmann, 1989.

[BAC+90] Boral, H., W. Alexander, L. Clay, et al.: *Prototyping Bubba, a highly parallel database system.* IEEE Transactions on Knowledge and Data Engineering, 2(1):4–23, 1990.

[Ban88] Bancilhon, F.: *Object-oriented database systems.* In *Proc. PODS'88*, pages 152–162. ACM Press, 1988.

[Bat86] Batory, D. S.: *GENESIS: A project to develop an extensible database management system.* In Dittrich, K. R. and U. Dayal [DD86], pages 207–208.

[BBB+88] Bancilhon, F., G. Barbedette, V. Benzaken, et al.: *The design and implementation of O_2, an object-oriented database system.* In Dittrich,

K. R. (editor): *Proc. OODBS'88*, number 334 in *Lecture Notes in Computer Science*, pages 1–22. Springer, 1988.

[BBC+98] Bernstein, P. A., M. L. Brodie, S. Ceri, et al.: *The Asilomar report on database research.* ACM SIGMOD Record, 27(4):74–80, December 1998.

[BCG+87] Banerjee, J., H.-T. Chou, J. F. Garza, et al.: *Data model issues for object-oriented applications.* ACM Transactions on Information Systems, 5(1):3–26, 1987.

[BCN92] Batini, C., S. Ceri, and S. Navathe: *Conceptual Database Design — an Entity-Relationship Approach.* Benjamin Cummings, 1992.

[BDD+89] Bernstein, P. A., U. Dayal, D. DeWitt, et al.: *Future directions in DBMS research — The Laguna Beach participants.* ACM SIGMOD Record, 18(1):17–26, March 1989.

[BDK92] Bancilhon, F., C. Delobel, and P. Kanellakis (editors): *Building an Object-Oriented Database System - The Story of O_2.* Morgan Kaufmann, 1992.

[BGBG95] Baecker, R. M., J. Grudin, W. Buxton, and S. Greenberg: *A historical and intellectual perspective.* In Baecker, R. M., J. Grudin, W. Buxton, and S. Greenberg (editors): *Readings in Human-Computer Interaction*, chapter 1, pages 35–48. Morgan Kaufmann, 2nd edition, 1995.

[BHPR97] Ballard, D. H., M. M. Hayhoe, P. K. Pook, and R. P. N. Rao: *Deictic codes for the embodiment of cognition.* Behavioral and Brain Sciences, 20(4):723–767, 1997.

[Bil92a] Biliris, A.: *An efficient database storage structure for large dynamic objects.* In Golshani, F. (editor): *Proc. ICDE'92*, pages 301–308. IEEE Computer Science Press, 1992.

[Bil92b] Biliris, A.: *The performance of three database storage structures for managing large objects.* In Stonebraker, M. (editor): *Proc. SIGMOD'92*, ACM SIGMOD Record 21(2). ACM Press, 1992.

[Bon99] Bonnert, E.: *Netscape will mit User-Interface Browser-Boden zurückgewinnen.* Computer Zeitung, 29:18, Juli 1999.

[BOS91] Butterworth, P., A. Otis, and J. Stein: *The GemStone object database management system.* Communications of the ACM, 34(10):64–77, 1991.

[BP98] Bray, T. and C. M. Paoli, Jean Sperberg-McQueen: *Extensible markup language (XML) 1.0.* Technical Report REC-xml-19980210, W3C, February 1998. Status: W3C Recommendation.
`http://www.w3.org/TR/1998/REC-xml-19980210.html`

[Bro84] Brodie, M. L.: *On the development of data models.* In Brodie, M. L., J. Mylopoulos, and J. W. Schmidt (editors): *On Conceptual Modelling*, pages 19–48. Springer, 1984.

[BS99] Basu, J. and J. Shome: *Essential SQLJ Programming : The Complete Guide to the ANSI Standard for Embedded SQL in Java.* John Wiley, July 1999.

[Cat91] Cattell, R. G. G.: *Object Data Managment.* Addison-Wesley, 1991.

[Cat94] Cattell, R. G. G. (editor): *The Object Database Standard: ODMG-93, Release 1.1.* Morgan Kaufmann, 1994.

[Cat96] Cattell, R. G. G. (editor): *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufmann, 1996.

[Cat97] Cattell, R. G. G. (editor): *The Object Database Standard: ODMG-2.0.* Morgan Kaufmann, 1997.

[CB91] Collet, C. and E. Brunel: *Definition and manipulation of forms with FO2.* In Knuth, E. and L. Wegner (editors): *Visual Database Systems '91*, pages 432–449. Elsevier Science, 1991.

[CCLB97] Catarci, T., M. F. Costabile, S. Levialdi, and C. Batini: *Visual query systems for databases: A survey.* Journal of Visual Languages and Computing, 8(2):215–260, April 1997.

[CCM92] Consens, M. P., I. F. Cruz, and A. O. Mendelzon: *Visualizing queries and querying visualizations.* ACM SIGMOD Record, 21(1):39–46, mar 1992.

[CD96] Carey, M. J. and D. J. DeWitt: *Of objects and databases: A decade of turmoil.* In Vijayaraman, T. M., A. P. Buchmann, C. Mohan, and N. L. Sarda (editors): *Proc. VLDB'96*, pages 3–14. Morgan Kaufmann, 1996.

[CDF+86] Carey, M. J., D. J. DeWitt, D. Frank, et al.: *The architecture of the EXODUS extensible DBMS.* In Dittrich, K. R. and U. Dayal [DD86], pages 52–65.

[CDN+94] Carey, M. J., D. J. DeWitt, J. F. Naughton, et al.: *Shoring up persistent applications.* In Snodegrass, R. T. and M. Winslett (editors): *Proc. SIGMOD'94*, ACM SIGMOD Record 23(2), pages 383–394. ACM Press, 1994.

[CGY81] Chamberlin, D. D., A. M. Gilbert, and R. A. Yost: *A history of System R and SQL/Data System.* In *Proc. VLDB'81*, pages 456–464. IEEE Computer Science Press, 1981.

[Che76] Chen, P. P.: *The entity-relationship model — towards a unified view of data.* ACM Transactions on Database Systems, 1(1):9–36, 1976.

[Chi95] Chimera, R.: *Platform independent GUI builders advance software engineering to handle HCI issues.* In Taylor, R. M. and J. Coutaz (editors): *Proc. ICSE'94 Workshop on SE-HCI*, number 896 in *Lecture Notes in Computer Science*, pages 28 ff. Springer, 1995.

[CJ92] Cellary, W. and G. Jomier: *Consistency of versions in object-oriented databases.* In Bancilhon, F. et al. [BDK92], chapter 19, pages 447–462.

[CM84] Copeland, G. and D. Maier: *Making Smalltalk a database system.* In Yormark, B. (editor): *Proc. SIGMOD'84*, ACM SIGMOD Record 14(2), pages 316–325. ACM Press, 1984.

[CNJ+98] Chen, H., J. Nunamaker Jr., et al.: *Information visualization for collaborative computing.* Computer, 31(8):75–82, August 1998.

[CNJOT98] Chen, H., J. Nunamaker Jr., R. Orwig, and O. Titkova: *Groupware and collaborative computing.* Computer, 31(8):76–78, August 1998. Sidebar in [CNJOT98].

[Cod70] Codd, E. F.: *A relational model of data for large shared data banks.* Communications of the ACM, 13(6):377–387, 1970.

[Com99] Computer Associates Inc.: *Jasmine TND.* WWW, 1999.
`http://www.cai.com/products/jasmine.htm`

[Cre89] Creasy, P. N.: *ENIAM: A more complete conceptual schema language.* In Apers, P. M. G. and G. Wiederhold [AW89], pages 107–114.

[D+90] Deux, O. et al.: *The story of O_2.* IEEE Transactions on Knowledge and Data Engineering, 2(1):91–108, 1990.

[Dad88] Dadam, P.: *Advanced Information Management (AIM): Research in extended nested relations.* Database Engineering, 7:119–129, 1988.

[DD86] Dittrich, K. R. and U. Dayal (editors): *International Workshop on Object-Oriented Database Systems, Pacific Grove, September 23–26, 1986.* IEEE Computer Science Press, 1986.

[DD98] Date, C. J. und H. Darwen: *SQL — Der Standard.* Addison-Wesley, 1998.

[DFF+98] Deutsch, A., M. Fernandez, D. Florescu, et al.: *XML-QL: A query language for XML.* Technical Report NOTE-xml-ql-19980819, W3C, August 1998. Status: Submission to the W3C.
`http://www.w3.org/TR/1998/NOTE-xml-ql-19980819.html`

[DGN95] Dayal, U., P. M. D. Gray, and S. Nishio (editors): *Proceedings of 21st International Conference on Very Large Data Bases, September 11–15, 1995, Zurich, Switzerland.* Morgan Kaufmann, 1995.

[DGS+90] DeWitt, D. J., S. Ghandeharizadeh, D. A. Schneider, et al.: *The Gamma database machine project.* IEEE Transactions on Knowledge and Data Engineering, 2(1):44–61, 1990.

[Dit86] Dittrich, K.: *Object-oriented database systems: The notions and the issues.* In Dittrich, K. R. and U. Dayal [DD86], pages 2–4.

[DKA+86] Dadam, P., K. Kuespert, F. Andersen, et al.: *A DBMS prototype to support extended NF^2 relations: An integrated view on flat tables and hierarchies.* In Zaniolo, C. (editor): *Proc. SIGMOD'86*, ACM SIGMOD Record 15(2), pages 356–367. ACM Press, 1986.

[DOP+85] Deppisch, U., V. Obermeit, H.-B. Paul et al.: *Ein Subsystem zur stabilen Speicherung versionsbehafteter, hierarchisch strukturierter Tupel.* In: Blaser, A. und P. Pistor (Herausgeber): *Proc. BTW'85*, Band 94 der Reihe *Informatik Fachberichte*, Seiten 421–440. Springer, 1985.

[DPS86] Deppisch, U., H.-B. Paul, and H.-J. Schek: *A storage system for complex objects.* In Dittrich, K. R. and U. Dayal [DD86], pages 183–195.

[DT87] Dadam, P. and J. Teuhola: *Managing schema versions in a time-versioned non-first-normal-form relational database.* Technical Report TR 87.01.001, IBM Germany, Heidelberg Scientific Center, January 1987.

[DTLS91] Dyballa, A., H. Toben, V. Linnemann und G. Saake: *Integration geometrischer Daten in ein erweiterbares Datenbanksystem.* Technischer Bericht TR 75.91.12, IBM Germany, Heidelberg Scientific Center, 1991.

[EM98] Eisenberg, A. und J. Melton: *Standards in Practice.* ACM SIGMOD Record, 27(3):53–85, 1998.

[End84] Enderle, G.: *Seeheim workshop on user interface management systems — first report.* Computer Graphics Forum, 3(2):169–170, June 1984.

[Fag77] Fagin, R.: *Multivalued dependencies and a new normal form for relational databases.* ACM Transactions on Database Systems, 2(3):262–278, 1977.

[Fra98] Franz, M.: *The Java virtual machine — a passing fad?* IEEE Software, 15(6):26–29, 1998.

[FS95] Fowler, S. L. and V. R. Stanwick: *The GUI Style Guide.* Academic Press, 1995.

[GHK+96] Goyal, N., C. Hoch, R. Krishnamurthy, et al.: *Is GUI programming a database research problem?* In Jagadish, H. V. and I. S. Mumick (editors): *Proc. SIGMOD'96*, ACM SIGMOD Record 25(2), pages 517–528. ACM Press, 1996.

[GKKZ92] Gottheil, K., H.-J. Kaufmann, T. Kern und R. Zhao: *X und Motif.* Springer, 1992.

[Gol88] Goldberg, A. (editor): *A History of Personal Workstations.* Addison-Wesley, 1988.

[Gou97] Gould, J.: *Interfacing with the future: Muse technologies.* Desktop Engineering, December 1997.
`http://www.deskeng.com/muse1297.htm`

[GWZ] Gillner, R., L. Wegner, and C. Zirkelbach: *Collaborative project management with a Web-based database editor.* Accepted paper for the 5th International Workshop on Multimedia Information Systems, Indian Wells, Palm Springs Desert, CA, USA, October 1999.

[Har97] Harrison, M.: *Tcl/Tk Tools.* O'Reilly, 1997.

[HBP+93] Hill, R. D., T. Brink, J. F. Patterson, et al.: *The Rendezvous language and architecture.* Communications of the ACM, 36(1):62–67, January 1993.

[HCF97] Hamilton, G., R. Cattell, and M. Fisher: *JDBC Database Access with Java: A Tutorial and Annotated Reference.* Addison-Wesley, 1997.

[HCL+90] Haas, L. M., W. Chang, G. M. Lohman, et al.: *Starburst mid-flight: As the dust clears.* IEEE Transactions on Knowledge and Data Engineering, 2(1):143–160, 1990.

[Heu89] Heuer, A.: *A data model for complex objects based on a semantic database model and nested relations.* In Abiteboul, S. et al. [AFS89], pages 297–312.

[Heu92] Heuer, A.: *Objektorientierte Datenbanken.* Addison-Wesley, 1992.

[HIL95] Haber, E. M., Y. E. Ioannidis, and M. Livny: *OPOSSUM: Desk-top schema management through customizable visualization.* In Dayal, U. et al. [DGN95], pages 527–538.

[HM98] Harrison, M. and M. McLennan: *Effective Tcl/Tk Programming.* Addison-Wesley, 1998.

[HMW+87] Härder, T., K. Meyer-Wegener, et al.: *PRIMA — a DBMS prototype supporting engineering applications.* In Stocker, P. M. et al. [SKH87], pages 433–442.

[HR99] Härder, T. und E. Rahm: *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, 1999.

[Hul90] Hulin, G.: *On restructuring nested relations in partitioned normal form.* In McLeod, D., R. Sacks-Davis, and H.-J. Schek (editors): *Proc. VLDB'90*, pages 626–637. Morgan Kaufmann, 1990.

[Ing91] Ingres Corporation: *Application Editor User's Guide for INGRES/Windos 4GL.* Ingres Corporation, Alameda, CA, USA, June 1991.

[Int87] International Organization of Standadization: *Database Language SQL.* ISO Copyright Office, Genf, 1987. Document ISO/IEC 9075:1987.

[Int96] International Organization of Standadization: *Information Technology – Database Language – SQL – Technical Corrigendum 2.* ISO Copyright Office, Genf, 1996. Document ISO/IEC 9075:1992/Cor. 2.

[Ioa92] Ioannidis, Y. E.: *Advanced user interfaces for database systems.* ACM SIGMOD Record, 21(1):4, March 1992. Letter from the special issue editor.

[Jai97] Jain, R.: *Visual information management.* Communications of the ACM, 40(12):30–32, December 1997. Introduction from the guest editor.

[Jon88] Jones, O.: *Introduction to the X Windows System.* Prentice-Hall, 1988.

[JRV+89] Johnson, J., T. L. Roberts, W. Verplank, et al.: *The Xerox Star: A retrospective.* Computer, 22(9):11–29, September 1989.

[JS82] Jaeschke, G. and H.-J. Schek: *Remarks on the algebra of non first normal form relations.* In *Proc. PODS'82*, pages 124–138. ACM Press, 1982.

[JWZ93] Janssen, C., A. Weisbecker, and J. Ziegler: *Generating user interfaces from data models and dialogue net specifications.* In Ashlund, S. et al. [AMH+93], pages 418–423.

[Kam99] Kamps, T.: *Diagram Design, A Constructive Theory.* Springer, July 1999.

[Kay93] Kay, A. C.: *The early history of Smalltalk.* ACM SIGPLAN Notices, 28(3):69–95, March 1993.

[KB96] Khoshafian, S. and A. B. Baker: *MultiMedia and Imaging Databases.* Morgan Kaufmann, 1996.

[KC88] Kim, W. and H.-T. Chou: *Versions of schema for object-oriented databases.* In Bancilhon, F. and D. J. DeWitt (editors): *Proc. VLDB'88*, pages 148–159. Morgan Kaufmann, 1988.

[KD95] Kalus, C. and P. Dadam: *Flexible relations — operational support of variant relational structures.* In Dayal, U. et al. [DGN95], pages 539–550.

[KDD95] Kotz-Dittrich, A. and K. R. Dittrich: *Where object-oriented DBMSs should do better: A critique based on early experiences.* In Kim, W. [Kim95], chapter 12, pages 238–254.

[Ker99] Kernighan, B.: *What have you learned today?* IEEE Software, 16(2):66–68, March/April 1999. Interview.

[KGBW90] Kim, W., J. F. Garza, N. Ballou, and D. Woelk: *Architecture of the ORION next generation database system.* IEEE Transactions on Knowledge and Data Engineering, 2(1):109–124, 1990.

[Kim95] Kim, W. (editor): *Modern Database Systems: The Object Model, Interoperability, and Beyond.* Addison-Wesley, 1995.

[KK93] Kemper, A. and D. Kossmann: *Adaptable pointer swizzling strategies in object bases.* In *Proc. ICDE'93*, pages 155–162. IEEE Computer Science Press, 1993.

[Kli96] Klingert, A.: *Einführung in Graphische Fenstersysteme: Konzepte und Reale Systeme.* Springer, 1996.

[KN92] King, R. and M. Novak: *Building data representations with FaceKit.* ACM SIGMOD Record, 21(1):11–17, March 1992.

[KS91] Korth, H. F. and A. Silberschatz: *Database System Concepts.* McGraw-Hill, 2nd edition, 1991.

[KSW89] Küspert, K., G. Saake, and L. M. Wegner: *Duplicate detection and deletion in the extended NF^2 data model.* In Litwin, W. and H.-J. Schek (editors): *Proc. FODO'89*, number 367 in *Lecture Notes in Computer Science*, pages 83–100. Springer, 1989.

[Kun89] Kunii, T. L. (editor): *Proceedings of the 1st IFIP Working Conference on Visual Database Systems, Tokyo, 1989.* Elsevier Science, 1989.

[Kun92] Kuntz, M.: *A versatile browser-editor for NF^2 relations.* In Chen, Q., Y. Kambayashi, and R. Sacks-Davis (editors): *Future Databases '92*, volume 3 of *Advanced Database Research and Development Series*, pages 266–275. World Scientific, 1992.

[KW87] Kemper, A. and M. Wallrath: *An analysis of geometric modeling in database systems.* ACM Computing Surveys, 19(1):47–91, 1987.

[KZ95a] Krishnamurthy, R. and M. Zloof: *RBE: Rendering by example.* In Yu, P. S. and A. L. P. Chen (editors): *Proc. ICDE'94*, pages 288–297. IEEE Computer Science Press, 1995.

[KZ95b] Krishnamurthy, R. and M. Zloof: *The Evolution of User Interface Tools for Database Applications*, July 1995. Tutorial C.

[Leu97] Leung, C. H. (editor): *Visual Information Systems*, number 1306 in *Lecture Notes in Computer Science*. Springer, 1997. Selected papers from the "1st Int. Conf. on Visual Information Systems", Feb. 1996.

[Lew98] Lewis, T. G.: *Java holy war '98*. Computer, 31(3):126–128, March 1998.

[Lie95] Lievaart, R.: *My life with Escher*. Study report, University of Kassel, Department of Mathematics and Computer Science, January 1995.

[LK86] Lyngbaek, P. and W. Kent: *A data modelling methodology for the design and implementation of information systems*. In Dittrich, K. R. and U. Dayal [DD86].

[LLOW91] Lamb, C., G. Landis, J. Orenstein, and D. Weinreb: *The ObjectStore database system*. Communications of the ACM, 34(10):50–63, 1991.

[Loe97] Loeser, H.: *Datenbankanbindung an das WWW — Techniken, Tools und Trends*. In: Dittrich, K. R. und A. Geppert (Herausgeber): *Proc. BTW'97*, Informatik Aktuell, Seiten 83–99. Springer, 1997.

[LPS91] Linnemann, V., P. Pistor, and N. Südkamp: *User manual of the AIM-P online interface*. Technical Report TN 91.08, IBM Germany, Heidelberg Scientific Center, June 1991.

[Mac85] Mac an Airchinnigh, M.: *Seeheim workshop on user interface management systems: Report of the working group on the user's conceptual model*. Computer Graphics Forum, 4(1):63–69, January 1985.

[Mak77] Makinouchi, A.: *A consideration on normal form of not-necessarily-normalized relation in the relational data model*. In *Proc. VLDB'77*, pages 447–453. IEEE Computer Science Press, 1977.

[Mal98] Malaika, S.: *Resistance is futile: The web will assimilate your database*. Data Engineering Bulletin, 21(2):4–13, June 1998.

[Mel95] Melton, J. (editor): *Database Language SQL (SQL3)*. ANSI, March 1995. ISO-ANSI Working Draft X3H2-95-084/DBL:YOW-004.

[Mel96] Melton, J. (editor): *Database Language SQL*. ISO, July 1996. ISO/IEC CD 9075 Committee Draft.

[MGD+90] Myers, B. A., D. Giuse, R. B. Dannenberg, et al.: *Garnet: Comprehensive support for graphical highly-interactive user interfaces*. Computer, 23(11):71–85, November 1990.

[MHC96] Myers, B. A., J. D. Hollan, and I. F. Cruz: *Strategic directions in human-computer interaction.* ACM Computing Surveys, 28(4):794–809, December 1996.

[Mic93] Microsoft Corporation: *Win32 Programmer's Reference*, volume 1. Microsoft Press, 1993.

[MMM+97] Myers, B. A., R. G. McDaniel, R. C. Miller, et al.: *The Amulet environment: New models for effective user interface software development.* IEEE Transactions on Software Engineering, 23(6):347–365, June 1997.

[MNE96a] Mok, W. Y., Y.-K. Ng, and D. W. Embley: *A normal form for precisely characterizing redundancy in nested relations.* ACM Transactions on Database Systems, 21(1):77–106, March 1996.

[MNE96b] Mok, W. Y., Y.-K. Ng, and D. W. Embley: *A normal form for precisely characterizing redundancy in nested relations.* ACM Transactions on Database Systems, 21(1):77–106, March 1996.

[Mos92] Moss, J. E. B.: *Working with persistent objects: To swizzle or not to swizzle.* IEEE Transactions on Software Engineering, 18(8):657–673, 1992.

[MR86] Mark, L. and N. Roussopoulos: *Metadata management.* Computer, 19(12):26–36, December 1986.

[MR92] Myers, B. A. and M. B. Rosson: *Survey on user interface programming.* In Bauersfield, P., J. Bennett, and G. Lynch (editors): *Proc. CHI'92*, pages 195–202. ACM Press, 1992.

[Muß92] Mussmann, B.: *Entwurf und Implementierung der internen Ebene von OSCAR: Objektrepräsentation durch Speicherstrukturen und Zugriffspfade.* Diplomarbeit, TU Clausthal, Institut für Informatik, März 1992.

[MW91] Meyer-Wegener, K.: *Multimedia-Datenbanken.* B. G. Teubner, 1991.

[MW94] Manthey, A. und P. Wehner: *Oracle Browser als Beitrag zur Informationsgewinnung im Rahmen der Individuellen Datenverarbeitung (IDV).* In: Wegner, L. M. [Weg94], Seiten 52–54.

[Mye94] Myers, B. A.: *User interafce software tools.* Technical Report CMU-CS-94-182, School of Computer Science, Carnegie Mellon University, August 1994.

[Mye96] Myers, B. A.: *A brief history of human computer interaction technology.* Technical Report CMU-CS-96-163, School of Computer Science, Carnegie Mellon University, December 1996.

[MZ95] Mowbray, T. J. and R. Zahavi: *The Essential CORBA — Systems Integration Using Distributed Objects.* OMG/John Wiley, August 1995.

[Nav92] Navathe, S. B.: *Evolution of data modeling for databases.* Communications of the ACM, 35(9):112–123, 1992.

[NGP+99] Nicol, J., Y. Gutfreund, J. Paschetto, et al.: *How the internet helps build collaborative multimedia applications.* Communications of the ACM, 42(1):79–86, January 1999.

[Nie99] Nielsen, J.: *User interface directions for the Web.* Communications of the ACM, 42(1):65–72, jan 1999.

[Obj98] Objectivity Inc.: *Objectivity Product Family Version 5.* WWW, 1998. `http://www.objectivity.com/Products/Objy/Objy5DB1.htm`

[OMG98] OMG: *CORBA/IIOP 2.2 Specification.* WWW, February 1998. `http://www.omg.org/corba/corbaiiop.html`

[Ora97] Oracle Corporation: *Oracle8 Server Concepts, The Data Dictionary*, chapter 4. Oracle Corporation, 1997.

[Ous94] Ousterhout, J. K.: *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Ous98] Ousterhout, J. K.: *Scripting: Higher-level programming for the 21st century.* Computer, 31(3):23–30, March 1998.

[Ous99] Ousterhout, J. K.: *Integration: A new style of programming.* Computer, 32(5):53, May 1999.

[OY87] Ozsoyoglu, Z. M. and L.-Y. Yuan: *A new normal form for nested relations.* ACM Transactions on Database Systems, 12(1):111–136, 1987.

[OY89] Ozsoyoglu, Z. M. and L.-Y. Yuan: *On the normalization in nested relational databases.* In Abiteboul, S. et al. [AFS89], pages 243–271.

[PA86] Pistor, P. and F. Andersen: *Designing a generalized NF2 model with an SQL-type language interface.* In Chu, W., G. Gardarin, S. Ohsuga, and Y. Kambayashi (editors): *Proc. VLDB'86*, pages 278–285. Morgan Kaufmann, 1986.

[Pau88] Paul, H.-B.: *DAS Datenbank-Kernsystem für Standard- und Nicht-Standard-Anwendungen — Architekur, Implementierung, Anwendungen.* Dissertation, TH Darmstadt, 1988.

[Pau94] Paul, M.: *Typerweiterungen im eNF^2-Datenmodell.* Dissertation, Universität Gh Kassel, August 1994.

[PCD92] Pausch, R., M. Conway, and R. DeLine: *Lessons learned from SUIT, the simple user interface toolkit.* ACM Transactions on Information Systems, 10(4):320–344, October 1992.

[PD89] Pistor, P. and P. Dadam: *The advanced information management system.* In Abiteboul, S. et al. [AFS89], pages 4–26.

[Pis93] Pistor, P.: *Objektorientierung in SQL3: Stand und Entwicklungstendenzen.* Informatik-Spektrum, 16:89–94, 1993.

[PKP98] Prinz, H. und U. Kirch-Prinz: *Objektorientiert Programmieren mit ANSI C++.* Prentice-Hall, 1998.

[Pla85] Plaff, G. (editor): *User Interface Management Systems: Proceedings of the Seeheim Workshop.* Springer, 1985.

[POD84] *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2–4, 1984, Waterloo, Ontario, Canada.* ACM Press, 1984.

[POE99] POET Software: *POET Object Server Suite — Im Überblick.* WWW, 1999. `http://www.poet.de/produkte/oss/overview.html`

[Pop98] Pope, A.: *The CORBA Reference Guide: Understanding the Common Object Request Broker Architecture.* Addison-Wesley, January 1998.

[POS+98] Pajarola, R., T. Ohler, P. Stucki, K. Szabo, and P. Widmayer: *The alps at your fingertips: Virtual reality and geoinformation systems.* In *Proc. ICDE'98*, pages 550–557. IEEE Computer Science Press, 1998.

[Pre93] Press, L.: *Before the Altair: The history of personal computing.* Communications of the ACM, 36(9):27–33, September 1993.

[Pre94] Preece, J.: *Human-Computer Interaction.* Addison-Wesley, 1994.

[PVG92] Paredaens, J. and D. Van Gucht: *Converting nested algebra expressions into flat algebra expressions.* ACM Transactions on Database Systems, 17(1):65–93, March 1992.

[RKS88] Roth, M. A., H. F. Korth, and A. Silberschatz: *Extended algebra and calculus for nested relational databases.* ACM Transactions on Database Systems, 13(4):389–417, 1988.

[RKS+91] Rowe, L. A., J. A. Konstan, B. C. Smith, et al.: *The* picasso *application framework.* In *Proceedings of the ACM Symposium on User Interface Software and Technology, Hilton Head, SC, USA*, pages 95–105, November 1991.

[RLPM96] Rudisill, M., C. Lewis, P. G. Polson, and T. McKay (editors): *Human-Computer Interface Design: Success Stories, Emerging Methods, and Real-World Context.* Morgan Kaufmann, 1996.

[Rod92] Roddick, J. F.: *Schema evolution in database systems — an annotated bibliography.* ACM SIGMOD Record, 21(4):35–40, 1992.

[Rod96] Rodden, T.: *Populating the application: A model of awareness for cooperative applications.* In *Proc. ACM CSCW'96, Conf. on Computer-Supported Cooperative Work, Learning from Space and Place, November 16–20, 1996, Boston, Mass., USA*, pages 87–96. ACM Press, 1996.

[Row92] Rowe, L. A.: *A retrospective on database application development frameworks.* ACM SIGMOD Record, 21(1):5–10, March 1992.

[RS82] Rowe, L. A. and K. A. Shoens: *FADS — A form application development system.* In Schkolnick, M. (editor): *Proc. SIGMOD'82*, pages 28–39. ACM Press, 1982.

[RS99] Reichenberger, K. und R. Steinmetz: *Visualisierungen und ihre Rolle in Multimedia-Anwendungen.* Informatik-Spektrum, 22(2):88–98, April 1999.

[SAD+93] Stonebraker, M., R. Agrawal, U. Dayal, et al.: *DBMS research at a crossroads: The Vienna update.* In Agrawal, R., S. Baker, and D. A. Bell (editors): *Proc. VLDB'93*, pages 688–692. Morgan Kaufmann, 1993.

[SB98] Stonebraker, M. and P. Brown: *Object-Relational DBMSs.* Morgan Kaufmann, 2nd edition, September 1998.

[SCF+86] Schwarz, P. M., W. Chang, J. C. Freytag, et al.: *Extensibility in the Starburst database system.* In Dittrich, K. R. and U. Dayal [DD86], pages 85–92.

[Sch93] Schildt, H.: *Windows NT Programming Handbook.* McGraw-Hill, 1993.

[Sci91] Sciore, E.: *Using annotations to support multiple kinds of versioning in an object-oriented database system.* ACM Transactions on Database Systems, 16(3):417–438, 1991.

[SFG93] Sukaviriya, P., J. D. Foley, and T. Griffith: *A second generation user interface design environment: The model and the runtime architecture.* In Ashlund, S. et al. [AMH+93], pages 375–382.

[Shn92] Shneiderman, B.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley, 1992.

[SKH87] Stocker, P. M., W. Kent, and P. Hammersley (editors): *Proceedings of the 13th International Conference on Very Large Data Bases, September 1–4, 1987, Brighton, England.* Morgan Kaufmann, 1987.

[SL97] Sears, A. and A. M. Lund: *Creating effective user interfaces.* IEEE Software, 14(4):21–24, July/August 1997. Guest editors' introduction.

[SLPW89] Saake, G., V. Linnemann, P. Pistor, and L. M. Wegner: *Sorting, grouping and duplicate elimination in the Advanced Information Management Prototype.* In Apers, P. M. G. and G. Wiederhold [AW89], pages 307–316.

[Spa94] Spaccapietra, S.: *User interfaces; who cares?* In Bocca, J. B., M. Jarke, and C. Zaniolo (editors): *Proc. VLDB'94*, page 751. Morgan Kaufmann, 1994. Panel Abstract.

[SPS87] Scholl, M. H., H.-B. Paul, and H.-J. Schek: *Supporting flat relations by a nested realtional kernel.* In Stocker, P. M. et al. [SKH87], pages 137–146.

[SPSW90] Schek, H.-J., H.-B. Paul, M. H. Scholl, and G. Weikum: *The DASDBS project: Objectives, experiences, and future prospects.* IEEE Transactions on Knowledge and Data Engineering, 2(1):25–43, March 1990.

[SRH90] Stonebraker, M., L. A. Rowe, and M. Hirohama: *The implementation of POSTGRES.* IEEE Transactions on Knowledge and Data Engineering, 2(1):125–142, 1990.

[SRL+90] Stonebraker, M., L. A. Rowe, B. Lindsay, et al.: *Third-generation database system manifesto.* ACM SIGMOD Record, 19(3):31–44, 1990.

[SS86] Schek, H.-J. and M. H. Scoll: *The relational model with relation-valued attributes.* Information Systems, 11(2):137–147, 1986.

[SSU96] Silberschatz, A., M. Stonebraker, and J. Ullman: *Database research: Achievements and opportunities into the 21st century.* ACM SIGMOD Record, 25(1):52–63, 1996.

[Sto86] Stonebraker, M.: *Object management in Postgres using procedures.* In Dittrich, K. R. and U. Dayal [DD86], pages 66–72.

[Sto90] Stonebraker, M.: *Introduction to the special issue on database prototype systems.* IEEE Transactions on Knowledge and Data Engineering, 2(1):1–3, March 1990.

[Sto96] Stonebraker, M.: *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, 1996.

[SZBH86] Swinehart, D., P. Zellweger, R. Beach, and R. Hagemann: *A structural view of the Cedar programming environment.* ACM Transactions on Programming Languages and Systems, 8(4):419–490, October 1986.

[TF76] Taylor, R. W. T. and R. L. Frank: *CODASYL data-base management systems.* ACM Computing Surveys, 8(1):67–103, March 1976.

[TF86] Thomas, S. J. and P. C. Fischer: *Nested relational structures.* In *Advances in Computing Research*, volume 3, pages 269–307. JAI Press, 1986.

[Tha94] Thamm, J.: *Entwurf und Implementierung der internen Ebene von OSCAR: Verwendung von Indexstrukturen.* Diplomarbeit, TU Clausthal, Institut für Informatik, September 1994.

[Tha98] Thamm, J.: *A Database Approach to GUI Management.* In: Scholl, M. H., H. Riedel, T. Grust und D. Gluche (Herausgeber): *10. Workshop „Grundlagen von Datenbanken“*, Nummer 63 in *Konstanzer Schriften in Mathematik und Informatik*, Seiten 125–129, Mai 1998.

[The96] Thelemann, S.: *Semantische Anreicherung eines Datenmodells für komplexe Objekte.* Dissertation, Universität Gh Kassel, Juni 1996.

[TL76] Tsichritzis, D. and F. H. Lochovsky: *Hierarchical data-base management: A survey.* ACM Computing Surveys, 8(1):105–123, March 1976.

[TTW96] Thamm, J., S. Thelemann, and L. Wegner: *Visual information systems — a database perspective.* In Du, D. and O. R. Liu Sheng (editors): *Proc. DMS'96*, pages 274–285. Knowledge Systems Institute, Skokie, IL, USA, 1996.

[Tur99] Turau, V.: *Techniken zur realisierung web-basierter anwendungen.* Informatik-Spektrum, 22(1):3–12, February 1999.

[TW91] Teuhola, J. and L. Wegner: *Minimal space, average linear time duplicate deletion.* Communications of the ACM, 34(3):62–73, 1991.

[TW98] Thamm, J. and L. Wegner: *What you see is what you store: Database-driven interfaces.* In Ioannidis, Y. and W. Klas (editors): *Visual Database Systems 4 (VDB4)*, pages 69–84. Chapman & Hall, 1998.

[TWW99] Thamm, J., S. Wilke, and L. Wegner: *A Web solution to concurrency awareness in shared data spaces.* In Kambayashi, Y., D. L. Lee, E.-P. Lim, et al. (editors): *Advances in Database Technologies, Proc. ER'98 Workshops*, number 1552 in *Lecture Notes in Computer Science*, pages 382–395. Springer, 1999.

[UGB93] Ukelson, J. P., J. D. Gould, and S. J. Boies: *User navigation in computer applications.* IEEE Transactions on Software Engineering, 19(3):297–306, March 1993.

[Ull88] Ullman, J. D.: *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988. Classical Database Systems.

[Ver99] Versant Corporation: *Versant Release 5.* WWW, 1999. `http://www.versant.com/us/products/release/index.html`

[Vin95] Vincent, M. W.: *Redundancy elimination and a new normal form for relational database design.* In Libkin, L. and B. Thalheim (editors): *Semantics in Databases*, number 1358 in *Lecture Notes in Computer Science*, pages 247–264. Springer, 1995.

[VZM90] Vander Zanden, B. and B. A. Myers: *Automatic, look-and-feel independent dialog creation for graphical user interfaces.* In Carrasco, J. and J. Whiteside (editors): *Proc. CHI'90*, pages 27–34. ACM Press, 1990.

[WBB+90] Wiecha, C., W. Bennett, S. Boies, et al.: *ITS: A tool for rapidly developing interactive applications.* ACM Transactions on Information Systems, 8(3):204–236, July 1990.

[Weg89] Wegner, L.: *ESCHER — interactive, visual handling of complex objects in the extended NF^2-database model.* In Kunii, T. L. [Kun89], pages 277–297.

[Weg90] Wegner, L.: *A Portable Record Manager.* Mathematische Schriften Kassel 11/90, Universität Gh Kassel, Fachbereich Mathematik/Informatik, Oktober 1990.

[Weg91a] Wegner, L.: *Let the fingers do the walking: Object manipulation in an NF^2 database editor.* In Maurer, H. (editor): *New Results and New Trends in Computer Science*, number 555 in *Lecture Notes in Computer Science*, pages 337–358. Springer, 1991.

[Weg91b] Wegner, L.: *Managing Persistence in ESCHER.* Mathematische Schriften Kassel 7/91, Universität Gh Kassel, Fachbereich Mathematik/Informatik, Juni 1991.

[Weg94] Wegner, L. M. (Herausgeber): *Beiträge zum Fachgruppentreffen „Benutzungsschnittstellen für Datenbanken" der GI-Fachgruppe „Datenbanken", Universität Gh Kassel, 17. und 18. März 1994*, Datenbank-Rundbrief 13, S. 22–69, Mai 1994.

[Wil96a] Wilke, D.: *Anforderungen und Konzepte der Handhabung komplexer Objekte in einem Datenbankeditor.* Diplomarbeit, Universität Gh Kassel, Fachbereich Mathematik/Informatik, April 1996.

[Wil96b] Wilke, S.: *Einsatz optimierter Visualisierungstechniken zur Darstellung komplexer Datenbankobjekte.* Diplomarbeit, Universität Gh Kassel, Fachbereich Mathematik/Informatik, März 1996.

[WLH90] Wilkinson, K., P. Lyngbæk, and W. Hasan: *The Iris architecture and implementation.* IEEE Transactions on Knowledge and Data Engineering, 2(1):63–75, 1990.

[WMJ+99] Witten, I. H., R. J. McNab, S. Jones, et al.: *Managing complexity in a distributed digital library.* Computer, 32(2):74–79, February 1999.

[WPC92] Wegner, L., M. Paul, and R. Colomb: *Variants and recursive types in the eNF^2 data model.* Technical Report 233, University of Queensland, June 1992.

[WPTT96] Wegner, L., M. Paul, J. Thamm, and S. Thelemann: *Pointer swizzling in non-mapped object stores.* Australian Computer Science Communications, 18(2):11–20, 1996. R. Topor (editor): *Proc. 7th Australasian Database Conference (ADC'96), Melbourne, Australia, 29–30 Jan. 1996.*

[WT89] Wegner, L. M. and J. Teuhola: *The external heapsort.* IEEE Transactions on Software Engineering, 15(7):917–925, July 1989.

[WTWL96] Wegner, L., S. Thelemann, S. Wilke, and R. Lievaart: *QBE-like queries and multimedia extensions in a nested relational DBMS.* In *Proc. Visual'96*, pages 437–446. Victoria University, Melbourne, 1996.

[WTZ97] Wegner, L., J. Thamm und C. Zirkelbach: *Data Manager Redesign: Implementation Notes.* Universität Gh Kassel, Fachbereich Mathematik/Informatik, Fachgruppe Datenbanken, interner Bericht, September 1997.

[WWT97] Wilke, D., L. Wegner, and J. Thamm: *Database-driven GUI programming.* In *Proceedings of the Second International Conference on Visual Information Systems, San Diego, December 15–17, 1997*, pages 205–214. Knowledge Systems Institute, Skokie, IL, USA, 1997.

[YH97] Yee, L. and C. Hsu: *A virtual reality interface to an enterprise metadatabase.* In Embley, D. W. and R. C. Goldstein (editors): *Proc. ER'97*, number 1331 in *Lecture Notes in Computer Science*, pages 436–449. Springer, 1997.

[Zir92] Zirkelbach, C.: *Geometrisches Clustern - ein metrischer Ansatz.* Dissertation, Julius-Maximilians-Universität Würzburg, 1992.

[Zlo82] Zloof, M. M.: *A language for office and business automation.* In Goos, G. and J. Hartmanis (editors): *Database Design Techniques*, number 133 in *Lecture Notes in Computer Science*, pages 297–319. Springer, 1982.