DIPLOMARBEIT

Beiträge zur typsicheren generischen Datenvisualisierung

Juli 1994

Sven Müßig
Falladabogen 2
22175 Hamburg

Tel.: 040 / 640 14 60

Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

Betreuer:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Horst Oberquelle

Widmung

Diese Arbeit ist meinem ehemaligen
Studienfreund Folker Kirch gewidmet.

Mit ihm zusammen habe ich in unserer
gemeinsamen Studienarbeit die Grundlagen
für die vorliegende Arbeit erstellt.

Danksagung

An dieser Stelle möchte ich mich bei meinen Betreuern, den Herren Prof. Dr. Joachim W. Schmidt und Prof. Dr. Horst Oberquelle sowie Dr. Florian Matthes, für die Anregungen zu diesem interessanten Thema sowie für ihre Unterstützung bei der Durchführung dieser Arbeit bedanken.

Andreas Rudloff danke ich für seine Einführungen in DBPL und INGRES/Windows 4GL und Rainer Müller für die Einführung in O_2. Andreas gilt darüber hinaus besonderer Dank für die unermüdliche Unterstützung bei ungezählten Problemen mit Unix und dem Tycoon-System.

Den ungenannten Anwendern und Testern der beiden im Rahmen dieser Arbeit erstellten Bibliotheken möchte ich für die konstruktive Kritik und die Fehlerbeschreibungen danken.

Dank gebührt ebenfalls Gerald Schröder, Klaus Reimers und Kerstin Lutze, welche mich nach der Durchsicht der Arbeit auf Unklarheiten aufmerksam gemacht haben. Meiner Mutter und Imke Goebel danke ich für das Korrekturlesen der Arbeit.

Der Text dieser Arbeit ist auf drei unterschiedlichen Computersystemen entstanden. Erste Texte sind unter Verwendung der TEX-Implementation von Stefan Lindner auf einem Atari ST [Lindner 93] entstanden. Fertiggestellt wurde die Arbeit unter DirectTEX [Ricken 94] von Wilfried Ricken auf einem Power Macinstosh. Die endgültige Fassung wurde dann auf einem Unix-Rechner von Sun übersetzt und ausgedruckt [Zierke 93].

Viele Fragen im Zusammenhang mit LATEX sind durch [Goossens et al. 94] und [Partl et al. 90] beantwortet worden. Darüber hinaus danke ich Dieter Nützel und Uwe Ellermann für deren Unterstützung in allen Fragen, die LATEX betrafen. Mein Dank gilt auch Frank Neukam für die von ihm erstellte Document-Style-Familie *Script* [Neukam 93], die das Layout dieser Arbeit wesentlich beeinflußt hat und Anselm Lingnau für seinen *float*-Style [Lingnau 92], ohne den die Erstellung der Tabellen im Anhang in dieser Form nicht möglich gewesen wäre.

Meinen Eltern möchte ich an dieser Stelle besonders dafür danken, daß sie mir das Informatikstudium und damit auch diese Arbeit ermöglicht haben.

Hamburg, im Juli 1994 Sven Müßig

Typographie

In dieser Arbeit werden unterschiedliche Schriftarten verwendet, um die Lesbarkeit zu steigern. In der folgenden Tabelle sind alle verwendeten Schriftarten jeweils mit ihrer Bedeutung und einem Beispiel aufgeführt.

Schriftart	**Bedeutung**	**Beispiel**
Emphasized	wichtiger oder englischer Begriff	*generische* Dienste *look & feel*
Sans Serif	Firmen- oder Produktname	Sun Microsystems OpenWindows
Bold Face	reserviertes Schlüsselwort	**let** a = *3*
Slanted	Bezeichner	*succ(number)*

Für die Darstellung der Programmbeispiele wird aus Tycoon die Konvention übernommen, daß Schlüsselwörter und sonstige Bezeichner, die im Zusammenhang mit Typen Verwendung finden, mit einem großen Buchstaben beginnen. Alle anderen Bezeichner beginnen mit einem kleinen Buchstaben.

Inhaltsverzeichnis

1. Einführung und Motivation **1**

1.1 Zielsetzung und Lösungsansätze 1

1.2 Struktur der Arbeit 4

2. Entwicklungswerkzeuge für graphische Benutzerschnittstellen **7**

2.1 Fenstersysteme 8

2.2 Toolkits 9

2.3 Ressource-Editoren 9

2.4 Quelltext-Programmgeneratoren 9

2.5 Interpreter-Entwicklungsumgebungen 10

2.6 Klassenbibliotheken 10

2.7 Systemübergreifende Bibliotheken 11

2.8 4GL-Systeme 12

2.9 Zusammenfassung und Klassifizierung 12

3. Programmierumgebungen für datenintensive Anwendungen **15**

3.1 Lose Kopplung durch Wirtsspracheneinbettung 16

3.2 Systemintegration in Datenbankprogrammiersprachen 17

3.3 Systemintegration im Tycoon-System 18

4. Datenvisualisierung in bestehenden Systemen **23**

4.1 Untersuchungsmethodik und Kriterien 24

4.1.1 Untersuchungsmethodik 24

4.1.2 Untersuchte Kriterien 25

4.2 Vorstellung der untersuchten Systeme 28

4.2.1 DBPL 28
4.2.2 INGRES 35
4.2.3 O_2 39
4.2.4 GemStone 41
4.2.5 VisualBASIC 44
4.3 Zusammenfassung 48

5. Kommerzielle Dienste einer speziellen Anwendungsumgebung 51

5.1 Die Seitenbeschreibungssprache PostScript 51
5.1.1 Graphikmodell 52
5.1.2 Programmierung 52
5.2 Das Fenstersystem NeWS 54
5.2.1 Grundbegriffe 54
5.2.2 Programmierung 55
5.3 Die OPEN LOOK Benutzerschnittstelle 57
5.3.1 Bildschirmhintergrund 57
5.3.2 Interaktionselemente 58
5.3.3 Fenster und Fensterelemente 60
5.3.4 Rückmeldungen 61
5.4 Das NeWS Toolkit 61
5.4.1 Organisation 62
5.4.2 Kommunikation und Ereignisverwaltung 64
5.4.3 Textbearbeitung 64
5.4.4 Datenaustausch 65
5.4.5 Anwendungsentwicklung 65
5.5 Die OpenWindows Anwendungsumgebung 66
5.5.1 Der X11/NeWS Server 67
5.5.2 Der Developer's Guide 68
5.5.3 Zusammenspiel der kommerziellen Dienste 69

6. Graphische Benutzerschnittstellen in Tycoon **71**

6.1 Dynamische Erzeugung graphischer Benutzerschnittstellen 71

6.1.1 Kommunikation und Ereignisverwaltung 72

6.1.2 Schnittstelle zum NeWS Toolkit 76

6.1.3 Schnittstelle zum Devguide 80

6.1.4 Schnittstelle zu benutzerdefinierten Klassen 80

6.1.5 Texteditoren 81

6.2 Graphische Anwendungsprogrammierung 81

6.2.1 Ein Klasseneditor zur Entwurfsunterstützung 81

6.2.2 Ein Graphikeditor zur objektorientierten Datenmodellierung 83

7. Datenvisualisierung in Tycoon **85**

7.1 Gestaltung der graphischen Benutzerschnittstelle 85

7.1.1 Basiseditoren 86

7.1.2 Verbundeditoren 87

7.1.3 Massendateneditoren 88

7.1.4 Rahmen für Editoren 89

7.2 Anwendungsprogrammierung am Beispiel 92

7.2.1 Visualisierung unstrukturierter Werte 92

7.2.2 Visualisierung strukturierter Werte 94

7.3 Module und Schnittstellen der Editorbibliothek 95

7.3.1 Abstraktes Datenstrukturmodell 96

7.3.2 Hilfsmodule 101

7.3.3 Editoren 104

7.3.4 Schnittstelle zum Anwendungsprogramm 112

7.4 Benutzerdefinierte Editoren 112

8. Bewertung und Ausblick **115**

8.1 Zusammenfassung 115

8.2 Bewertung 116

8.3 Ausblick 117

A. Programmbeispiel **121**

A.1 Tycoon 121

A.2 DBPL 124

A.3 INGRES 125

A.4 O_2 126

B. Tabellen **127**

B.1 Hard- und Softwareplattformen 128

B.2 Systemkomponenten 129

B.3 Visualisierungsmöglichkeiten 131

B.4 Interoperabilität 133

Literaturverzeichnis **135**

1. Einführung und Motivation

Früher wurden Computersysteme überwiegend von Experten eingesetzt, um damit naturwissenschaftliche Probleme zu lösen. Die zur Verfügung stehende Rechenkapazität war knapp und teuer. Deshalb war die Effizienz der Programme wichtiger als die Anforderungen der Anwender an die Benutzerunterstützung. Heutzutage werden Computer von einer immer breiteren Anwenderschicht z.B. zur Erledigung von Büroarbeiten oder im privaten Bereich eingesetzt.

Um die Bedienungsfreundlichkeit der Computer zu steigern und den Einarbeitungsaufwand in komplexe Anwendungsprogramme zu verringern, sind graphische Benutzerschnittstellen (*graphical user interfaces*, GUI's) entwickelt und zum Einsatz gebracht worden. Solche Benutzerschnittstellen stellen jedoch hohe Anforderungen an Ressourcen wie Speicher, Prozessorleistung und Graphikauflösung. Erst in den letzten Jahren konnten diese Anforderungen durch entsprechend leistungsfähige Computersysteme erfüllt werden [Mandelkern 93].

Der hohe Aufwand für die Erstellung von Anwendungsprogrammen mit graphischer Benutzerschnittstelle steht im krassen Gegensatz zu den angestrebten Vereinfachungen bei deren Bedienung. Dadurch wird die weitere Entwicklung und Verbreitung von graphischen Benutzerschnittstellen für Anwendungsprogramme erheblich erschwert.

Darüber hinaus tritt die Entwicklung portabler Informationssysteme und hier speziell portabler Benutzerschnittstellen für unterschiedliche Hard- und Software-Plattformen immer stärker in den Vordergrund. Aus den genannten Gründen stellen graphische Benutzerschnittstellen neben Datenbanken und Kommunikationssystemen die dritte wichtige technologische Säule für Systemarchitekturen zukünftiger Informationssysteme dar [Fähnrich et al. 92].

1.1 Zielsetzung und Lösungsansätze

Eine besondere Bedeutung bekommen graphische Benutzerschnittstellen im Zusammenhang mit datenintensiven Anwendungen. Die Anforderungen an die Funktionalität von Benutzerschnittstellen datenintensiver Anwendungen bestehen typischerweise darin, daß in großen und komplexen Datenbeständen navigiert werden muß, um eine benutzerdefinierte Auswahl anzuzeigen oder zu verändern. Die Spannweite der Daten reicht dabei von einfachen Zahlen und Zeichenketten bis hin zu Mengen komplexer Multimediaobjekte.

Diese speziellen Anforderungen *datenintensiver Anwendungen* an die graphische Benutzerschnittstelle stehen im direkten Zusammenhang mit den Eigenschaften datenintensiver Anwendungen. Die Anforderungen können wie folgt zusammengefaßt werden:

Datenumfang: Datenintensive Anwendungen lassen sich vor allem durch die Notwendigkeit zur Verwaltung sehr großer Datenmengen charakterisieren. Die Daten passen nicht mehr in den Hauptspeicher. An der Benutzerschnittstelle muß dieser Tatsache durch intelligente Ansteuerungen Rechnung getragen werden. Zum Beispiel ist es nicht möglich, alle Daten einer Datenbanktabelle in eine Tabelle der Benutzerschnittstelle zu kopieren und damit die Navigationsoperationen dem Bildschirmserver zu überlassen. Vielmehr ist es notwendig, immer nur die gleichzeitig sichtbaren Daten an die Benutzerschnittstelle zu übergeben und bei entsprechenden Operationen fehlende Daten zu ergänzen.

Datensemantik: Zwischen den Daten einer Anwendung bestehen oft komplexe Beziehungen, wie z.B. Assoziations- ($1:1$, $1:n$ und $n:m$), Aggregations- und Kollektionsbeziehungen. Diese Beziehungen gilt es an der Benutzerschnittstelle für den Anwender geeignet und unmißverständlich zu repräsentieren.

Datenintegrität: Integritätsbedingungen auf Massendaten werden typischerweise durch Prädikate beschrieben. Dies führt zu Vor- und Nachbedingungen, die bei Änderungsoperationen überprüft werden müssen. Der Anwender muß Rückmeldungen über den Erfolg bzw. den Mißerfolg solcher Operationen erhalten. Datenbanken geben Rückmeldungen jedoch nur in Form von Zahlenkodes oder Kürzeln an das Anwendungsprogramm zurück. Über die Benutzerschnittstelle müssen sie in geeigneter Form an den Anwender weitergeleitet werden.

Langlebigkeit: Eine weitere wichtige Eigenschaft von datenintensiven Anwendungen ist die Langlebigkeit. Damit sind jedoch nicht allein die Daten selbst gemeint, sondern auch immer wiederkehrende Benutzeranfragen und Funktionen zur Visualisierung der Anfrageergebnisse.

Benutzerklassen: Die letzte hier betrachtete Eigenschaft sind die unterschiedlichen Benutzerklassen. Verschiedene Anwender besitzen auf den langlebig verwalteten Massendaten auch verschiedene Sichten und unterschiedliche Zugriffsrechte. Daher ist es erforderlich, für eine Tabelle der Datenbank mehrere Bildschirmmasken anzubieten. Der Aufbau solcher Masken vereinfacht sich durch die Verwendung von funktionalen bzw. prozeduralen Abstraktionen.

Diese Anforderungen werden von den heute am Markt angebotenen Entwicklungswerkzeugen für graphische Benutzerschnittstellen i.a. nicht erfüllt. Es besteht daher ein Bedarf an Werkzeugen, durch die die Entwicklung von datenintensiven Anwendungen mit graphischer Benutzerschnittstelle vereinfacht wird.

Aufgrund dieser Problemstellung wird im Rahmen der vorliegenden Arbeit ein Ansatz zur typsicheren generischen Datenvisualisierung für datenintensive Anwendungen entwickelt und durch zwei Bibliotheken realisiert. Zur Implementation der beiden Bibliotheken wird die Programmiersprache TL (*Tycoon Language*) verwendet. Sie ist Bestandteil des am Arbeitsbereich *Datenbanken und Informationssysteme* (DBIS) entwickelten Tycoon-Systems (*TYped Communicating Objects in Open eNvironments*). Das Ziel ist ein modellneutraler Datenbankbrowser zum Anzeigen und Verändern von sowie zum Navigieren in den komplexen Datenstrukturen des Tycoon-Objektspeichers. Die Realisierung des Datenbankbrowsers als generische Bibliothek hat den Vorteil der universellen Verwendbarkeit. Der Browser läßt sich sowohl in komplexen Anwendungsprogrammen als auch zur Anzeige von interaktiven Datenbankanfragen verwenden. Als Grundlage für diese Bibliothek wird eine Schnittstelle zum NeWS Toolkit realisiert und in der Arbeit vorgestellt. Die Schnittstelle ermöglicht Tycoon-Programmen die typsichere Erstellung und Manipulation von graphischen Benutzerschnittstellen. Beide Bibliotheken sind im Rahmen dieser Arbeit auch implementiert worden.

Bevor der Aufbau der vorliegenden Arbeit genauer erläutert wird, ist es notwendig, drei häufig verwendete Begriffe zu definieren und ihren Zusammenhang mit dem Tycoon-System und der Visualisierung von Daten zu erläutern.

Typsicherheit: Ein Typfehler kann durch die Anwendung von Funktionen auf für sie nicht zugelassene Argumente oder durch den Versand von Nachrichten an Objekte, die diese Nachricht nicht verstehen können, auftreten. Eine Sprache heißt *statisch typsicher*, wenn alle Typfehler zur Übersetzungszeit erkannt werden können. Zur Laufzeit können dann keine Typfehler mehr auftreten [Atkinson, Bunemann 87; Heuer 92].

Generik: Eine Funktion heißt *generisch* oder *polymorph*, wenn sie auf Werte verschiedener Typen angewendet werden kann [Cardelli, Wegner 85].

Persistenz: Eine Sprache heißt *orthogonal persistent*, wenn sie Langlebigkeit für alle Daten unabhängig von ihrem Typ ermöglicht. Darüber hinaus ist eine persistente Sprache dadurch gekennzeichnet, daß keine Transportfunktionen notwendig sind, um die Daten zwischen dem Hauptspeicher und dem Externspeicher zu bewegen [Atkinson, Bunemann 87].

Diese Spracheigenschaften sind sowohl für die Programmierer von Bibliotheken als auch für deren Anwender von Vorteil. In statisch typisierten Datenbanksprachen wird durch eine textuelle statische Analyse des Quelltextes das Auftreten von Typfehlern zur Laufzeit vermieden. Dadurch kann z.B. zugesichert werden, daß ein Browser für Personendaten auch nur auf Variablen dieses Typs angewendet wird.

Das Konzept der statischen Typsicherheit führt jedoch zu einem Zielkonflikt in monomorphen Datenbankprogrammiersprachen. Zum einen ergeben sich die eben beschriebenen Konsistenzvorteile. Zum anderen ist die Wiederverwendbarkeit der erstellten Visualisierungsfunktionen sehr gering, da die Funktionen nur auf Werten der in ihren Signaturen angegebenen Typen arbeiten. Zum Beispiel müssen zum Visualisieren von Mengen von Personen,

Studenten und Adressen jeweils eigene Funktionen erstellt werden. In monomorphen Datenbanksprachen, wie z.B. DBPL, ist es daher unumgänglich, für die Implementation generischer Funktionen auf untypisierte Zeiger und byte-orientierte Strukturen auszuweichen. Auch in anderen Kombinationen, z.B. C mit den kommerziellen Datenbanksystemen INGRES oder ORACLE, kann nicht auf untypisierte Zeiger verzichtet werden. Solche Programme können nicht komplett statisch typüberprüft werden, da, wenn überhaupt, erst zur Laufzeit Typinformationen über die Strukturen zur Verfügung stehen, auf die die Zeiger verweisen.

Mit Hilfe des Konzepts des Polymorphismus kann dieser Zielkonflikt gelöst werden. Nach [Cardelli, Wegner 85] kann zwischen *ad hoc-* und *universellem Polymorphismus* unterschieden werden. In TL ist das Konzept des universellen Polymorphismus zu finden. Dieser kann weiter in *parametrischen* und *Subtyp-Polymorphismus* unterteilt werden.

In TL wird eine Funktion zu einer generischen (parametrisch polymorphen) Funktionen, indem ihre Signatur durch einen oder mehrere Typparameter erweitert wird. In TL können auch solche Funktionen typsicher implementiert werden. Der Programmierer implementiert den Funktionsrumpf nur einmal. Die verschiedenen Implementierungen leiten sich aus den verschiedenen Parametrisierungen ab. Für die Datenvisualisierung bedeutet dies, daß nur eine Funktion zum Anzeigen beliebiger Instanzen eines Kollektionstyps implementiert werden muß. Zum Beispiel können mit einer Funktion zum Visualisieren von Listen, durch eine entsprechende Parametrisierung, sowohl Listen von Adressen als auch Listen von Personen visualisiert werden.

Auch das in TL vorhandene Konzept des Subtyp-Polymorphismus unterstützt die Wiederverwendbarkeit von Funktionen. Zum Beispiel arbeitet eine Funktion zum Anzeigen einer Liste von Personen auch auf einer Liste von Studenten, wenn die folgende Subtypbeziehung gilt: *Student <:Person*.

Im Tycoon-System gibt es keine sprachliche Unterscheidung zwischen persistenten und transienten Daten. Das Kriterium für die Persistenz ist die Erreichbarkeit [Atkinson, Bunemann 87]. Die orthogonale Einführung dieses Konzeptes erlaubt nicht nur die persistente Speicherung von Daten, sondern auch die Speicherung von immer wiederkehrenden Benutzeranfragen und Funktionen zur Visualisierung der Anfrageergebnisse.

Mit den von TL angebotenen Funktionen höherer Ordnung, dem parametrischen Polymorphismus und dem reichen Typsystem soll versucht werden, die repetitiven Programmieraufgaben, wie das Erstellen von Funktionen zur formatierten und sicheren Dateneingabe sowie zur Ausgabe von Datenobjekten auf unterschiedlichen Ausgabemedien und für unterschiedliche Benutzerklassen, zu vereinfachen.

1.2 Struktur der Arbeit

In diesem Abschnitt wird ein kurzer Überblick über den Aufbau der vorliegenden Arbeit gegeben. Dazu werden die jeweils in den Kapiteln vorgestellten Ansätze und Werkzeuge kurz beschrieben.

Entwicklungswerkzeuge für graphische Benutzerschnittstellen: Nach Erläuterung von Motivation, Zielsetzung und Struktur der Arbeit in diesem Kapitel beginnt die Arbeit in Kapitel 2 mit der Vorstellung verschiedener Ansätze, die die aufwendige Entwicklung von datenintensiven Anwendungen mit graphischer Benutzerschnittstelle unterstützen und vereinfachen sollen. Untersucht wird dabei, inwieweit sich die einzelnen Ansätze dazu eignen, als generischer Dienst in eine Programmiersprache (hier TL) integriert zu werden. Das Hauptaugenmerk liegt dabei auf der dynamischen Erzeugung und Manipulation von graphischen Benutzerschnittstellen. Abgeschlossen wird dieses Kapitel durch eine klassifizierende Zusammenfassung der vorgestellten Ansätze.

Programmierumgebungen für datenintensive Anwendungen: Drei verschiedene Ansätze zur Einbindung eines Dienstes zur dynamischen Erzeugung und Manipulation von graphischen Benutzerschnittstellen in eine Programmierumgebung werden in Kapitel 3 miteinander verglichen. Ziel dieses Kapitels ist die Darstellung der sprachlichen und architektonischen Vorteile der Tycoon-Entwicklungsumgebung gegenüber anderen Entwicklungsumgebungen. Diese Umgebung bildet die Grundlage für die Implementation der Schnittstelle zum NeWS Toolkit und die Implementation der generischen Editorbibliothek.

Datenvisualisierung in bestehenden Systemen: In Kapitel 4 werden verschiedene neuere kommerzielle Systeme und Forschungssysteme hinsichtlich ihrer Möglichkeiten zur Datenvisualisierung analysiert. Aus den Ergebnissen dieser Untersuchung ergeben sich Anforderungen an die typsichere generische Datenvisualisierung in Tycoon. Im einzelnen werden die kommerziellen Datenbanksysteme DBPL, INGRES, O_2 und GemStone betrachtet. Ein weiterer Abschnitt ist der Programmiersprache VisualBASIC gewidmet. Von jedem System werden zunächst überblicksweise die Architektur und die vorrangigen Entwicklungsziele vorgestellt. Anschließend wird die Visualisierungskomponente genauer beschrieben und ein Programmbeispiel implementiert. Den Abschluß dieses Kapitels bildet eine Zusammenfassung der aus der Untersuchung gewonnenen Erkenntnisse.

Kommerzielle Dienste einer speziellen Anwendungsumgebung: Die am Arbeitsbereich DBIS in der OpenWindows-Anwendungsumgebung vorhandenen kommerziellen Dienste werden in Kapitel 5 vorgestellt. Die Umgebung bildet die Entwicklungsplattform für die in den nachfolgenden Kapiteln vorgestellten generischen Bibliotheken. Die Zusammensetzung der Entwicklungsplattform ist zusätzlich durch das Ergebnis der in Kapitel 2 beschriebenen Suche nach einem geeigneten Entwicklungswerkzeug für graphische Benutzerschnittstellen beeinflußt. Das Zusammenspiel dieser kommerziellen Dienste wird im letzten Abschnitt dieses Kapitels beschrieben.

Graphische Benutzerschnittstellen in Tycoon: In den Kapiteln 6 und 7 werden die beiden im Rahmen dieser Arbeit implementierten Bibliotheken, *newsenv* und *editenv*, vorgestellt. Dieses Kapitel stellt ausführlich die Konzepte und die Implementation des *newsenv*, einer Bibliothek zur dynamischen Erzeugung und Manipulation graphischer

Benutzerschnittstellen unter OPEN LOOK in Tycoon, vor. Die Bibliothek baut auf den Ergebnissen der Untersuchungen in Kapitel 2 auf. Die umfangreiche Bibliothek bietet Schnittstellen zum NeWS Toolkit, zum Wire Service und zum Jot. Diese sind unter anderem eine Voraussetzung für die Implementation der in Kapitel 7 vorgestellten typsicheren generischen Editoren. Im Anschluß an die Vorstellung der Bibliothek werden einige praktische Anwendungsgebiete des *newsenv* aufgezeigt.

Datenvisualisierung in Tycoon: Nachdem in den vorherigen Kapiteln alle Voraussetzungen für die Datenvisualisierung in Tycoon beschrieben worden sind, wird in Kapitel 7 auf die Implementation der typsicheren generischen Editoren eingegangen. Zunächst wird dafür die Gestaltung der graphischen Benutzerschnittstelle anhand von mehreren Abbildungen ausführlich beschrieben. Es folgt die Beschreibung der Anwendungsprogrammierung mit der Editorbibliothek anhand eines Beispiels. In einem weiteren Abschnitt werden ausgewählte Beispiele aus den zur Bibliothek gehörenden Schnittstellen und Modulen erläutert. Es folgt abschließend ein Abschnitt über die Erweiterungsmöglichkeiten der Bibliothek.

Bewertung und Ausblick: Das letzte Kapitel faßt alle in dieser Arbeit gewonnenen Ergebnisse zusammen. Dazu wird in einem ersten Abschnitt die Vorgehensweise bei der Entwicklung der Editorbibliothek unter software-ergonomischen und -technischen Gesichtspunkten beschrieben. Anschließend werden die beim Vergleich der Visualisierungskomponenten bestehender Systeme in Kapitel 4 gewonnenen Erkenntnisse, die als Grundlage für die Implementation der Bibliothek dienten, mit den in der Implementation tatsächlich erreichten Zielen verglichen. Den Abschluß dieses Kapitels und damit der Arbeit bildet eine kurze Übersicht über offene Probleme und mögliche Lösungsalternativen.

2. Entwicklungswerkzeuge für graphische Benutzerschnittstellen

Wie in der Einleitung kurz angeführt, steht der hohe Aufwand für die Erstellung von Anwendungsprogrammen mit graphischer Benutzerschnittstelle im krassen Gegensatz zu den angestrebten Vereinfachungen bei deren Bedienung. Die Erstellung von Anwendungsprogrammen mit graphischer Benutzerschnittstelle ist erheblich schwieriger als die Erstellung anderer Software. Dies hat verschiedene Ursachen. Sie liegen vor allem im iterativen Entwicklungsprozeß, den hohen Geschwindigkeits- und Stabilitätsanforderungen, der Mehrprozeßfähigkeit, der Notwendigkeit zur Abhandlung asynchroner Ereignisse und den Portabilitätsanforderungen [Myers 92; Myers, Rosson 92; Fähnrich et al. 92; Meyer 94].

Aus den genannten Gründen sind verschiedene Ansätze für Erstellung der graphischen Benutzerschnittstelle von Anwendungsprogrammen entwickelt worden. Als Hauptgrund für den Einsatz solcher Entwicklungswerkzeuge ist die erhoffte Einsparung bei der Entwicklungszeit der graphischen Benutzerschnittstelle zu nennen. Weitere Vorteile sind z.B. die schnelle Erstellung von Prototypen und die erreichbare Konsistenz über verschiedene Anwendungen hinweg. Durch die Trennung von Benutzerschnittstelle und Anwendungsprogramm wird zusätzlich die Wartung der beiden Komponenten vereinfacht.

Dadurch ist insbesondere der Bereich der Methoden und Werkzeuge zur Erstellung von Anwendungsprogrammen mit graphischer Benutzerschnittstelle in den Mittelpunkt der Softwareentwicklung gerückt. Mittlerweile gibt es sehr verschiedene Ansätze, um die aufwendige Entwicklung von Programmen mit graphischer Benutzerschnittstelle zu unterstützen und zu vereinfachen. Angefangen von einfachen Toolkits bis hin zu systemübergreifenden Bibliotheken werden in diesem Kapitel einige Ansätze vorgestellt [Heller et al. 91; Sheldon et al. 91; Markus 92; Myers 92; Myers, Rosson 92; Lee 93]. Untersucht wird hierbei, inwieweit sich die einzelnen Ansätze dazu eignen, als generischer Dienst in eine Programmiersprache (hier TL) integriert zu werden. Das Hauptaugenmerk liegt dabei auf der Möglichkeit zur dynamischen Erzeugung von graphischen Benutzerschnittstellen. Abgeschlossen wird dieses Kapitel von einer klassifizierenden Zusammenfassung der vorgestellten Ansätze.

Die in den folgenden Abschnitten als Beispiele erwähnten Produkte sind in Tabelle 2.1 klassifiziert zusammengefaßt. Es sei noch auf Kapitel 5 hingewiesen, in dem einige der kommerziellen Produkte für Unix-Systeme ausführlicher beschrieben werden.

	Unix		Macintosh	PC
Fenstersysteme	X11	OpenWindows	Finder	Windows
Stilrichtlinien	OSF/Motif	OPEN LOOK	UI Guidelines	Design Guide
Bibliotheken	Xlib		QuickDraw	Win32
Toolkits	Xt	NeWS Toolkit	Toolbox	SDK
Ressource-Editoren	UIL		ResEdit	SDKPaint Dialog Editor
Quelltext-Programmgeneratoren		Devguide	AppMaker	AppWizzard
Interpreter	TCL		HyperCard	VisualBASIC
Klassenbibliotheken			MacApp	MFC, OWL
Systemübergreifende Bibliotheken	XVT, ZINC, StarView			
4GL-Systeme	INGRES/Windows 4GL		4th Dimension	Access

Tabelle 2.1: Entwicklungswerkzeuge für graphische Benutzerschnittstellen

2.1 Fenstersysteme

Fenstersysteme (*window systems*) bilden die Grundlagen von direktmanipulativen Benutzerschnittstellen. Der Zugriff auf Fenstersysteme erfolgt im einfachsten Fall direkt über die dazugehörigen Bibliotheken. Die Erstellung von Benutzerschnittstellen auf dieser Ebene erfordert jedoch einen enorm hohen Programmieraufwand, da alle höheren Komponenten aus vielen einzelnen Funktionsaufrufen zusammengesetzt werden müssen.

Die Funktionalität der Bibliotheken hängt eng mit den Aufgaben der Fenstersysteme zusammen. Die grundlegenden Aufgaben werden im folgenden beschrieben. In allen Fenstersystemen werden die Eingaben über nicht-sequentielle Eingabegeräte, wie Maus oder Tastatur, abgewickelt. Das Fenstersystem leitet die Eingaben an die Anwendungen weiter. Die in Fenstersystemen verwendeten Konzepte und Modelle für die Bilderzeugung, die Wiederherstellung beschädigter Bildausschnitte, die Implementation und die Möglichkeiten zur dynamischen Erweiterung unterscheiden sich zum Teil erheblich voneinander [Krogull 92]. Ältere Fenstersysteme basieren auf einem pixelorientierten Bilderzeugungsmodell (*pixel oriented model*). In solchen Systemen wird die Farbe für jedes Pixel in einem festen Koordinatensystem bestimmt und ausgegeben. In neueren Fenstersystemen werden schablonenorientierte Bilderzeugungsmodelle (*stencil-paint imaging model*) verwendet. Dabei wird die Farbe nicht für einzelne Pixel bestimmt, sondern durch *Füllen* von Schablonen für Flächen. Dadurch sind schablonenorientierte Modelle von Auflösung und Farbanzahl des Ausgabegeräts unabhängig.

Auf den untersuchten Plattformen existieren u.a. die folgenden Fenstersysteme mit den dazugehörigen Stilrichtlinien und Bibliotheken: X11 mit OSF/Motif und der Xlib, Open-

Windows mit OPEN LOOK (vgl. Abschnitt 5.3) und der Xlib für Unix-Systeme, der Finder mit den UI Guidelines und QuickDraw für den Macintosh sowie Microsoft Windows mit dem Design Guide und Win32 für PC's.

2.2 Toolkits

Toolkits sind Bibliotheken, die von den Bibliotheken der Fenstersysteme abstrahieren. Toolkits stellen die durch die Stilrichtlinien (*style guides*) definierten Schnittstellenkomponenten (*widgets*), wie Menüs (*menus*), Knöpfe (*buttons*), Rollbalken (*scroll bars*), Dialogboxen usw. zur Verfügung. Zusätzlich vereinfacht sich mit Toolkits auch die Ereignisverwaltung gegenüber den Bibliotheken der Fenstersysteme.

Sowohl die Bibliotheken der Fenstersysteme als auch die Toolkits stellen i.d.R. keine höheren Dialogkomponenten, wie z.B. zur Dateiauswahl (*file selection*) oder zur Datensatzbearbeitung, zur Verfügung. Solche Dialogkomponenten müssen aus einzelnen Schnittstellenkomponenten selbst aufgebaut werden.

In Abschnitt 5.4 wird mit dem NeWS Toolkit für OPEN LOOK eine solche Bibliothek ausführlicher vorgestellt.

2.3 Ressource-Editoren

Ressource-Editoren (*resource editors*) ermöglichen die Erstellung von Benutzerschnittstellen durch *direkte Manipulation* [Ilg, Ziegler 88; Krogull 92]. Die Schnittstelle eines Anwendungsprogramms wird interaktiv aus den in einem Werkzeugkasten zur Verfügung gestellten Schnittstellenkomponenten aufgebaut. Der Vorteil des bei dieser Art des Entwurfs angewandten *WYSIWYG*-Prinzips (*What You See Is What You Get*) ist, daß bereits während der Erstellung der Benutzerschnittstelle das Aussehen validiert werden kann. Das Resultat ist eine Ressource-Datei. Zusätzlich werden die Definitionen der verwendeten Konstanten und Bezeichner für Schnittstellenkomponenten in Form einer *Include*-Datei ausgegeben, die für die Kopplung der Elemente der graphischen Benutzerschnittstelle mit dem Kode des Anwendungsprogramms erforderlich sind.

Diese Vorgehensweise verringert zwar den Programmieraufwand erheblich, hat aber den Nachteil, daß nur weitgehend statische Benutzerschnittstellen erzeugt werden können. Sobald sich die Struktur der anzuzeigenden Objekte ändert, muß auch die Ressource-Datei geändert werden.

2.4 Quelltext-Programmgeneratoren

Quelltext-Programmgeneratoren (*interface builder*) ermöglichen wie Ressource-Editoren die interaktive Erstellung von Benutzerschnittstellen aus den in einem Werkzeugkasten zur

Verfügung gestellten Schnittstellenkomponenten. Darüber hinaus erlauben Quelltext-Programmgeneratoren auch, Teile des Programmablaufs des zu erstellenden Anwendungsprogramms festzulegen und in einem Test-Modus zu simulieren.

Aus der interaktiv erstellten Benutzerschnittstelle können anschließend Quelltext-Dateien generiert werden. Dazu zählen die Ressource-Datei, eine Datei mit dem Quelltext des erstellten Anwendungsprogramms in Hochsprachenform, z.B. C oder C++, sowie eine *Make*-Datei, in der alle für die Erzeugung eines lauffähigen Programms mit einem Übersetzer (*compiler*) und Binder (*linker*) erforderlichen Schritte und die Abhängigkeiten der einzelnen Dateien voneinader enthalten sind. Der Programmierer muß noch die Anwendungsfunktionen spezifizieren.

Auch Quelltext-Programmgeneratoren eignen sich nur für die Erstellung von weitgehend statischen Benutzerschnittstellen. In Abschnitt 5.5.2 wird mit dem Devguide von Sun Microsystems ein Quelltext-Programmgenerator für OPEN LOOK vorgestellt.

2.5 Interpreter-Entwicklungsumgebungen

Interpreter besitzen gegenüber Übersetzern den Vorteil, daß die Entwicklungszeit bei der Erstellung von Anwendungsprogrammen verkürzt wird. Alle eingegebenen Programmzeilen werden sofort auf Korrektheit überprüft. Ist eine eingegebene Zeile fehlerfrei, wird sie sofort in eine für den Computer ausführbare Form umgewandelt. Durch inkrementelles Binden von Bibliotheken wird die Entwicklungszeit zusätzlich verkürzt.

Zur Laufzeit dagegen sind Interpreter den Übersetzern unterlegen, da ein Laufzeitinterpreter (*runtime interpreter*) bei jeder Programmausführung die Zeilen wieder neu interpretieren muß. Abhilfe schaffen hier zusätzliche Übersetzer, die nach Abschluß der Entwicklung selbständig ablauffähige Programme erzeugen.

Zu einer Interpreter-Entwicklungsumgebung gehört i.d.R. ein Ressource-Editor, mit dem die Benutzerschnittstelle interaktiv erzeugt wird. Die Entwicklungsumgebung erzeugt daraus automatisch Prozedurköpfe für die verwendeten Schnittstellenkomponenten. Damit entfällt für den Anwendungsprogrammierer die Notwendigkeit explizit eine Verbindung zwischen einer Schnittstellenkomponente und den Prozeduren, die ausgeführt werden sollen, wenn die Schnittstellenkomponente manipuliert wird, herzustellen.

Eine z.Zt. sehr verbreitete Entwicklungsumgebung ist VisualBASIC (vgl. Abschnitt 4.2.5) von Microsoft. Es vereinfacht die Erstellung von Windows-Programmen gegenüber anderen Ansätzen, wie z.B. C und SDK (*Software Development Kit*), erheblich.

2.6 Klassenbibliotheken

Klassenbibliotheken vereinfachen die Entwicklung von Anwendungsprogrammen mit graphischer Benutzerschnittstelle, indem sie objektorientierten Sprachen, wie z.B. C++ oder

Smalltalk, spezielle Klassen zur Verfügung stellen. Die Klassen definieren Schnittstellenkomponenten und dazugehörige Methoden zur Manipulation der Instanzen. Die Klassen beinhalten die Funktionalität, die ansonsten nur durch eine Vielzahl von Bibliotheksaufrufen erreichbar ist. Durch die Zusammenfassung der immer wieder benötigten Funktionsaufrufe verringert sich die Fehlerquote, da weniger Programmkode geschrieben werden muß.

Entsprechen die von einer Klasse zur Verfügung gestellten Instanzen nicht den gestellten Anforderungen, so kann eine Subklasse erzeugt werden, in der die entsprechenden Eigenschaften modifiziert bzw. erweitert werden. Alle Instanzen dieser Subklasse erben dabei grundsätzlich alle Eigenschaften der Klasse, außer denen, die explizit überschrieben werden. Dadurch müssen jeweils nur Details geändert werden, während alle anderen Eigenschaften weiterhin automatisch konsistent sind.

Ein *Class Browser* ermöglicht die einfache Übersicht über die im System vorhandenen Klassen mit den dazugehörigen Vererbungsbeziehungen und Methoden.

2.7 Systemübergreifende Bibliotheken

Systemübergreifende Bibliotheken (*virtual toolkits*) bieten identische Programmierschnittstellen und eine identische Sprache für die Beschreibung von Benutzerschnittstellen für mehrere Systemplattformen an und ermöglichen damit die einfache Portierung von Anwendungsprogrammen auf verschiedene Fenstersysteme [Isau 93]. Anwendungsentwickler können nach diesen Spezifikationen entwickelte portable Quelltexte auf allen unterstützten Systemen ohne Modifikation in ausführbare Programme umwandeln.

Erreicht wird diese Portabilität durch eine Bibliothek, die zwischen dem Anwendungsprogramm und dem Zielfenstersystem arbeitet. Eine derartige Bibliothek wandelt alle Funktionsaufrufe der systemunabhängigen Programmierschnittstelle in die entsprechenden Funktionsaufrufe des Fenstersystems um. In der Gegenrichtung werden alle vom Fenstersystem erzeugten Ereignisse in eine systemunabhängige Darstellung umgewandelt.

Allerdings ergeben sich aus der Verschiedenartigkeit der von den einzelnen Fenstersystemen unterstützten Funktionen Probleme. Die einfachste Lösung ist der *Schnittmengenansatz*, bei dem nur Funktionen angeboten werden, die auf allen unterstützten Systemen verfügbar sind und somit den kleinsten gemeinsamen Nenner darstellen. Das führt zu einer Beschränkung des Funktionsumfangs auf die Funktionen des schwächsten durch die Bibliothek unterstützten Fenstersystems.

Dagegen werden beim *Obermengenansatz* alle Funktionen und Elemente aller unterstützten Fenstersysteme durch die Bibliothek verfügbar gemacht. Fehlt in einem System eine auf einem anderen System verfügbare Funktion, so wird diese Funktion durch die Bibliothek nachgebildet. Diese Vorgehensweise erlaubt es auch, den Funktionsumfang aller Bibliotheken um komplexere Funktionen zu erweitern.

Systemübergreifende Bibliotheken haben aber auch Nachteile. Zum Beispiel wird die Unabhängigkeit von Systemplattformen durch eine neue Abhängigkeit vom Bibliotheksherstellers teilweise wieder kompensiert.

2.8 4GL-Systeme

Sprachen der vierten Generation (*fourth generation languages*, 4GL's) sind meist Bestandteil eines Datenbanksystems. In solchen Systemen sind oft Editoren zur Gestaltung der graphischen Benutzerschnittstelle durch direkte Manipulation zu finden. Im Gegensatz zu Quelltext-Programmgeneratoren sind sie jedoch speziell auf den Einsatz in Datenbankumgebungen abgestimmt. In diesem Anwendungsgebiet weisen 4GL's eine erweiterte Funktionalität auf, die über das bei Quelltext-Programmgeneratoren übliche Maß hinausgeht. Zum Beispiel können die im Datenwörterbuch (*data dictionary*) vorhandenen Typinformationen dazu genutzt werden, um automatisch Bildschirmmasken zu erstellen. Durch diese Spezialisierung treten Defizite in anderen Bereichen auf, für deren Einsatz sie nicht speziell ausgerichtet sind.

Neben der Gestaltung der graphischen Benutzerschnittstelle bieten sie die Möglichkeit, den Programmablauf des zu erstellenden Anwendungsprogramms festzulegen. Dabei stehen in Abhängigkeit von der Mächtigkeit der 4GL-Programmiersprache unterschiedliche Konzepte zur Verfügung. Nachteile solcher Systeme sind das oft eingeschränkte Datenmodell, die fehlende Möglichkeit zur Integration neuer Dienste und der fehlende Polymorphismus.

In Abschnitt 4.2.2 wird mit INGRES/Windows 4GL ein 4GL-System für unterschiedliche Hard- und Softwareplattformen vorgestellt.

2.9 Zusammenfassung und Klassifizierung

In diesem Abschnitt werden die vorgestellten Ansätze nach zwei Kriterien klassifiziert. Dazu wird in Abbildung 2.1 ein zweidimensionales Klassifizierungsschema vorgestellt. Auf der vertikalen Achse sind die durch das Werkzeug unterstützten Architekturschichten aufgetragen und auf der horizontalen Achse ist der Abstraktionsgrad des Werkzeuges zu finden [Fähnrich et al. 92].

Zunächst werden die auf der vertikalen Achse aufgetragenen Architekturschichten beschrieben. Die Präsentationskomponente verwaltet zum einen die externen Darstellungen der Benutzerschnittstelle und ermöglicht zum anderen die Interaktion mit dem Anwender. Die Dialogsteuerung definiert den Ablauf des Dialoges zwischen Anwender und Anwendung. Die Anwendungsschicht enthält die eigentlichen Funktionen der Anwendung sowie die Objekte, die mit diesen Funktionen bearbeitet werden.

Auf der horizontalen Achse wächst von links nach rechts der Abstraktionsgrad des Werkzeuges. Programmierwerkzeuge wie Bibliotheken und Toolkits besitzen den niedrigsten Abstraktionsgrad. Sie werden von allgemeinen Programmiersprachen aus benutzt. Ähnlich sieht es für Klassenbibliotheken und systemübergreifende Bibliotheken aus. Der Vorteil dieser Ansätze ist der große Funktionsumfang und die daraus resultierende Mächtigkeit und Flexibilität der Werkzeuge. Dieser Vorteil kann jedoch bei unzureichender Dokumentation auch zum Nachteil werden, da vor allem Anfänger vor der Mächtigkeit und dem enormen

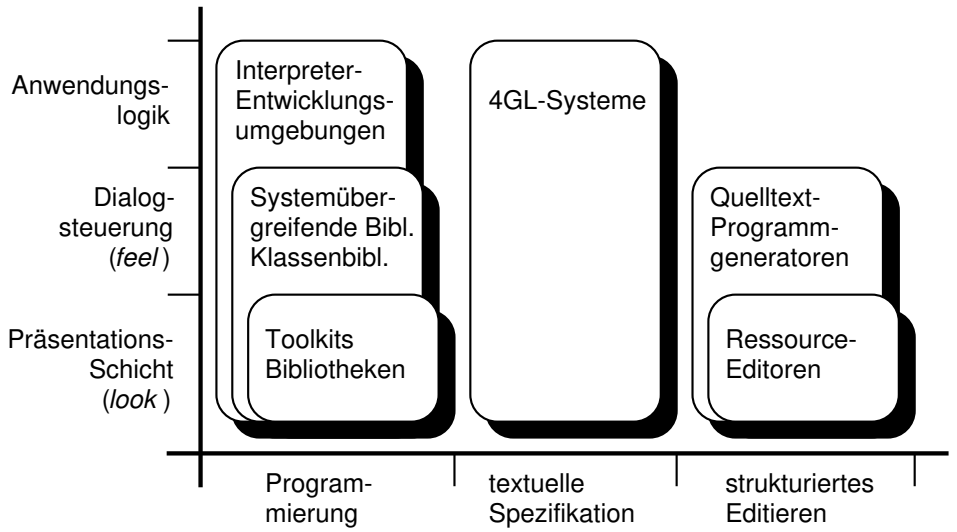


Abbildung 2.1: Klassifizierungsschema für Entwicklungswerkzeuge (nach [Fähnrich et al. 92])

Funktionsumfang resignieren. Interpreter-Entwicklungsumgebungen haben den Vorteil der vollständigen Integration aller drei Architekturschichten.

Die textuelle Spezifikation von vollständigen Benutzerschnittstellen bzw. der Präsentationskomponente wird durch spezielle Beschreibungssprachen ermöglicht. Solche Sprachen enthalten Konstrukte für die Beschreibung von Schnittstellenkomponenten, z.B. Fenstern, Menüs, Knöpfen und Texteingabefeldern, und ihren Darstellungsattributen, wie Position, Größe und Farbe.

Strukturierte Editoren vereinfachen die Erstellung von graphischen Benutzerschnittstellen weiter. Für die direktmanipulative Entwicklung der Präsentationskomponente gibt es Ressource-Editoren, mit deren Hilfe die Auswahl und die Attributierung der Schnittstellenkomponenten vorgenommen werden kann. Darüber hinaus erleichtern spezielle Quelltext-Programmgeneratoren zusätzlich die Spezifikation der Dialogabläufe und die Verwaltung der vom Dialogmodell abhängigen Sprachkonstrukte in einem ereignisorientierten Modell [Fähnrich et al. 92].

Zusammenfassend kann festgestellt werden, daß nur 4GL-Systeme für die Entwicklung der graphischen Benutzerschnittstelle von datenintensiven Anwendungen eine ausreichende Unterstützung bieten. Dieses System ist jedoch für die Integration als generischer Dienst in eine Programmiersprache ungeeignet, da es alle drei vertikalen Ebenen (vgl. Abbildung 2.1) selbst abdeckt.

Für die Integration als generischer Dienst in eine allgemeine Programmiersprache ist die Kombination aus Klassenbibliothek und Quelltext-Programmgenerator zu empfehlen. Klas-

senbibliotheken besitzen die bekannten Vorteile objektorientierter Systeme (vgl. [Watt 90]). Wird zusätzlich eine portable Oberfläche benötigt, so ist die Verwendung von systemübergreifenden Bibliotheken anzuraten. Benutzerschnittstellen können durch Anbindung der Bibliothek an eine Programmiersprache dynamisch erzeugt und verändert werden. Ein Quelltext-Programmgenerator erlaubt darüber hinaus, statische Teile der Benutzerschnittstelle auf einfache Weise zu erstellen. Die Anbindung erfolgt auch hier über eine Bibliothek.

Die für die Implementation verwendete Entwicklungsumgebung besteht aus den folgenden Komponenten:

- Tycoon-System (vgl. Abschnitt 3.3)
- NeWS Toolkit (vgl. Abschnitt 5.4)
- Devguide (vgl. Abschnitt 5.5.2)

Eine Schnittstelle vom Tycoon-System zum NeWS Toolkit ermöglicht die dynamische Erzeugung von graphischen Benutzerschnittstellen. Darüber hinaus ist es über diese Schnittstellen auch möglich, statische, mit dem Devguide erzeugte Benutzerschnittstellen anzubinden.

3. Programmierumgebungen für datenintensive Anwendungen

Bei Informationssystemen zur Unterstützung datenintensiver Anwendungen wird zwischen System- und Anwendungskomponenten unterschieden. Sie sind durch hohe Qualitätsanforderungen an die Systemprogramme (Korrektheit, Effizienz, Mehrbenutzerfähigkeit und Fehlererholung), den großen Umfang und die Langlebigkeit der entwickelten Anwendungssoftware sowie eine starke Spezialisierung auf immer wiederkehrende algorithmische und strukturelle Muster (mengenorientierte Anfragen, Konsistenzbedingungen, generische Datenstrukturen) gekennzeichnet [Matthes 93; Atkinson et al. 92; Stonebraker et al. 90].

Technologische und marktwirtschaftliche Entwicklungen erfordern den zunehmenden Einsatz von Informationssystemen für neuartige (graphische, geographische, statistische, multimediale) Informationsstrukturen in substantiell veränderten (interaktiven, vernetzten, verteilten) Gesamtsystemen [Matthes 93; Blaser 90; Cattell 91; Balzer, Mylopoulos 91].

Um diesen erweiterten Anforderungen spezieller Anwendungsklassen und Systemumgebungen gerecht zu werden, kommen in der Praxis zahlreiche Systeme und Werkzeuge zur Erbringung *generischer Dienste* zum Einsatz. Klassische Beispiele solcher generischen Diensterbringer, die eine wichtige Rolle bei der Konstruktion interaktiver Informationssysteme spielen, sind Programmiersprachen, Datenbankprogrammiersprachen sowie textuelle und graphische Benutzerschnittstellen. Neben Datenvisualisierungswerkzeugen werden zahlreiche Werkzeuge, etwa zur Textrecherche, Datendeduktion und Transaktionskontrolle in verteilten Systemen, kommerziell angeboten.

Die Vielzahl spezialisierter Werkzeuge, Systemerweiterungen und Modellierungsvarianten führt zu einer hohen Komplexität und Heterogenität der mit ihnen entwickelten Gesamtsysteme und untergräbt damit wesentliche Qualitäten, die zur Einführung integrierter Informationssysteme geführt haben. Trotz großer Fortschritte bei der Lösung isolierter Problemklassen ist daher eine generelle Unzufriedenheit mit der Qualität heutiger Informationssysteme festzustellen (mangelnde Interoperabilität, Adaptierbarkeit und Offenheit) [Matthes 93; Appelrath et al. 92].

Die Qualität zukünftiger datenintensiver Anwendungen wird wesentlich durch die Flexibilität, Effizienz und Korrektheit des Zusammenspiels verschiedenartiger, unabhängig entwickelter Werkzeuge und Dienste bestimmt werden.

In den nachfolgenden Abschnitten werden drei unterschiedliche Ansätze zur Einbindung eines Dienstes zur Ansteuerung der Benutzerschnittstelle in Datenbankprogrammierumgebungen vorgestellt und bewertet. Ziel dieser Beschreibung ist es, die Vorteile der Tycoon-Entwicklungsumgebung hinsichtlich der Anbindung externer Dienste zu verdeutlichen. In Kapitel 6 wird die Integration eines Dienstes zur dynamischen Erzeugung und Manipulation von graphischen Benutzerschnittstellen vorgestellt.

3.1 Lose Kopplung durch Wirtsspracheneinbettung

Die Mehrheit aller Informationssysteme wird gegenwärtig in Programmiersprachen der dritten Generation z.B. COBOL [ANSI 74], FORTRAN [ANSI 78], C [Kernighan, Ritchie 77] und Pascal [Wirth 71] implementiert. Alle datenbankrelevanten Funktionen werden über Programmierschnittstellen zu kommerziellen Datenbanksystemen, wie z.B. zu INGRES [Ingres 90e] und ORACLE [Bronzite 89], angesprochen. Eine ähnliche Situation ergibt sich bei der Anbindung von Benutzerschnittstellen und der Visualisierung von Daten.

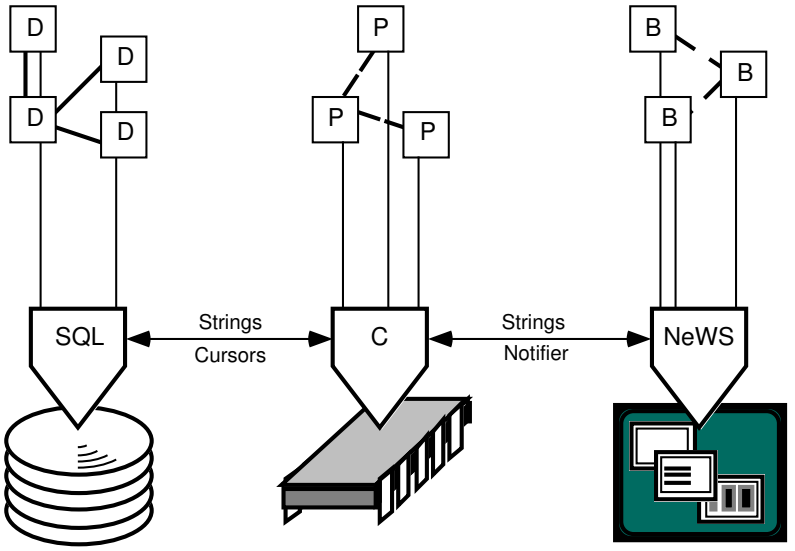


Abbildung 3.1: Schmale Dienstschnittstellen auf niedrigem Abstraktionsniveau [Matthes 93]

Solche *Dienste* sind auf einem niedrigen Abstraktionsniveau über eine schmale Schnittstelle an die Sprache gebunden (vgl. Abbildung 3.1). Über diese Schnittstellen werden i.d.R. Zeichenketten ausgetauscht. Der Kontrolltransfer zwischen Benutzerschnittstelle und Anwendungsprogramm wird über Benachrichtigungsprozeduren (*notifier*) und dazugehörige Aktionen (*callbacks*) abgewickelt.

Wie durch verschiedenartige Linienmuster zwischen den Datenbank-, Programm- und Bildschirmobjekten in Abbildung 3.1 angedeutet, muß der Programmierer für die Objekte von verschiedenen Servern verschiedene Benennungs-, Lebensdauer- und Bindungskonzepte erlernen. Ebenso muß er Bindungen zwischen den Objekten auf verschiedenen Medien durch spezielle Namenskonventionen (Hostvariable, Datenbankvariable, Identifikator für ein Bildschirmobjekt) realisieren.

Der Vorteil, der sich aus der *losen Kopplung* ergibt, ist die durch sie erreichbare Offenheit der Anwendungsumgebung. Bei Bedarf ist die Integration eines neuen Diensterbringers als Bibliothek in die Programmierumgebung ohne größere Probleme möglich.

3.2 Systemintegration in Datenbankprogrammiersprachen

In Datenbankprogrammiersprachen vereinfacht sich das Programmierszenario (vgl. Abbildung 3.2). In diesen Umgebungen ist ein transparenter Zugriff auf kurz- und langlebige Datenobjekte möglich. Desgleichen können kurz- und langlebige Datenobjekte uniform benannt werden.

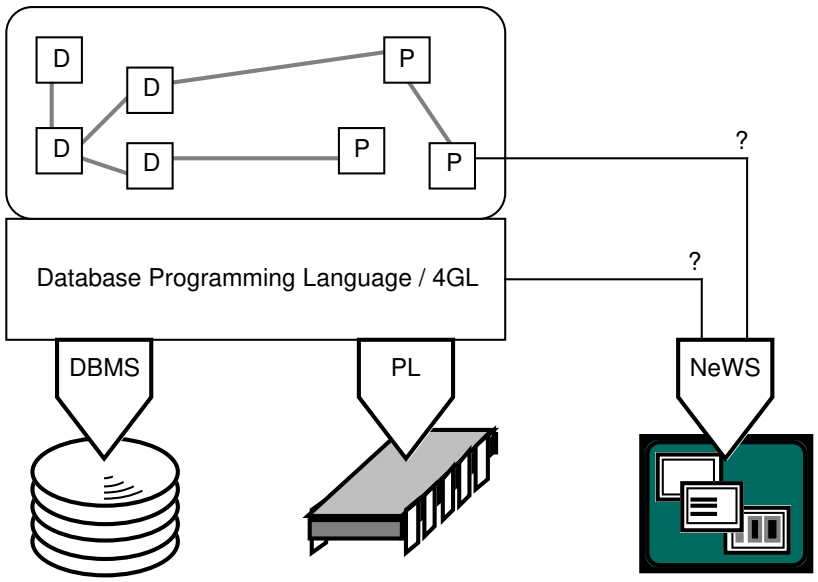


Abbildung 3.2: Geschlossene Programmierumgebungen zur Nutzung vordefinierter Dienste [Matthes 93]

Neben problemadäquateren Datenmodellierungsstrukturen, wie z.B. Verbunde und Mengen, stehen in Datenbankprogrammiersprachen neuartige Kontrollstrukturen, wie z.B. generische

Operationen zur Iteration über komplexe Objekte, zur Verfügung. Weitere generische Operationen dienen dem Suchen in und dem Anzeigen von komplexen Objekten. Da innerhalb des Systems semantisch reichhaltigere Strukturen verwaltet werden können, stehen für Datenbankprogrammiersprachen im allgemeinen wesentlich bessere Werkzeuge zur Programmentwicklung und -wartung zur Verfügung (Debugger, Datenwörterbücher und Browser). Zum Beispiel können die Informationen aus den Datenwörterbüchern dazu verwendet werden, dynamische Bildschirmrepräsentationen für Datenobjekte zu generieren.

Es muß jedoch zwischen den für den Anwendungsprogrammierer sichtbaren Abstraktionen, z.B. Relationen oder Formulardefinitionen, und den sie implementierenden Datenstrukturen und Algorithmen unterschieden werden. Zur strikt typisierten Anwendungsprogrammierung werden relationale Sprachen wie INGRES/Windows 4GL [Ingres 90b] (vgl. Abschnitt 4.2.2), DBPL [Schmidt, Matthes 92] (vgl. Abschnitt 4.2.1), PL-SQL [Oracle 91] und Informix 4GL [Informix Software 86] sowie objektorientierte Systeme wie O_2 [Bancilhon et al. 92] (vgl. Abschnitt 4.2.3) und GemStone [Butterworth et al. 91] (vgl. Abschnitt 4.2.4) verwendet. Hingegen werden zur weitgehend typunsicheren Systemimplementation Sprachen wie C oder Modula-2 [Wirth 85; Schröder 91] verwendet.

Nachteil des von Datenbankprogrammiersprachen verfolgten *built-in* Ansatzes [Matthes, Schmidt 91a] ist, daß die Integration neuer generischer Dienste ausschließlich durch den Entwickler des Systems vorgenommen werden kann. Zusätzlich gefährdet eine solche Erweiterung die Kompatibilität und Portabilität bereits existierender Anwendungen. Diese Problematik ist in Abbildung 3.2 durch den Zugriff auf einen nicht vordefinierten generischen Diensterbringer (NeWS) angedeutet.

Weitere Nachteile sind die gegenüber Programmiersprachen der dritten Generation oft mangelnden sprachlichen Qualitäten von Datenbankprogrammiersprachen und die fehlenden Konzepte für Modularisierung [Matthes, Schmidt 93].

3.3 Systemintegration im Tycoon-System

In den beiden vorherigen Abschnitten sind die Vor- und Nachteile existierender Sprachen der dritten und vierten Generation aufgezeigt worden. Mit dem Tycoon-System wird versucht, die Vorteile von Systemen der dritten und vierten Generation in einer Programmierumgebung zu vereinen [Matthes, Schmidt 93].

Das Tycoon-System ist eine moderne Datenbankprogrammierumgebung, die den sprachlichen und architektonischen Rahmen für eine flexible Definition und Integration generischer Dienste in offenen Systemumgebungen bereitstellt. Zur Definition neuer und zur Einbindung bestehender externer Dienste wird die algorithmisch vollständige, strikt typisierte und polymorphe Programmiersprache höherer Ordnung TL verwendet. Sie ermöglicht eine flexible Benennung, Bindung und Typisierung der für datenintensive Anwendungen relevanten Objekte und Dienste [Matthes 93].

Das Tycoon-Modell weicht erheblich von dem Modell existierender Datenbankprogrammiersprachen ab. Wie in Abbildung 3.3 dargestellt, wird der gesamte Prozeß der Integration, Erweiterung, Spezialisierung, Nutzung und Definition generischer Dienste in einem unifizierenden sprachlichen Rahmen abgewickelt.

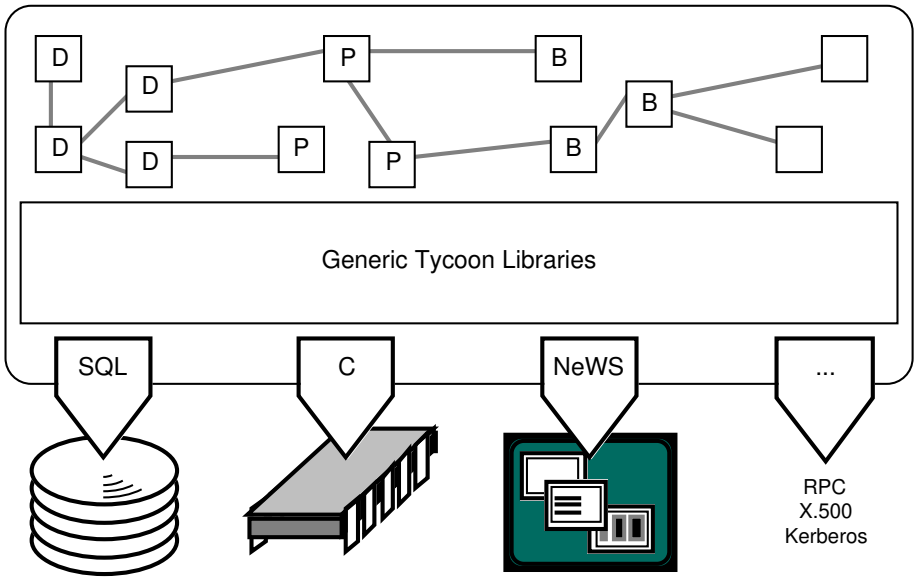


Abbildung 3.3: Integration, Erweiterung und Nutzung generischer Dienste in Tycoon [Matthes 93]

Einflüsse anderer Programmiersprachen

Die im Tycoon-System verwendete Programmiersprache TL ist eine Weiterentwicklung der experimentellen Sprachen Quest (*Quantifiers and Subtypes*) [Cardelli 89; Cardelli 90] und P-Quest [Matthes 91; Müller 91; Niederée et al. 92]. P-Quest ist ein um orthogonale Persistenz erweitertes Quest-System. Abgesehen von geringen syntaktischen Unterschieden werden alle sprachlichen Konstrukte dieser Sprachen in TL voll unterstützt. TL beseitigt einige Orthogonalitätsrestriktionen von Quest und führt wenige neue Sprachkonzepte ein, wie Subtypisierung zwischen Typoperatoren beliebiger Ordnung, rekursive Typoperatoren, erweiterbare Verbundwerte sowie Bibliotheken.

Die syntaktische Struktur und das Modulkonzept von TL lehnen sich an die Sprachen der Modula-Familie an (Modula-2, Oberon [Wirth 87] Modula-2+ [Rovner et al. 85], Modula-3 [Nelson 91] und Ada [Ichbiah 83]). Semantisch ist TL eng mit polymorphen funktionalen Sprachen der ML-Sprachfamilie verwandt [Cardelli 89; Cardelli 90; Mauny 91; Field, Harrison 88; Hudak 89]. Der Kern der semantischen Konzepte von TL entspricht der Sprache $F_{\leq}$ [Cardelli et al. 91], einer formalen Basis für die Studie neuartiger Typsysteme.

TL eignet sich wie C sowohl zur Anwendungs- als auch zur Systemprogrammierung. Aufgrund seines polymorphen Typsystems läßt sich TL auch als Datenmodellierungsprache einsetzen. Dies erinnert an Lisp-Entwicklungssysteme [Bobrow et al. 88] und an kommerzielle objektorientierte Programmiersprachen, wie Smalltalk [Goldberg, Robson 83] und C++ [Stroustrup 92]. Von den integrierten Datenbankprogrammiersprachen, wie PS-Algol [Atkinson et al. 82], Napier88 [Dearle et al. 89], Amber [Cardelli 86] und P-Quest, erbt Tycoon die orthogonale Kombinierbarkeit elementarer Basiskonzepte für Persistenzabstraktion, typvollständige Datenstrukturierung und Iterationsabstraktion [Atkinson, Bunemann 87; Schmidt, Matthes 90]. Darüber hinaus lassen sich die multiplen Programmrepräsentationen, wie TL, TML (*Tycoon Machine Language*, vgl. Abbildung 3.4) und PTML (*Portable TML*), zur dynamischen Optimierung nutzen [Gawecki, Matthes 94; Kiradjiev 94].

Ausgehend von einer Analyse der konzeptionellen und technologischen Grundlagen existierender Datenbanksprachen [Matthes, Schmidt 91a; Matthes, Schmidt 91b], verfolgt das Tycoon-System die Idee einer stark reduzierten Kernsprache zur Benennung, Bindung und Typisierung vordefinierter semantischer Objekte, wie Variablen, Funktionen, Typvariablen und Typoperatoren. Gleichzeitig ist es möglich, den Sprachkern vollständig typsicher um externe semantische Objekte (ganze Zahlen, Fließkommazahlen, Zeichenketten, Felder, Relationen, Sichten, Dateien, Fenster usw.) und die mit ihnen assoziierten generischen Funktionen zu erweitern (*add-on* vs. *built-in*) [Matthes, Schmidt 91b].

Von persistenten Systemen stammt das Konzept der Abwicklung aller Speicherzugriffe auf Daten und Programmrepräsentationen über eine Softwareschnittstelle auf sehr niedrigem Abstraktionsniveau (vgl. Abbildung 3.4). Dies führt zu einer Vereinfachung der Gesamtkomplexität persistenter Systeme, die durch einen zu vernachlässigenden Effizienzverlust erkauft wird [Matthes 93; Müller 91].

Das Tycoon-System besitzt eine interaktive Programmierumgebung, wie sie auch in funktionalen Systemen (ML, Lisp) zu finden ist. Dadurch unterscheidet es sich von konventionellen Übersetzersystemen, wie z.B. C-, Modula-2- oder Ada-Übersetzern. Neben dem Einsatz als Datenbankprogrammiersprache können daher auch ad hoc-Datenbankanfragen gestellt werden. Zusammen mit dem Konzept der Persistenz ist ein inkrementelles Arbeiten über mehrere Benutzersitzungen hinweg möglich. Gleichzeitig unterstützt das Bibliothekskonzept von TL die kontrollierte, gemeinsame Nutzung von Daten und Programmen durch mehrere Benutzer.

TL ermöglicht die Programmierung in verschiedenen Modellierungsstilen. *Funktionale* und *imperative* Programmierung werden direkt unterstützt. Aufgrund weitgehender sprachlicher Neutralität wird auch eine Variante des *objektorientierten* Programmierens ermöglicht. *Relationale* und *logikbasierte* Programmierung [Minker 88] werden nicht unmittelbar gefördert, weil sich unifikationsbasierte Evaluationsmodelle und deklarative Ansätze zu stark von funktionalen und imperativen Strukturen unterscheiden.

Architektur des Tycoon-Systems

Damit die Sprache TL im Tycoon-System eingeordnet werden kann, wird nachfolgend ein kurzer Überblick über die Tycoon-Systemarchitektur gegeben. Die Komponenten werden

z.B. in [Matthes 93] ausführlich beschrieben. Abbildung 3.4 vermittelt einen Überblick über die Komponenten und Schnittstellen der Tycoon-Systemarchitektur.

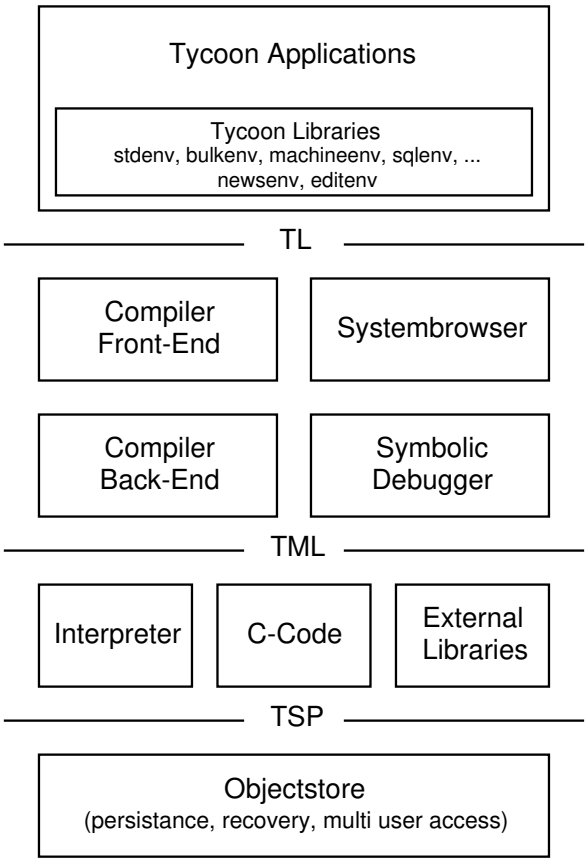


Abbildung 3.4: Schichten und Schnittstellen der Tycoon-Systemarchitektur [Matthes 93]

Die *Tycoon Language* (TL) bildet die Schnittstelle, über die Anwendungsprogrammierer auf das Tycoon-System zugreifen. Als weitere Schnittstellen existieren die *Tycoon Machine Language* (TML) und das *Tycoon Store Protocol* (TSP). TML ist ein portables, untypisiertes Programmformat, und TSP stellt ein abstraktes Speicherprotokoll dar.

Tycoon Literatur

Da bei der Erläuterung der Implementationskonzepte und -ideen für die beiden, im Rahmen dieser Arbeit erstellten Bibliotheken (vgl. Kapitel 6 und 7), nicht auf die Syntax der Sprache eingegangen werden soll, sind grundlegende Kenntnisse in der Tycoon-Programmierung notwendig. Eine schrittweise Einführung in die Tycoon-Programmierung ist jedoch im Rahmen dieser Arbeit nicht möglich. Daher sei hier auf weiterführende Literatur verwiesen. Eine

sehr umfangreiche und detaillierte Einführung in die Sprache TL ist in [Matthes, Müßig 93] zu finden. Im Anhang von [Matthes, Müßig 93] sind auch ausführliche Übersichten über die Grammatik und die vordefinierten Bezeichner der Sprache TL zu finden. Die praktische Benutzung der interaktiven Tycoon-Systemumgebung, die implementierten Bibliotheken, die Anbindung externer Bibliotheken sowie Formatierungs- und Namensrichtlinien für TL Programme werden in [Mathiske et al. 93] vorgestellt.

4. Datenvisualisierung in bestehenden Systemen

Ziel neuerer Datenbanksysteme (*next-generation database systems*) ist die Erschließung neuer Anwendungsgebiete für Datenbanken, z.B. im CAD- oder Multimedia-Bereich. Um die in diesen Anwendungsgebieten bestehenden Anforderungen zu erfüllen, sind in den dazugehörigen Datenmodellen neuartige Konzepte zu finden.

Zum Beispiel können Daten, die in relationalen Systemen über mehrere Tabellen verteilt werden müssen, in objektorientierten Systemen als zusammenhängende Objekte gespeichert werden. In objektorientierten Systemen werden Objekte auch nicht über Schlüssel sondern über eine eindeutige Identität identifiziert. Dadurch werden Redundanzprobleme auf elegante Weise vermieden. Neue Datentypen erlauben die Ablage von Multimediadaten, wie Audio- und Video-Sequenzen. Neben den eigentlichen Daten kann meist auch das prozedurale Verhalten der Objekte mit abgespeichert werden [Cattell 91].

Oft sind in solchen Systemen neben der eigentlichen Datenbankfunktionalität eine Programmiersprache und eine *Visualisierungskomponente* zu finden. In diesem Kapitel werden verschiedene neuere kommerzielle und Forschungssysteme hinsichtlich ihrer Möglichkeiten zur Datenvisualisierung analysiert. Im einzelnen sind das die kommerziellen Datenbanksysteme DBPL, INGRES, O_2 und GemStone. Ein weiterer Abschnitt ist der Programmiersprache VisualBASIC gewidmet (vgl. Tabelle 4.1 auf Seite 24).

Ziel dieser Untersuchungen ist die Auslotung der Möglichkeiten dieser Systeme hinsichtlich ihrer Visualisierungskomponente. Eine Zusammenfassung der Untersuchungsergebnisse ist am Ende dieses Kapitels zu finden. Eine tabellarische Übersicht der Ergebnisse befindet sich in Anhang B.

In einem ersten Abschnitt werden die Vorgehensweise bei der Analyse der Systeme und die dabei herangezogen Kriterien vorgestellt. Die Vorstellung der einzelnen Systeme erfolgt in einem weiteren Abschnitt. Dazu werden von jedem System zunächst überblicksweise die Architektur und die vorrangigen Entwicklungsziele vorgestellt. Anschließend wird jeweils die Visualisierungskomponente genauer beschrieben. Zuletzt wird die Implementation des Programmbeispiels aus Anhang A vorgestellt und die sich dabei abzeichnenden Möglichkeiten und Grenzen der Systeme hinsichtlich ihrer Visualisierungskomponente diskutiert.

System	Version	Literatur	Hersteller
DBPL	2.0	[Schmidt, Matthes 92] [Matthes et al. 92a]	Universität Hamburg
INGRES	6.3	[Ingres 89] [Ingres 90f]	INGRES Corp.
O_2	4.3.1	[Deux 91] [Bancilhon et al. 92]	O_2 Technology
GemStone	3.2	[Bretl et al. 89] [Butterworth et al. 91]	Servio Corp.
VisualBASIC	3.0	[Ehrmann 93b] [Maslo, Dittrich 93]	Microsoft Corp.

Tabelle 4.1: Übersicht über die untersuchten Systeme

4.1 Untersuchungsmethodik und Kriterien

Dieser Abschnitt wird in zwei Teile untergliedert. Im ersten Teil wird die Untersuchungsmethodik vorgestellt, die angewendet wird, um die in Tabelle 4.1 genannten Systeme zu analysieren. Anschließend werden im zweiten Teil die zur Analyse verwendeten Kriterien vorgestellt.

4.1.1 Untersuchungsmethodik

Um die Analyse durchführen zu können, ist es einerseits notwendig, unterschiedliche Systeme zu betrachten, denn nur so können die verschiedenen Möglichkeiten der Datenvisualisierung erkannt und analysiert werden. Andererseits ist es auch notwendig, detaillierte Kriterien festzulegen, um die Systeme daran zu bewerten.

Die Auswahl der Systeme erfolgt nach unterschiedlichen Gesichtspunkten. Mit INGRES, O_2 und VisualBASIC sind drei gegenwärtig sehr verbreitete Systeme vertreten. DBPL ist ein universitäres Forschungssystem, mit dem am Arbeitsbereich DBIS erste Erfahrungen in der generischen Datenvisualisierung gesammelt wurden. GemStone wird behandelt, weil das System bei der Benutzerschnittstelle einen von den anderen Systemen abweichenden Weg verfolgt.

Die Auswahl der Analysekriterien ist unter dem Gesichtspunkt zu betrachten, daß es *nicht* das Ziel der Untersuchung ist, die Systeme hinsichtlich bestehender Prüfkriterien softwareergonomisch zu evaluieren (vgl. z.B. [Oppermann et al. 92]), sondern die technischen Möglichkeiten der Visualisierungskomponenten zu betrachten und daraus detaillierte Anforderungen an die typsichere und generische Datenvisualisierung in Tycoon zu erhalten.

Um zunächst einen groben Überblick über die Systeme zu bekommen, ist als erstes zu untersuchen, auf welchen Soft- und Hardwareplattformen die Systeme jeweils verfügbar sind.

Anschließend werden alle mit dem System ausgelieferten Komponenten nach Programmier-, Datenbank- und Benutzerschnittstellenwerkzeugen getrennt. Dadurch werden für die weitere Untersuchung die im Zusammenhang mit der Benutzerschnittstelle verwendeten Werkzeuge isoliert. Schwerpunkte für die weitere Untersuchung bilden die Visualisierungs- und Interoperabilitätsmöglichkeiten der Werkzeuge. Nicht Bestandteil dieser Untersuchungen sind die zu den Systemen zusätzlich verfügbaren Komponenten. An einigen Stellen wird jedoch auf solche Komponenten hingewiesen.

Bei der Analyse wird für alle Systeme zunächst auf beschreibende Literatur zurückgegriffen. Aus der verwendeten Literatur ergeben sich u.a. die vorrangigen Entwicklungsziele und die funktionale Architektur. Vier der fünf Systeme (DBPL, INGRES, O_2 und VisualBASIC) stehen darüber hinaus auch für praktische Untersuchungen zur Verfügung. In diesen Systemen wird das Programmbeispiel aus Anhang A implementiert. Einige Systeme verfolgen besonders erwähnenswerte Ansätze und werden daher ausführlicher beschrieben.

Da die Daten objektiv ermittelt werden können und nicht von subjektiven Einstellungen abhängen, ist der Einsatz von mehreren Testpersonen zum Erlangen der Ergebnisse nicht notwendig. Die Ergebnisse der Analyse sind in Anhang B tabellarisch zusammengefaßt, denn in dieser Form lassen sich die Ergebnisse gut überblicken und miteinander vergleichen.

4.1.2 Untersuchte Kriterien

In diesem Abschnitt werden die Kriterien vorgestellt, die zur Analyse der einzelnen Systeme herangezogen werden. Die Kriterien lassen sich grob nach den folgenden vier Schwerpunkten aufteilen:

- Hard- und Softwareplattformen
- Systemkomponenten
- Visualisierungsmöglichkeiten
- Interoperabilität

Die vier Schwerpunkte werden nachfolgend ausführlich beschrieben. Die Ergebnisse dieser Analyse sind sowohl in Abschnitt 4.3 als auch in tabellarischer Form in Anhang B zu finden. Bei der Vorstellung der einzelnen Kriterien ist zusätzlich jeweils die Tabelle angegeben, in der die zum Kriterium gehörenden Untersuchungsergebnisse zu finden sind.

Da die Systeme unterschiedliche Ziele verfolgen, können nicht immer alle Kriterien untersucht werden. Es kann z.B. vorkommen, daß ein System eine Teilkomponente nicht beinhaltet. In solchen Fällen ist dies in der entsprechenden Tabelle vermerkt.

Hard- und Softwareplattformen

Als erstes wird festgestellt, auf welchen Hard- und Softwareplattformen die einzelnen Systeme verfügbar sind. Dies dient der groben Einordnung in graphische und textuelle Systeme.

Textuelle Systeme können i.d.R. nur mit der Tastatur bedient werden. Die Untersuchung der Interoperabilitätsmöglichkeiten solcher Systeme erübrigt sich weitgehend.

Untersucht werden die vier Hardwareplattformen Sparc (Sun Microsystems, Inc.), Vax (Digital Equipment Corp.), PC und Macintosh (Apple Computer, Inc.). Als Benutzerschnittstellen werden OSF/Motif und OPEN LOOK (Sparc), VMS (Vax), Microsoft Windows und DOS (PC) sowie der Finder (Macintosh) untersucht (vgl. Tabelle B.3).

Systemkomponenten

Als nächstes wird festgestellt, welche Komponenten in den Systemen für einen Anwendungsentwickler zur Verfügung stehen. Dabei werden nur die direkt mit dem System gelieferten Komponenten berücksichtigt. Unberücksichtigt bleiben hier allgemein verfügbare Werkzeuge, wie z.B. ein Texteditor eines Unix-Systems, der zur Editierung eines Quelltextes verwendet werden kann, aber nicht zur Entwicklungsumgebung gehört. Des weiteren wird untersucht, ob sich in das System ohne großen Aufwand weitere Dienste oder Komponenten integrieren lassen. In diesem Zusammenhang ist auch wichtig, über welche Programmiersprachen ein System verfügt, angesprochen oder erweitert werden kann.

Die Komponenten können ihrerseits wieder unterteilt werden. Es wird zwischen Komponenten zur Programmerstellung, zur Erstellung der Benutzerschnittstelle sowie zur Administration, Pflege und Abfrage der Datenbank unterschieden.

Als Komponenten zur Programmerstellung kommen eine interaktive Entwicklungsumgebung, ein Quelltext-Editor, ein Interpreter und/oder Übersetzer, ein Debugger, eine Projektverwaltung sowie eine interaktive und kontextsensitive Hilfe (*online help*) in Betracht (vgl. Tabelle B.4).

Die graphische Benutzerschnittstelle kann entweder interaktiv über einen Ressource-Editor, automatisch mit einem Maskengenerator oder durch den Aufruf von Bibliotheksfunktionen erstellt werden. Ein Maskengenerator vereinfacht die Erstellung von Benutzerschnittstellen erheblich, da i.d.R. nur noch geringe Änderungen an den automatisch generierten Masken durchgeführt werden müssen. Weiterhin wird festgestellt, ob das System über ein generisches Werkzeug zum Anzeigen von Werten und Objekten unabhängig vom ihrem Typ (*browser*) verfügt (vgl. Tabelle B.5).

Für eine Datenbankkomponente sind Werkzeuge zur Administration, zum interaktiven Design des Datenbankschemas sowie eine Abfragesprache von Bedeutung. Daneben wird das Vorhandensein von vier der grundlegenden Eigenschaften von Datenbanken untersucht. Diese vier Eigenschaften sind Transaktionsunterstützung, Fehlererholung (*recovery*), Freispeicherverwaltung (*garbage collection*) und orthogonale Persistenz (vgl. Tabelle B.6).

Bei der Untersuchung der Sprachschnittstellen wird zunächst zwischen in das System integrierten und externen Programmiersprachen unterschieden. Als externe Programmiersprachen kommen C, C++, Modula-2 und Smalltalk in Betracht (vgl. Tabelle B.7).

Visualisierungsmöglichkeiten

Für diese Arbeit am interessantesten ist die Analyse der Visualisierungsmöglichkeiten. Untersucht wird dabei, welche Datentypen bzw. welche Werte visualisierbar sind. Angefangen

von einfachen Basistypen, wie ganzen Zahlen und Zeichenketten, über strukturierte Typen, wie Verbunde und Varianten, bis hin zu Massendatentypen, wie Listen und Mengen, werden die Systeme auf ihre Visualisierungsmöglichkeiten hin untersucht. Einige Systeme bieten auch im Multimediabereich Typ- und Visualisierungsunterstützung. Hier finden sich Text-, Audio- und Bildtypen. Darüber hinaus sind teilweise noch Visualisierungsmöglichkeiten in Spezialbereichen zu finden, wie z.B. von Währungs- und Datumstypen oder von Verzeichnissen und Dateien (vgl. Tabellen B.8 bis B.12).

Werden die Benutzerschnittstellen als Ganzes betrachtet, so kann zwischen automatisch zur Laufzeit generierten und statisch vom Anwender durch Ressource-Editoren oder Bibliotheksaufrufe erstellten Benutzerschnittstellen unterschieden werden. Anschließend wird untersucht, ob die einzelnen Interaktionskomponenten beliebig miteinander kombiniert werden können und die vorgegebenen Visualisierungskomponenten durch benutzerdefinierte erweitert werden können.

Ein Zustand auf den durch die Visualisierungskomponente hingewiesen werden sollte, ist das mehrfache Anzeigen eines Wertes oder Objekts und die daraus resultierenden Probleme beim Ändern. Auch kann eine visuelle Rückmeldung über die Hierarchie der Objekte für den Anwender nützlich sein. Wird eine Kollektion angezeigt, so ist z.B. die Anzeige der Größe der Kollektion und der Position des aktuellen Elements wünschenswert (vgl. Tabelle B.13).

Interoperabilität

In fensterorientierten Umgebungen, in denen mehrere Anwendungen gleichzeitig aktiv sein können, treten Situationen auf, die den Datenaustausch zwischen den Fenstern einer Anwendung oder zwischen verschiedenen Anwendungen erfordern. Aus diesem Grund werden die Interoperabilitätsmöglichkeiten der einzelnen Systeme untersucht. Der Datenaustausch wird dabei nach zwei Kriterien differenziert. Das erste Kriterium ist die Vorgehensweise beim Austausch der Daten und das zweite ist die Komplexität der Daten, die ausgetauscht werden können.

Die einfachste Art des Datenaustauschs erfolgt über das Dateisystem. Die von einem Anwendungsprogramm in einer Datei gespeicherten Daten können von anderen Anwendungsprogrammen gelesen werden. Dazu ist es notwendig, daß das einlesende Anwendungsprogramm die Daten in der Datei richtig interpretieren kann, also z.B. ein bestimmtes Bild- oder Textformat erkennt. Bei dieser Art des Datenaustauschs können i.d.R. beliebig komplexe Daten ausgetauscht werden. Diese Art des Austauschs von Daten wird in diesem Zusammenhang nicht weiter betrachtet.

Ein wesentlich eleganterer Weg ist der Austausch von Daten über die sogenannte Zwischenablage (*clipboard*). Aus der Anwendung können Daten mit den Operationen Ausschneiden (*cut*) oder Kopieren (*copy*) in die Zwischenablage übernommen werden. Anschließend können sie mit der Einfüge-Operation (*paste*) an beliebiger Stelle eingefügt werden. Dabei ist wichtig, wie komplex die Daten sein dürfen, die auf diese Weise übertragen werden. Darüber hinaus ist es wichtig, ob die Daten nur innerhalb einer Anwendung oder auch in andere Anwendungen übernommen werden können. Untersucht wird auch, ob sich die momentan

in der Zwischenablage befindlichen Daten anzeigen lassen. Für einige Anwendungsfälle ist darüber hinaus eine stapelbare Zwischenablage unabdingbar. Eine solche Zwischenablage ist mit einem Kellerspeicher vergleichbar (vgl. Tabelle B.14).

Ein anderer Weg des Datenaustauschs wird mit *drag & drop* bezeichnet. Hierbei werden in einer Anwendung selektierte oder durch eine Ikone symbolisierte Daten mit der Maus aufgenommen (*dragged*) und an anderer Stelle über einem Fenster oder einer Ikone fallengelassen (*dropped*). Auch hier ist wieder wichtig, wie komplex die zu übertragenden Daten sein dürfen und ob Anwendungsgrenzen überschritten werden können. Als letztes wird untersucht, ob zwischen Kopier- und Verschiebeoperationen gewählt werden kann (vgl. Tabelle B.15).

Die Möglichkeiten der Benutzerschnittstelle zum Rückgängig machen von Operationen (*undo*) werden als nächstes betrachtet. Auch in diesem Zusammenhang wird untersucht, wie komplex die Daten sein dürfen, die durch eine Rücknahmeoperation wiederhergestellt werden können. (vgl. Tabelle B.16)

Als letzter Punkt der Interoperabilitätsmöglichkeiten wird die Bedienung des Systems klassifiziert. Generell kann zwischen mausgesteuerter Bedienung und Bedienung über die Tastatur unterschieden werden (vgl. Tabelle B.17).

4.2 Vorstellung der untersuchten Systeme

In diesem Abschnitt werden die untersuchten Systeme ausführlich beschrieben. Dazu werden von jedem System zunächst überblicksweise die Architektur und die vorrangigen Entwicklungsziele vorgestellt. Anschließend wird jeweils die Visualisierungskomponente genauer analysiert. Zuletzt wird die Implementation des Programmbeispiels aus Anhang A vorgestellt. Ein Überblick über einige der vorgestellten Systeme ist auch in [Cattell 91; Heuer 92] zu finden.

4.2.1 DBPL

Die Datenbankprogrammiersprache DBPL [Schmidt, Matthes 92; Matthes et al. 92a] (*DataBase Programming Language*) ist das Ergebnis eines seit 1985 an der Universität Hamburg im Rahmen von FIDE [FIDE 90] (*Formally Integrated Data Environment*) laufenden Forschungsprojekts. Ziel des Projekts war u.a. die Beseitigung des konzeptuellen Bruchs (*impedance mismatch*) [Manthey 91] zwischen mengenorientierter, deklarativer Datenverarbeitung in Datenbanksystemen und dem elementorientierten, prozeduralen Paradigma konventioneller Programmiersprachen durch die Erweiterung der statisch typisierten Programmiersprache Modula-2 um relationale Konzepte, Persistenz und Transaktionen.

4.2.1.1 Systemüberblick

Bei DBPL handelt es sich um eine Erweiterung der Programmiersprache Modula-2. In DBPL gibt es jedoch keine Trennung zwischen Wirtsprogrammiersprache und eingebetteter

Datenbanksprache, wie es bei C-Programmen und Embedded SQL üblich ist. Aus diesem Grund mußten alle relationalen Anforderungen in das Syntaxkonzept von Modula-2 integriert werden. Dies führt zu einer orthogonalen Programmiersprache, und es kann auf einen Vorübersetzer (*pre-compiler*) verzichtet werden.

Die relationalen Konzepte von DBPL sind im Laufzeitsystem implementiert. Das Laufzeitsystem selbst ist in Modula-2 geschrieben. Es wird mit dem übersetzten Anwendungsprogramm des Benutzers zusammengebunden. Wie in Abbildung 4.1 dargestellt, ist das DBPL-Laufzeitsystem gemäß einer Schichtenarchitektur aufgebaut. Jede Schicht erfüllt ihre genau spezifizierten Aufgaben ausschließlich unter Benutzung wohldefinierter Schnittstellen zu den angrenzenden Schichten. Die Schichten erfüllen im einzelnen die folgenden Aufgaben [Rudloff 90]:

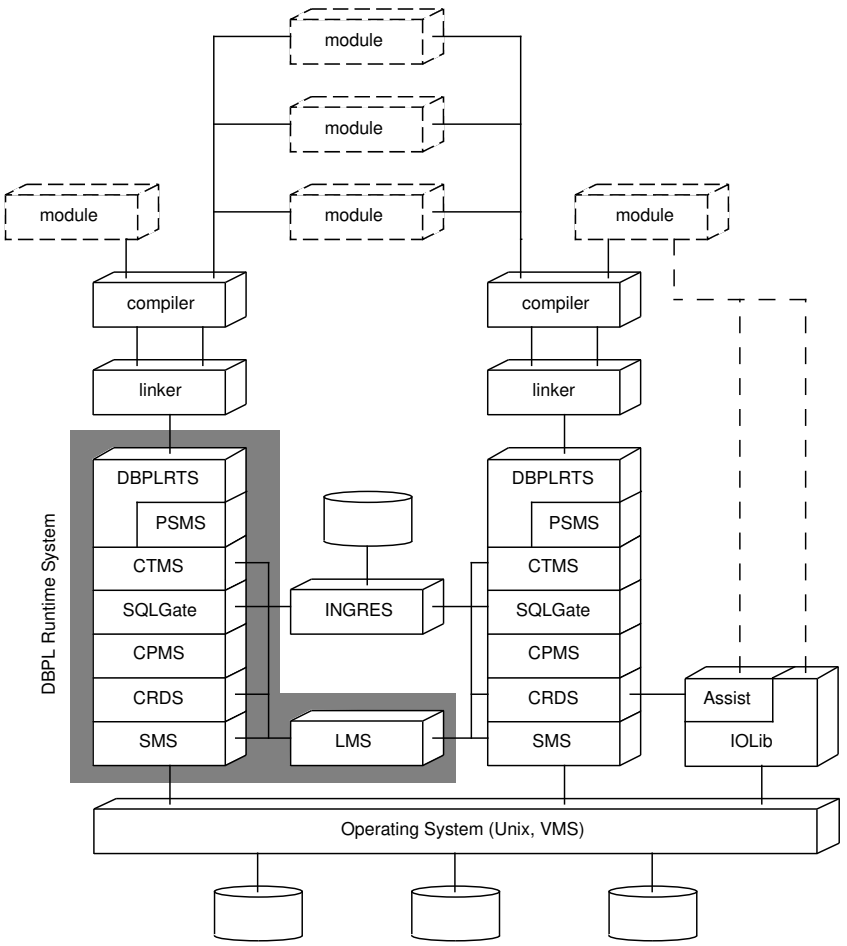


Abbildung 4.1: Architektur von DBPL [Matthes et al. 92a]

DBPLRTS: Die Hauptaufgabe des DBPL *Run Time Systems* ist es, eine saubere Schnittstelle für die Benutzer des Laufzeitsystems anzubieten, über die alle notwendigen Operationen durchgeführt werden können. Die geforderte Funktionalität selbst wird zum größten Teil durch tieferliegende Schichten realisiert, wobei dem Laufzeitsystem eher eine koordinierende Rolle zukommt.

PSMS: Das *Predicate Set Management System* realisiert den Aufbau einer systeminternen Repräsentation von Prädikaten sowie der notwendigen Operationen, die von höheren Schichten zur Bearbeitung dieser Prädikate benötigt werden.

CTMS: Das *Complex Transaction Management System* verwirklicht den anomalienfreien Mehrbenutzerbetrieb auf einer gemeinsamen Datenbasis. Dazu erfolgt jeder Zugriff auf diese unter Beachtung eines hierarchischen Sperrprotokolls, dessen Einhaltung durch einen zentralen Überwachungsprozeß erzwungen wird.

SQLGate: Das *SQL Gate* [Matthes et al. 92b] ermöglicht den transparenten Zugriff auf INGRES-Datenbanken.

CPMS: Zur Auswertung von Prädikaten dient das *Complex Predicate Management System*. Es zerfällt in zwei Teile. Der erste Teil ist ein Umformungssystem, das die Prädikate in eine standardisierte Struktur umwandelt, die es dem zweiten Teil, dem eigentlichen Auswertungssystem ermöglicht, Prädikate möglichst effizient auszuwerten.

CRDS: Das *Complex Relation Data System* realisiert den Aufbau der Relationen. Dazu gehören neben dem Aufbau von Typstrukturen auch der Aufbau von Relationenwerten sowie die eventuell dauerhafte (*persistente*) Speicherung dieser Werte. In dieser Schicht ist deswegen auch ein Datenwörterbuch enthalten.

SMS: Das *Storage Management System* sorgt für die Verwaltung des notwendigen Haupt- und Externspeicherplatzes unter Verwendung von Pufferungsstrategien zur Beschleunigung des Zugriffs.

LMS: Das *Lock Management System* realisiert einen generischen Sperrmechanismus.

DBPL verfügt im Gegensatz zu den anderen untersuchten Systemen nicht über eine integrierte Entwicklungsumgebung. Das DBPL-System besteht lediglich aus dem Übersetzer und dem Laufzeitsystem. Aus diesem Grund werden zur Entwicklung von Programmen die Werkzeuge der jeweiligen Systemumgebung verwendet. Als Werkzeuge kommen ein Quelltext-Editor, eine Projektverwaltung und ein Debugger in Betracht.

Benutzerschnittstellen können über die neben dem Laufzeitsystem in Abbildung 4.1 auf Seite 29 dargestellte Bibliothek (*IOLib*) aufgebaut werden. Bestandteil dieser Bibliothek ist auch ein universeller Datenbankbrowser (*Assist*). Beide werden im nächsten Abschnitt genauer beschrieben.

4.2.1.2 Benutzerschnittstelle

DBPL-Programme können mit einer textuellen, d.h. aus ASCII-Zeichen aufgebauten Benutzerschnittstelle versehen werden. Dazu existieren in der Bibliothek *IOLib* verschiedene Module zum Aufbau von anwendungsspezifischen Benutzerschnittstellen. Im einzelnen wird der Aufbau von einfachen Menüs (*Selector*) und Fenstern (*Window*) sowie von satz- (*Form*) und relationenorientierten (*SetForm*) Masken unterstützt. Wie in Abbildung 4.2 auf Seite 32 zu sehen ist, steht an der Benutzerschnittstelle eine Statuszeile zur Ausgabe von Meldungen und Fehlern zur Verfügung. Diese wird immer dann angezeigt, wenn eines dieser Module importiert wird.

Satzorientierte Masken dienen dem Anzeigen und Manipulieren von einzelnen Datensätzen. Relationenorientierte Masken ermöglichen die tabellarische Darstellung von Relationen. In beiden Maskenarten können alle in DBPL durch die vorhandenen Basistypen beschriebenen Werte dargestellt werden. Der Aufbau dieser Masken erfolgt durch Prozeduraufrufe mit entsprechenden Parametern und Angaben über die Position und Größe der Felder. Die Navigation in den Masken erfolgt über Cursor- und spezielle Funktionstasten.

Der Vorteil der freien Gestaltbarkeit solcher Masken muß durch die relativ aufwendige und fehleranfällige Kodierung durch Bibliotheksaufrufe erkauft werden. Im Gegensatz dazu ist es mit einer einzigen Funktion aus dem Modul *Assist* möglich, eine Tabelle beliebigen Relationentyps anzuzeigen. Dabei werden immer alle Attribute der Reihe nach angezeigt. Eine weitere Funktion aus diesem Modul ermöglicht das menügesteuerte Anzeigen von ganzen Datenbanken. Die Navigation erfolgt wieder über Cursor- und spezielle Funktionstasten.

4.2.1.3 Programmbeispiel

In diesem Abschnitt wird die Implementation des Programmbeispiels aus Anhang A in DBPL vorgestellt. Dabei werden drei unterschiedliche Ansätze aufgezeigt. Das erste Beispiel zeigt den Aufbau einer satzorientierten Benutzerschnittstelle durch Funktionsaufrufe des Moduls *Form*. Das nächste Beispiel illustriert die Verwendung des Moduls *SetForm* zum Aufbau einer tabellenorientierten Benutzerschnittstelle. Das dritte Beispiel zeigt die Benutzung des generischen Relationenbrowsers *Assist*.

Satzorientierte Visualisierung

Die Funktionen des Moduls *Form* ermöglichen einen flexiblen Aufbau der textuellen Benutzerschnittstelle. Dafür ist es als erstes notwendig, mit der Funktion *Create* ein Formular zu erzeugen. Alle nachfolgend aufgerufenen Funktionen zum Erzeugen von Feldern beziehen sich dann automatisch solange auf dieses Formular, bis durch einen erneuten Aufruf der *Create*-Funktion ein neues Formular erzeugt wird.

Für die Felder sind dann im weiteren jeweils zwei Funktionen aufzurufen. Mit dem ersten Funktionsaufruf (*ConstField*) wird der Name des Feldes plaziert.

PROCEDURE *ConstField(line : Line; column : Column;*
constval : ***ARRAY OF CHAR****);*

Mit der zweiten Funktion wird das Feld plaziert, welches den Wert der Komponente visualisieren soll. Da diese Funktion typisiert ist, kann mit ihr nur für jeweils einen Datentyp ein Feld erzeugt werden. Daher gibt es für jeden Basistyp von DBPL eine eigene Funktion. Beispielhaft ist hier die Signatur für die Funktion zum Erzeugen eines Feldes für positive ganze Zahlen angegeben:

PROCEDURE *CardField (line : Line; column : Column; min, max :* ***CARDINAL****;*
VAR *cardvar :* ***CARDINAL****);*

Um ein Feld mit dem Namen *Age* in der vierten Zeile und dritten Spalte und ein Feld zur Aus- und Eingabe positiver ganzer Zahlen in der gleichen Zeile, aber der zehnten Spalte zu erhalten, müssen die folgenden beiden Funktionen ausgeführt werden:

Form.ConstField(4, 3, "Age");
Form.CardField(4, 10, 0, 100, person.age);

Über die Positionsangaben ist es möglich, solche Masken frei aufzubauen. Durch die beiden der Funktion *CardField* zusätzlich übergebenen Parameter, *min* und *max*, kann schon an der Benutzerschnittstelle geprüft werden, ob der eingegebene Wert in einem gültigen Bereich liegt. Wie an diesem Beispiel weiter zu sehen ist, wird durch die Angabe von *person.age* die Verbindung zwischen Variable und Benutzerschnittstelle hergestellt. Dies muß für jedes Feld separat durchgeführt werden.

Für das Programmbeispiel kann mit den Funktionen des Moduls *Form* die im Anhang A.2 aufgeführte Ausgabefunktion *DisplayForm* implementiert werden. Das Ergebnis eines Aufrufs dieser Funktion ist in Abbildung 4.2 zu finden.

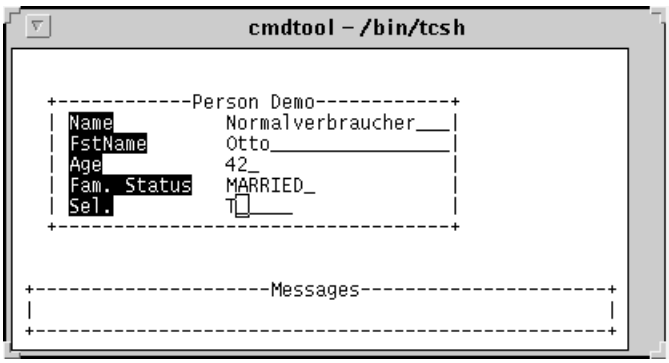


Abbildung 4.2: Benutzerschnittstelle von DBPL (*Form*)

Nachteilig an dem mit dem Modul *Form* verfolgten Ansatz ist die Tatsache, daß es von Anwendern nicht um eigene, anwendungsspezifische Funktionen erweitert werden kann. Für eine Erweiterung ist es notwendig, den Implementationsteil dieses Moduls zu ändern. Dadurch würde jedoch ein Grundprinzip (Kapselung) von Modula-2 bzw. DBPL verletzt werden. Zudem sind die zur Ausgabe verwendeten Funktionen stark systemspezifisch und daher wenig portabel.

Tabellenorientierte Visualisierung

Der Aufbau von tabellenorientierten Formularen ist in der gleichen Weise mit den Funktionen des Moduls *Form* durchführbar. Die Namen der Komponenten bilden hier jedoch die Spaltenüberschriften. Sie müssen in einem gesonderten Formular nebeneinander plaziert werden. Auch die Komponenten werden nebeneinander plaziert. Beide Formulare werden dann der Funktion *Create* des Moduls *SetForm* übergeben. Anschließend kann ein solches Formular angezeigt werden.

Die für das Programmbeispiel definierte Ausgabefunktion (*DisplaySetForm*) ist in Anhang A.2 zu finden. Das Ergebnis eines Aufrufs dieser Funktion ist in Abbildung 4.3 zu finden.

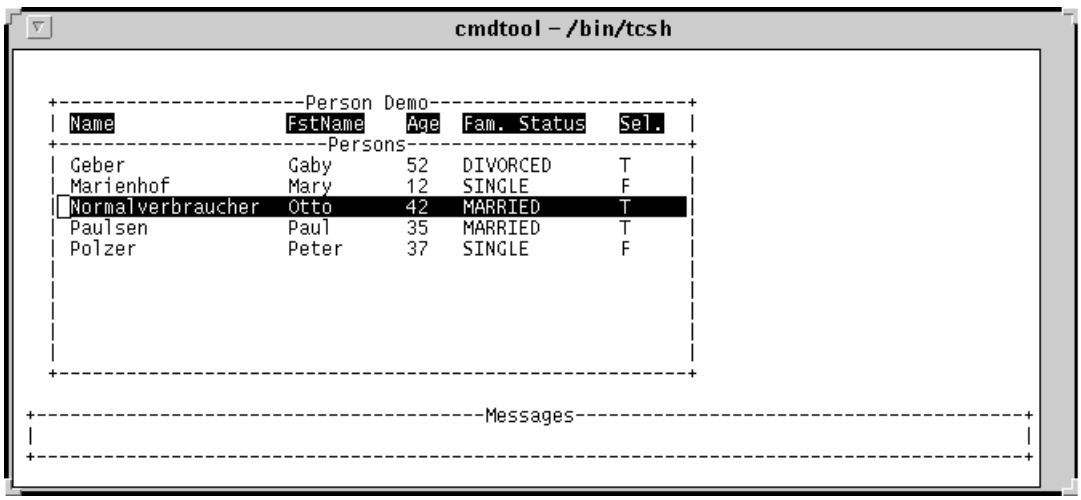


Abbildung 4.3: Benutzerschnittstelle von DBPL (*SetForm*)

In tabellenorientierten Formularen können mehrere Datensätze gleichzeitig angezeigt werden. Der Anwender bekommt so eine bessere Übersicht. Zum Ändern der Daten ist es jedoch notwendig, ein satzorientiertes Formular mit dem zu verändernden Datensatz aufzurufen.

Generische Visualisierung

Die dritte und einfachste Möglichkeit, in DBPL relationale Daten zu visualisieren, ist die Benutzung der Funktion *DisplayRelation* aus dem Modul *Assist*. Wie in Abbildung 4.1 auf

Seite 29 zu sehen ist, greift das Modul *Assist* für die Visualisierung der Relationenwerte auf die Funktionalität einer tieferen Schicht (CRDS) des Laufzeitsystems von DBPL zurück. In dieser Schicht erfolgt der Aufbau der Ausprägungen des Typs **RELATION**. Da das statische und monomorphe Typsystem von DBPL keine Typparameter unterstützt, kann eine generische Funktion in DBPL nicht typsicher implementiert werden. Wie an der Signatur dieser Funktion zu sehen ist, wird eine Relation als Zeiger des Typs **WORD** übergeben.

PROCEDURE *DisplayRelation(rel :* ***WORD****;*
line, column : ***CARDINAL****; maxWidth, maxHeight :* ***CARDINAL****;*
title : ***ARRAY OF CHAR****);*

Diese Funktion erlaubt das generische Anzeigen von beliebig verschachtelten Relationenwerten. Jedoch wird hier zur Anzeige eine Standardmaske verwendet, in der der Reihe nach alle Komponenten angezeigt werden. Der Anwender kann nur die Größe und die Position des Fensters bestimmen, auf die Plazierung der Felder im Fenster hat er keinen Einfluß. Die Informationen über Namen und Typen der Komponenten werden aus dem Datenwörterbuch von DBPL extrahiert. Wie in Abbildung 4.4 zu sehen, fehlt jedoch die Semantik der Werte. Zum Beispiel werden die Werte des Typs *familyStatus* durch ganze Zahlen repräsentiert und nicht, wie in Abbildung 4.3, durch Zeichenketten.

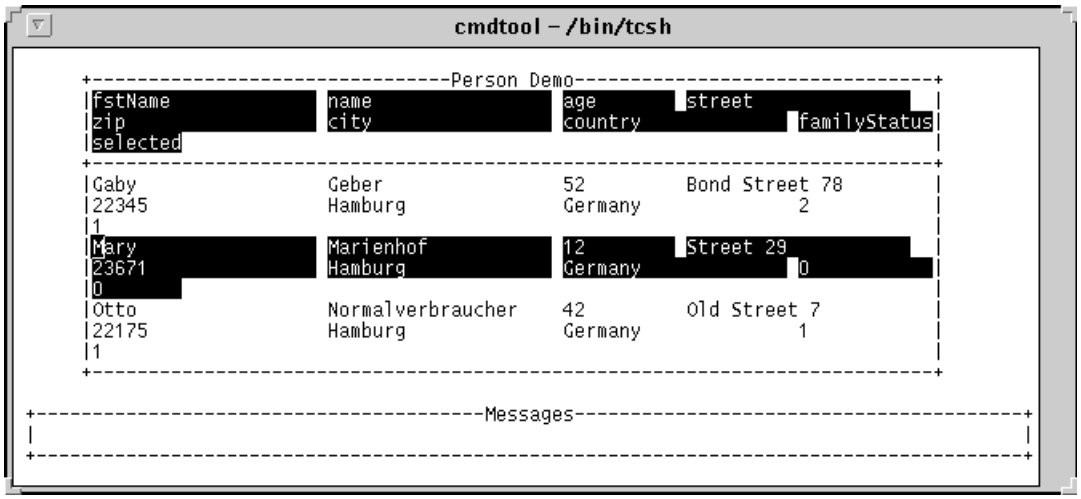


Abbildung 4.4: Benutzerschnittstelle von DBPL (*Assist*)

Im Unterschied zu den satz- und tabellenorientierten Formularen müssen hier keine Verbindungen zwischen den Komponenten der Verbundvariable und den Spalten der Tabelle hergestellt werden. Wie an dem folgenden Aufruf zu sehen ist, reicht es, der Funktion die Relationenvariable zu übergeben:

Assist.DisplayRelation(persons, 0, 4, 80, 15, "Person Demo");

4.2.2 INGRES

Der erste Prototyp des University INGRES (*INteractive Graphics and Retrieval System*) ist um 1974 an der University of California in Berkeley entwickelt worden. Das kommerzielle INGRES [Date 87; Ingres 89] ist ca. 1980 aus diesem Prototypen hervorgegangen. Es wird von der Firma INGRES Corp. vertrieben und weiterentwickelt.

4.2.2.1 Systemüberblick

INGRES ist ein relationales Datenbankmanagementsystem (*relational database management system*). In Abbildung 4.5 sind die beiden Entwicklungsumgebungen (*3GL-* und *4GL-Tools*) von INGRES graphisch dargestellt. Anwendungen können direkt in einer der beiden Entwicklungsumgebungen oder in einer Hochsprache entwickelt werden. Die Verbindung zwischen Entwicklungsumgebung und der eigentlichen Datenbank wird über eine Netzwerkkomponente realisiert [Ingres 90d].

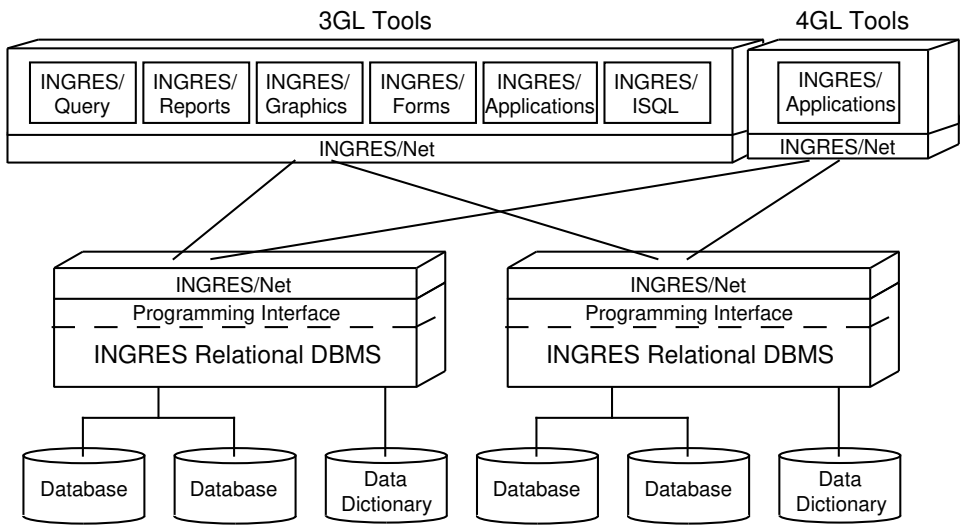


Abbildung 4.5: Architektur von INGRES

Die textuelle 3GL-Umgebung [Ingres 90b; Ingres 90c] ermöglicht neben der interaktiven Benutzung von INGRES über die Datenbanksprachen SQL (*Structured Query Language*), QUEL (*QUEry Language*) und QBE (*Query By Example*), die menügesteuerte Erstellung von Reports, Graphiken und Masken. Auch ganze Anwendungen können so in dieser Umgebung aufgebaut werden.

Für graphische Systeme ist die 4GL-Umgebung INGRES/Windows 4GL [Ingres 90a] eine komplette Entwicklungsumgebung. Vorhandene INGRES-Datenbanken können auf einfache

Weise in INGRES/Windows 4GL übernommen werden. Anwendungen bestehen hier aus einer Folge von Rahmen (*frames*). In den Rahmen befinden sich die Interaktionselemente, die vom Anwender manipuliert werden können. Zu den Rahmen gehören jeweils eine Reihe von ereignisabhängigen Prozeduren. Die Prozeduren enthalten die Anwendungsfunktionalität. Die gesamte Anwendung wird in der Datenbank gehalten.

4.2.2.2 Benutzerschnittstelle

INGRES/Windows 4GL erlaubt die interaktive Erstellung von graphischen Benutzerschnittstellen für Unix-Systeme. In der Umgebung gestaltet sich die Erstellung von Rahmen denkbar einfach, da fast alle möglichen Interaktionselemente über einen Werkzeugkasten abrufbar sind. Die Elemente werden per Maus an der gewünschten Stelle plaziert. Anschließend wird ihnen über einen Editor ereignisabhängiger Programmkode zugewiesen. Zusätzlich kann für jedes Formular mit einem speziellen Editor eine Menüzeile definiert werden.

Alle Interaktionselemente stellen in INGRES/Windows 4GL Objekte dar. Die Eigenschaften der Objekte können innerhalb der Ereignisprozeduren abgefragt und verändert werden. Dafür stehen eine Vielzahl von Methoden zur Verfügung [Ingres 90f].

4.2.2.3 Programmbeispiel

In diesem Abschnitt wird die Implementation des Programmbeispiels aus Anhang A an zwei Beispielen in INGRES/Windows 4GL vorgestellt. Das erste Beispiel demonstriert den Aufbau einer satzorientierten Benutzerschnittstelle und das zweite Beispiel zeigt den Aufbau einer tabellenorientierten Benutzerschnittstelle. In Anhang A.3 ist die Schemadefinition der Tabellen zu finden, die diesem Beispiel zugrunde liegen.

Satzorientierte Visualisierung

Im Gegensatz zu DBPL findet der Aufbau der Benutzerschnittstelle in INGRES/Windows 4GL nicht durch Bibliotheksaufrufe, sondern graphisch interaktiv statt. Die einzelnen Felder werden mit der Maus aus dem Werkzeugkasten ausgewählt und dann im Fenster plaziert. Aus der Schemadefinition einer Tabelle einer INGRES-Datenbank können automatisch die Felder der Maske erzeugt werden. Die Felder erhalten dann automatisch den Typ der entsprechenden Spalte der Tabelle. Sie müssen jedoch meist noch nachbearbeitet werden, da in INGRES nur wenige Typen zu Verfügung stehen.

Zum Beispiel existiert in INGRES kein Typ für boole'sche Werte. Solche Werte können über ganze Zahlen (*0* für *wahr* und *1* für *falsch*) in der Datenbank repräsentiert werden. An der Benutzerschnittstelle ist es jedoch wünschenswert, solche Werte über Ja/Nein-Auswahlboxen (*check boxes*) einzugeben. Die Änderung des Feldes sowie die Umsetzung einer ganzen Zahl aus der Datenbanktabelle in einen boole'schen Wert für die Benutzerschnittstelle und umgekehrt ist vom Anwendungsprogrammierer zu kodieren. In Abbildung 4.6 ist eine durch den Anwendungsprogrammierer angepaßte, satzorientierte Maske dargestellt.

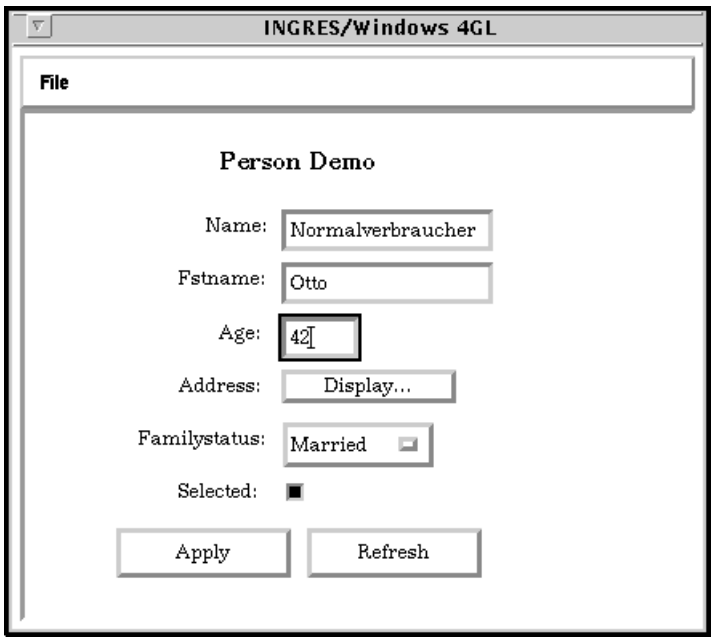


Abbildung 4.6: Benutzerschnittstelle von INGRES/Windows 4GL (satzorientiert)

Für drei Felder dieser Maske sind die durch das Datenwörterbuch vorgegebenen Repräsentationen zu ändern. Zum Beispiel ist es für einen Anwender einfacher, den Familienstatus über eine Auswahl mit entsprechenden Zeichenketten (*Single*, *Married* und *Divorced*) einzugeben, als ihn über Zahlen auszuwählen. Daher wird der in der Datenbank als Zahl dargestellte Familienstatus über eine Exklusivauswahl visualisiert. Das Feld *Selected* wird, wie weiter oben ausgeführt, als Ja/Nein-Feld visualisiert. Die Adresse einer Person wird, wie in relationalen Datenbanken üblich, über einen Fremdschlüssel ermittelt. An der Benutzerschnittstelle ist es sinnvoll, die Adresse als Knopf darzustellen. Betätigt der Anwender den Knopf, so kann über eine dann aufgerufene Funktion die Adresse ermittelt werden. Sie wird in einem weiteren Fenster dargestellt. Für die übrigen Felder ergeben sich durch die Benutzung der Schemainformationen geeignete Visualisierungen.

Tabellenorientierte Visualisierung

Wie schon erwähnt, können in INGRES/Windows 4GL die Schemadefinitionen von INGRES zum Erzeugen von Masken verwendet werden. Neben einer satzorientierten Anordnung der Felder ist es auch möglich, die Felder tabellenorientiert darzustellen. Aus der Schemadefinition werden die Namen für die Spaltenüberschriften sowie die Breitenangaben der einzelnen Spalten abgeleitet. Alle Angaben sind jedoch veränderbar. Ein Beispiel für eine solche Maske ist in Abbildung 4.7 zu finden.

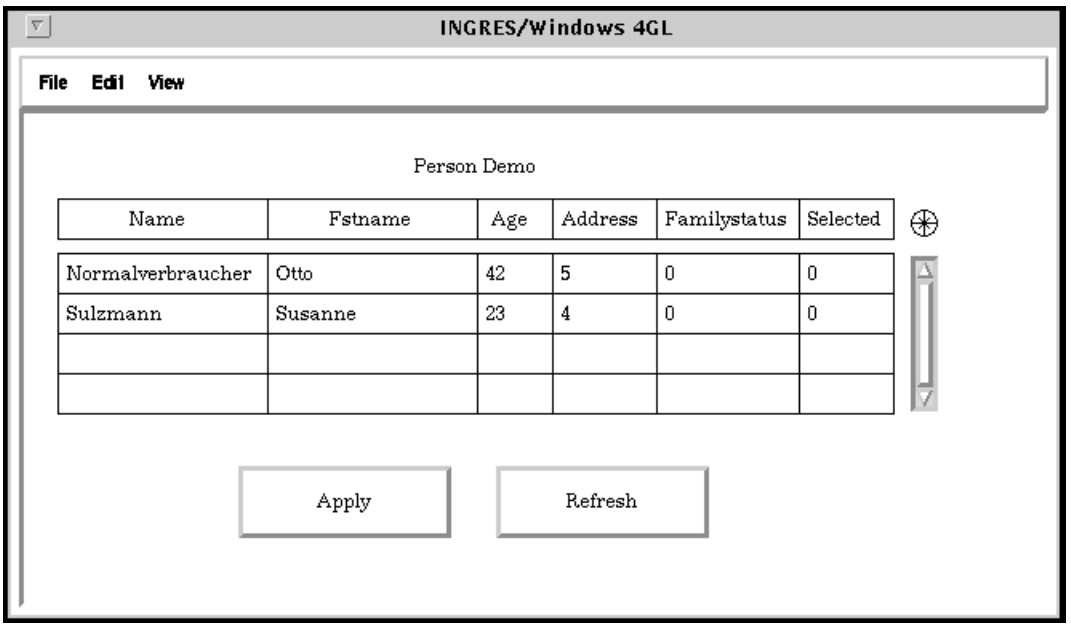


Abbildung 4.7: Benutzerschnittstelle von INGRES/Windows 4GL (tabellenorientiert)

Während es in DBPL möglich ist, ein Formular mit nur einem Bibliotheksaufruf anzuzeigen und zu füllen, so gestaltet sich das Füllen von Tabellen in INGRES/Windows 4GL sehr aufwendig. Tabellen müssen hier über eine Schleife in einem SQL-Befehl gefüllt werden. An dieser Stelle ist der konzeptuelle Bruch zwischen deklarativen Datenbankanfragen und der elementorientierten Vorgehensweise der Programmiersprache deutlich festzustellen.

```
select
   :personTable[i].name = p.name,
   :personTable[i].fstname = p.fstname,
   :personTable[i].age = p.age,
   :personTable[i].address = p.address
from person p
begin
   i = i + 1;
end
```

Mit dieser Anweisung wird die Verbindung zwischen den Spalten der INGRES-Datenbanktabelle und den Spalten der Tabelle an der Benutzerschnittstelle hergestellt, wobei *personTable* der Name der Tabelle im Formular ist. In jedem Schleifendurchlauf wird die Variable *i* inkrementiert und ein Element aus der Datenbanktabelle der entsprechenden Zeile der Bildschirmtabelle zugewiesen.

4.2.3 O_2

Das O_2-System [Deux 91; Bancilhon et al. 92; Soloviev 92] ist in einem 5-Jahres-Projekt bei Altaïr entwickelt worden. Die Forschungsgruppe ist von INRIA, Siemens-Nixdorf, Bull, CNRS und der Universität von Paris 1986 gegründet worden. Ziel der Gruppe war die Entwicklung und Implementation eines neuartigen Datenbanksystems. Nach Abschluß des Projekts wurde 1990 die Firma O_2 Technology gegründet. Sie übernahm die Weiterentwicklung, die Wartung und den Vertrieb des O_2-Systems.

4.2.3.1 Systemüberblick

O_2 ist ein objektorientiertes Datenbankmanagementsystem (*object-oriented database management system, OODBMS*) mit einer kompletten Entwicklungsumgebung. Die Entwicklungsumgebung vereinigt eine Programmiersprache, Datenbankfunktionalität sowie einen Generator zur Erstellung von Benutzerschnittstellen. Alle drei Komponenten basieren auf objektorientierten Konzepten und sind konform zu bestehenden Standards.

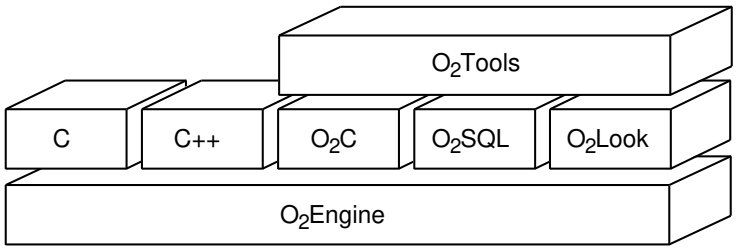


Abbildung 4.8: Funktionale Architektur des O_2-Systems

In Abbildung 4.8 ist die funktionale Architektur des O_2-Systems dargestellt. Die Basis des O_2-Systems bildet die O_2Engine, eine objektorientierte Datenbankmaschine. Zum O_2-System gehören neben der O_2Engine eine Programmiersprache der vierten Generation (O_2C), eine SQL-ähnliche Abfragesprache (O_2SQL), ein Benutzerschnittstellengenerator (O_2Look) sowie eine graphische Programmierumgebung (O_2Tools) mit interaktiver Kommandooberfläche (*shell*), Debugger, Schema- und Datenbankbrowser. Neben O_2C können auch C und C++ zur Anwendungsprogrammierung verwendet werden.

O_2 unterscheidet zwischen Werten (*values*) und Objekten (*objects*). Werte haben beliebige aus Konstruktoren für Tupel, Listen und Mengen sowie den Standardtypen zusammengesetzte Typen. Objekte gehören zu einer Klasse und haben neben einem Wert eine eindeutige Identität. Klassen haben einen Typ und beschreiben durch Methoden das Verhalten der zur Klasse gehörenden Objekte. Zusammengesetzte Werte und Objekte lassen sich auf zwei Arten konstruieren. Werte und Objekte können andere Objekte über deren Identität referenzieren (*shared objects*) oder sie beinhalten andere Werte, die nicht von anderen Objekten

referenziert werden können. Werte und Objekte sind automatisch persistent, wenn sie von einer benannten Wurzel aus einem Schema direkt oder indirekt erreichbar sind.

4.2.3.2 Benutzerschnittstelle

O_2Look ist die graphische Benutzerschnittstelle für das O_2-System [O_2Technology 93; Borras et al. 92]. Sie ermöglicht die Erstellung von OSF/Motif konformen Benutzerschnittstellen für das X Window System. O_2-Werte und -Objekte beliebiger Typen und Komplexität können mit generischen Editoren interaktiv und in Programmen auf dem Bildschirm anzeigt und manipuliert werden.

Alle O_2Look Repräsentationen (*presentations*) für Werte und Objekte bestehen aus zwei Komponenten, einer Maske (*mask*) und einem Menü (*menu*). Die Maske dient dem Anzeigen des Objekts auf dem Bildschirm. Sie kann eigenen Bedürfnissen angepaßt werden. Das Menü enthält alle für Objekte dieser Klasse definierten öffentlichen Methoden (*public methods*). Durch Auswahl eines Menüeintrags wird die entsprechende Methode für das angezeigte Objekt aufgerufen.

Für alle im O_2-System definierten Datentypen, z.B. *integer* und *string*, stehen generische Masken zur Verfügung. Für einige spezielle Typen, z.B. Texte und Pixelgraphiken, stehen eigene Masken zur Verfügung. Alle Masken dürfen beliebig miteinander kombiniert und ineinander verschachtelt werden, z.B. für zusammengesetzte Typen. Die vordefinierten Masken können um neue, anwendungsspezifische, ergänzt werden. Aus diesen generischen Repräsentationen erstellt O_2Look automatisch zur Laufzeit eine Maske für ein Objekt. Da das O_2-System eine dynamische Typüberprüfung unterstützt, ist es möglich, die Operationen zum Ausschneiden, Kopieren und Einfügen (vgl. Abschnitt 4.1.2) für Zahlen und Texte auf beliebige Typen von O_2 auszudehnen.

4.2.3.3 Programmbeispiel

In diesem Abschnitt wird die Implementation des Programmbeispiels aus Anhang A in O_2 vorgestellt. In diesem System können Benutzerschnittstellen auf zwei Arten aufgebaut werden. Zum einen durch Bibliotheksaufrufe aus den Hochsprachen C und C++und zum anderen durch dynamische Generierung zur Laufzeit.

Typinformationen stehen im O_2-System in Form von Klassendefinitionen zur Verfügung. Die Klassendefinition für das Programmbeispiel ist in Anhang A.4 zu finden. Im O_2-System kann eine Mengenvariable dieses Typs angelegt werden. Anschließend ist es in der interaktiven Entwicklungsumgebung möglich, direkt Objekte einzugeben. Dafür wird dynamisch eine Bildschirmmaske aus den Typinformationen abgeleitet. In Abbildung 4.9 ist die generierte Maske für das Programmbeispiel dargestellt.

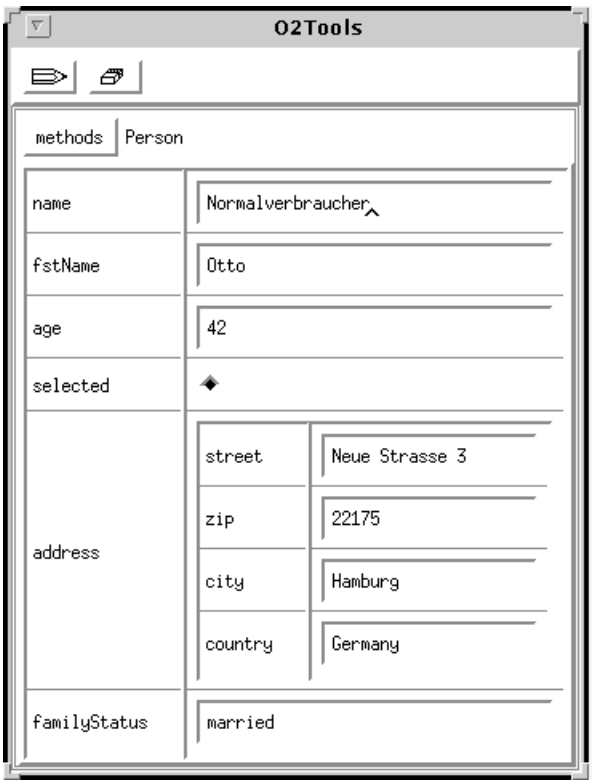


Abbildung 4.9: Benutzerschnittstelle von O_2 (satzorientiert)

Auffällig an dieser Darstellung ist die hierarchische Strukturierung der Komponenten an der Benutzerschnittstelle. Sie spiegelt die Struktur der zugrundeliegenden Klassendefinition wider. Der eingeschachtelte Verbund *Address* ist auch in der Maske sofort als solcher zu erkennen.

In der Kopfzeile der Maske sind zwei Ikonen zu finden. Mit dem Stift ist es möglich, das eingegebene oder veränderte Objekt in die Menge zu übernehmen und mit dem Radiergummi kann das Objekt aus der Menge gelöscht werden. Über den unter den beiden Ikonen dargestellten Knopf (*methods*) können alle für das Objekt definierten öffentlichen Methoden aufgerufen werden.

4.2.4 GemStone

Das GemStone-System [Bretl et al. 89; Butterworth et al. 91; Cattell 92] wird von der Servio Corp. seit 1987 entwickelt und vertrieben. Es ist das älteste kommerziell erhältliche System. Bei der Entwicklung standen Verteilungsaspekte (*distribution*), eine hohe Verfügbarkeit (*availability*) und Skalierbarkeit (*scalability*) des Systems im Vordergrund.

Da GemStone nicht für eine praktische Untersuchung zur Verfügung steht, basiert die Beschreibung allein auf der angegebenen Literatur. Darüber hinaus kann aus diesem Grund auch die Implementation des Programmbeispiels aus Anhang A nicht durchgeführt werden. Um trotzdem einen Eindruck der Benutzerschnittstelle dieses Systems zu vermitteln, ist in Abschnitt 4.2.4.2 ein Beispielformular von GemStone zu finden.

4.2.4.1 Systemüberblick

GemStone ist wie O_2 ein objektorientiertes Datenbankmanagementsystem mit einer Entwicklungsumgebung. Die visuelle Entwicklungsumgebung (*GemStone Object Development Environment*, GeODE) vereinigt, wie in Abbildung 4.10 zu sehen ist, verschiedene Werkzeuge. Die Werkzeuge unterstützen die Administration von Datenbanken, die Entwicklung von Datenbankschemata, die Erstellung von Bildschirmmasken sowie die visuelle Entwicklung von ganzen Programmen.

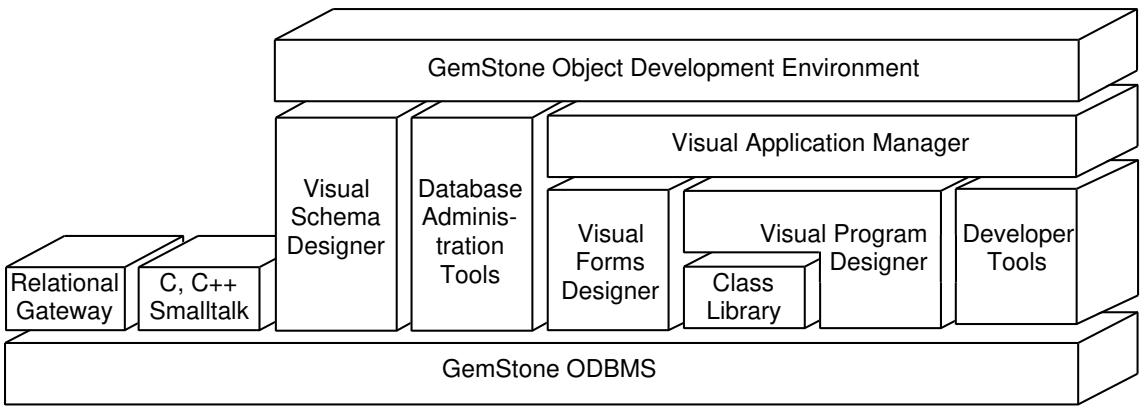


Abbildung 4.10: Funktionale Architektur von GemStone

Das besondere am GemStone-System ist die Möglichkeit, Programme zu entwickeln, ohne eine Zeile Kode zu schreiben. Dazu werden im *Visual Program Designer* Masken und Programmstücke (*program-building blocks*) visuell dargestellt. Durch die Verknüpfung von von Programmstücken und Masken können so komplette Anwendungen erstellt werden. Neue Programmstücke können jederzeit hinzugefügt und wiederverwendet werden. Die Datenbank ist vollständig in die visuelle Entwicklungsumgebung integriert. Alle bei der Anwendungsentwicklung anfallenden Daten, wie z.B. Masken oder Programmstücke, werden von ihr verwaltet.

Wie in Abbildung 4.10 weiter zu sehen ist, kann der Zugriff auf die Datenbank auch über Schnittstellen zu externen Sprachen erfolgen. Für C gibt es eine einfache Bibliothek, die den Zugriff auf die GemStone-Datenbank ermöglicht. Daneben kann von C-Programmen

auch über die DDL (*data definition language*) und DML (*data manipulation language*) von GemStone auf die Datenbank zugegriffen werden. C++-Programme können in der Datenbank gehalten werden und haben direkten Zugriff auf alle Objekte in der Datenbank. Die Schnittstelle zu Smalltalk besteht im wesentlichen aus einer Klassenbibliothek. Darüber hinaus existieren für GemStone Schnittstellen zu bestehenden relationalen Datenbanksystemen, wie z.B. Sybase, INGRES, ORACLE und Informix.

4.2.4.2 Benutzerschnittstelle

Bei der Benutzerschnittstelle verfolgt GemStone einen anderen Weg als die anderen in diesem Kapitel vorgestellten Systeme. GemStone-Anwendungen haben ein eigenes, von standardisierten Stilrichtlinien unabhängiges Aussehen und Verhalten (*look & feel*). Die Benutzerschnittstelle von GemStone-Anwendungen sieht auf allen unterstützten Systemen gleich aus und kann in der gleichen Weise vom Anwender bedient werden. Da die Anwendungen selbst in der Datenbank gehalten werden, laufen GemStone-Anwendungen ohne Änderung auf allen Soft- und Hardwareplattformen, auf die GemStone portiert worden ist.

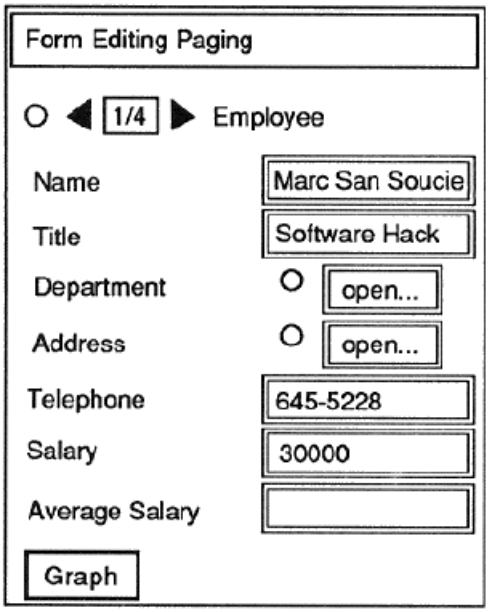


Abbildung 4.11: Formular von GemStone

Der *Form Designer* von GemStone ermöglicht die Erstellung von einfachen Masken, in denen Objekte aus der Datenbank angezeigt und verändert werden können. Standardmasken (vgl. Abbildung 4.11) werden durch Auswertung der Klassendefinitionen erzeugt. Sie können anschließend eigenen Bedürfnissen angepaßt werden.

4.2.5 VisualBASIC

Von Microsoft wird mit dem *Software Development Kit* (SDK) für Windows und der dazugehörigen Dokumentation ein Grundstock zur Verfügung gestellt, auf dem viele Programmiersprachen und Anwendungen für Windows aufsetzen. Ausgehend von diesem Fundament haben sich weitere Ansätze zur Anwendungsentwicklung etabliert [Siering 92b]. In diesem Abschnitt wird der Ansatz von VisualBASIC [Zoschke 91; Ehrmann 93a; Ehrmann 93b] weiter verfolgt. VisualBASIC wird seit ca. 1991 von der Microsoft Corp. entwickelt und vertrieben.

4.2.5.1 Systemüberblick

VisualBASIC [Maslo, Dittrich 93; Baloui 92] ist eine integrierte Entwicklungsumgebung (*Integrated Development Environment*, IDE) zum Erstellen, Testen und Ausführen von DOS- und Windows-Anwendungen. Die Entwicklungsumgebung besteht, wie in Abbildung 4.12 zu sehen ist, aus einem Maskengenerator (*form edit*), einem Quelltext-Editor, einem Interpreter, einem Übersetzer, einer interaktiven und kontextsensitiven Hilfe, einem Debugger, einer Projektverwaltung und einem Werkzeugkasten (*toolbox*), der sich mit externen Werkzeugen (*custom controls*) ergänzen läßt.

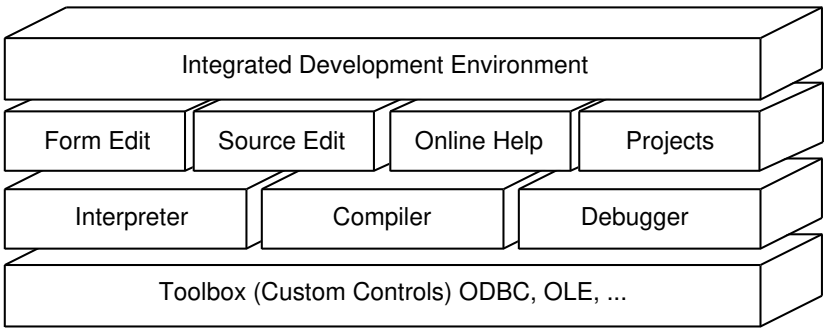


Abbildung 4.12: Komponenten der VisualBASIC-Entwicklungsumgebung

Programme können direkt in der Entwicklungsumgebung eingegeben, übersetzt, getestet und ausgeführt werden. Alle eingegebenen Zeilen werden sofort auf eine korrekte Syntax hin überprüft. Ist eine Anwendung fertiggestellt, so kann mit Hilfe des Übersetzers ein ohne die Entwicklungsumgebung lauffähiges Programm erzeugt werden. Zusätzlich zum übersetzten Programm wird dann eine Laufzeitbibliothek benötigt.

Im Unterschied zu herkömmlichen BASIC-Dialekten (*Beginners All Symbolic Instruction Code*) gibt es in VisualBASIC kein Hauptprogramm. Die Programmierung gestaltet sich weitgehend ereignis- und objektorientiert. Die Ursachen für die Unterschiede sind zum großen Teil auf Windows zurückzuführen. VisualBASIC-Programme können alle Windows-Schnittstellenkomponenten nutzen und auf die sogenannte Windows-API (*Application Pro-*

gramming Interface) zurückgreifen. Funktionen, die im Sprachumfang nicht zur Verfügung stehen, lassen sich durch dynamische Bibliotheken (*Dynamic Link Library*, DLL) ergänzen. Solche Bibliotheken können z.B. in Assembler oder C geschrieben werden. In VisualBASIC selbst können keine Bibliotheken erstellt werden.

Zu VisualBASIC gehört keine eigene Datenbank. Auf externe Datenbanksysteme verschiedener Hersteller kann jedoch mit ODBC-Treibern (*Open Database Connectivity*) über einheitliche Funktionsaufrufe zugegriffen werden. Für jedes Datenbanksystem gibt es einen eigenen ODBC-Treiber in Form einer Bibliothek, über den die Anwendung auf die Funktionalität der jeweiligen Datenbank zugreift. Die abstrahierende Schnittstelle der Treiber erlaubt den Zugriff auf Datenbanken, ohne sich um die Eigenheiten des jeweiligen Datenbanksystems kümmern zu müssen [Siering 92a; Borchert 93].

4.2.5.2 Benutzerschnittstelle

Die Anwendungsentwicklung mit VisualBASIC kann in mehrere Schritte unterteilt werden. Begonnen wird stets mit einem leeren Fenster, in VisualBASIC Formular (*form*) genannt. Mit Hilfe des interaktiven Maskengenerators wird zunächst das optische Erscheinungsbild einer Anwendung entworfen. Dazu können die gewünschten Schnittstellenkomponenten (*controls*) in den Formularen frei positioniert werden. Erst nach der Definition der Benutzerschnittstelle werden allen Objekten in Abhängigkeit von ihrer späteren Aufgabe Eigenschaften (*properties*) zugewiesen. Im Gegensatz zu den Schnittstellenkomponenten können die Eigenschaften nicht nur visuell, sondern auch durch Programmkode festgelegt und verändert werden. Allen Programmteilen wird im nächsten Schritt ereignisabhängiger prozeduraler Kode zugewiesen, wobei Namen und Ereignisse von VisualBASIC automatisch vorgegeben werden.

Neben den aus anderen Entwicklungswerkzeugen bekannten Schnittstellenkomponenten gibt es im Werkzeugkasten von VisualBASIC eine Reihe fertiger Standarddialoge. In der aktuellen Version gibt es Standarddialoge zur Dateibehandlung, zum Drucken, zur Zeichensatzauswahl und zum Blättern und Ändern in externen Datenbanken. Zusätzlich können die vorhandenen Schnittstellenkomponenten um benutzerdefinierte ergänzt werden. Solche Komponenten sind dann genau wie die vordefinierten über den Werkzeugkasten abzurufen.

Der Datenaustausch mit anderen Anwendungsprogrammen kann in Windows auf drei unterschiedlichen Wegen erfolgen. Der einfachste Weg ist die Verwendung der Zwischenablage (*clipboard*) durch Operationen wie Ausschneiden (*cut*), Kopieren (*copy*) und Einfügen (*paste*) [Lauer 92]. Die Zwischenablage ist ein Speicherbereich, der allen Windows-Programmen zur Verfügung steht. Jedes Programm kann dort Daten ablegen oder entnehmen. Wichtig ist die Übereinkunft zwischen den Kommunikationspartnern über das Format der auszutauschenden Daten. Die Aufnahmefähigkeit der Zwischenablage ist praktisch unbegrenzt, da der Anwender neben den Standardformaten auch eigene Formate einbringen kann. Durch die Operationen Ausschneiden und Kopieren werden Daten aus der Anwendung kopiert bzw. ausgeschnitten und in die Zwischenablage übernommen. Mit der Einfügen-Operation werden die Daten aus der Zwischenablage in die Anwendung kopiert [Salcher 93].

Eine weitere Möglichkeit ist der dynamische Datenaustausch (*Dynamic Data Exchange*, DDE). Diese Art des Datenaustauschs hat gegenüber der Verwendung der Zwischenablage einige Vorteile. Werden z.B. Daten über die Zwischenablage ausgetauscht, so muß dieser Austausch bei Änderung der Quelldaten wiederholt werden. Beim dynamischen Datenaustausch hingegen wird jede Änderung der Quelldaten automatisch in die Zieldaten übernommen. Dabei wird noch zwischen automatischer Aktualisierung (*hot links*) und Aktualisierung auf Nachfrage (*code links*) unterschieden. Die dafür notwendige Kommunikation regelt das DDE-Protokoll [Lauer 93a]. Über das DDE-Protokoll können Anwendungen auch ferngesteuert werden. Dabei sendet ein VisualBASIC-Programm Zeichenketten und Tastendrücke an die Anwendung, die ferngesteuert werden soll, und simuliert damit dort Eingaben.

Die dritte Möglichkeit ist das Einbetten und Verknüpfen von Objekten (*Object Linking and Embedding*, OLE 2.0) in andere Anwendungsprogramme. OLE ist ein Standardprotokoll, um Befehle und Daten auszutauschen [Lauer 93b; Hüskes 94]. Dabei können VisualBASIC Programme sowohl als Auftraggeber als auch als Datenlieferant dienen. OLE ermöglicht genau wie der DDE das Einbinden von Daten eines Programms in ein anderes. Im Gegensatz zum DDE ist es dabei aber nicht notwendig, daß das Programm, welches die Daten liefert, gestartet ist. Außerdem wird die Aufbereitung der Daten dem Server-Programm überlassen. Zusätzlich ist eine sogenannte *Vor-Ort-Aktivierung* (*inplace-editing*) möglich. Diese erlaubt die Bearbeitung der eingebundenen Daten direkt im Klienten-Programm.

4.2.5.3 Programmbeispiel

In diesem Abschnitt wird die Implementation des Programmbeispiels aus Anhang A in VisualBASIC vorgestellt. Dabei werden wieder zwei Ansätze aufgezeigt. Das erste Beispiel zeigt eine satzorientierte Maske und das zweite Beispiel zeigt eine tabellenorientierte Maske.

Satzorientierte Visualisierung

Alle Bildschirmmasken werden in VisualBASIC interaktiv zusammengebaut. Dafür stehen in einem Werkzeugkasten eine große Anzahl einfacher, aber auch hochspezialisierte Schnittstellenkomponenten zur Verfügung. Die einzelnen Komponenten werden in dem Formular über ein Raster plaziert. Zu jeder Schnittstellenkomponente gehört ein Satz von Eigenschaften. Sie bestimmen z.B. die Ausrichtung, den Namen und die Farbe der Komponente.

Die Schnittstellenkomponenten sind nicht an einen festen Typ gebunden. Zum Beispiel gibt es keine eigene Schnittstellenkomponente zum Anzeigen einer ganzen Zahl. Verwendet werden kann dazu aber das Textfeld. Der Wert muß jedoch beim Ausgeben und Lesen entsprechend konvertiert werden. Das gleiche gilt auch für andere Datentypen, da die von VisualBASIC vorgegebenen Schnittstellenkomponenten den Windows-Schnittstellenkomponenten entsprechen.

Die Verbindung zwischen einer Schnittstellenkomponente und dem Anwendungsprogramm muß vom Anwendungsprogrammierer hergestellt werden. Der Datenaustausch mit den Komponenten erfolgt dabei meist über Zeichenketten. Für das Programmbeispiel kann im Formulareditor die in Abbildung 4.13 dargestellte Maske aufgebaut werden.

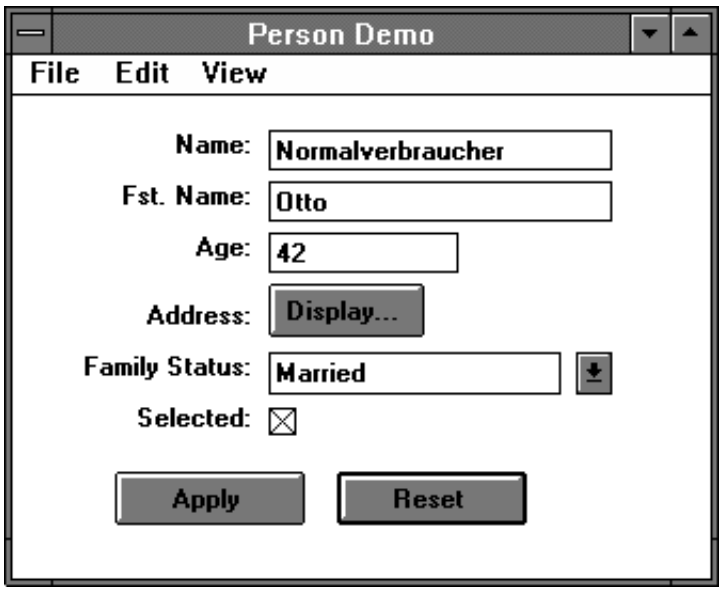


Abbildung 4.13: Benutzerschnittstelle von VisualBASIC (satzorientiert)

Für jede Komponente wird wird automatisch Kode generiert. Für einen Knopf wird z.B. folgendes Kodesegment vorgegeben:

Sub *Command_Click ()*

End *Sub*

Der Anwendungsprogrammierer kann in dieser Prozedur spezifizieren, was ausgeführt werden soll, wenn der Anwender den Knopf drückt. *Command* ist der Name des Knopfes und *Click* ist das erwartete Ereignis.

Die bisher erstellte Anwendung kann zu jeder Zeit getestet werden. Treten Fehler im prozeduralen Kode auf, so verzweigt die Entwicklungsumgebung automatisch an die Stelle, an der der Fehler aufgetreten ist.

Tabellenorientierte Visualisierung

Das zweite Beispiel zeigt eine tabellenorientierte Maske (vgl. Abbildung 4.14). Sie kann über eine spezialisierte Komponente (*custom control*) aus dem Werkzeugkasten aufgebaut werden. Als Eigenschaften kommen hier z.B. die Anzahl der Spalten und Zeilen in Betracht.

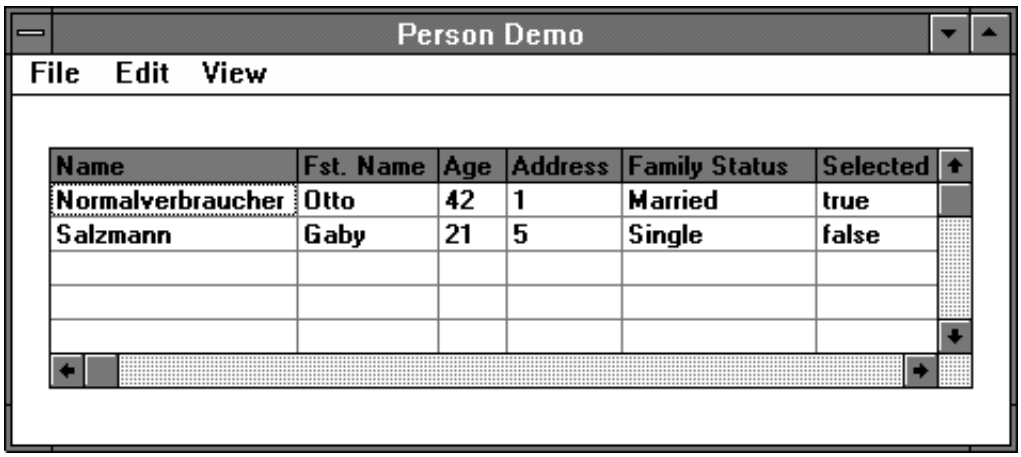


Abbildung 4.14: Benutzerschnittstelle von VisualBASIC (tabellenorientiert)

Wieviel Programmkode notwendig ist, um eine solche Tabelle zu füllen, hängt von der Mächtigkeit der Komponente ab.

4.3 Zusammenfassung

In diesem Abschnitt werden die Ergebnisse aus der Analyse der im vorherigen Abschnitt vorgestellten Systeme noch einmal kurz zusammengefaßt. Zusätzlich werden einige Kritikpunkte angegeben. Eine tabellarische Übersicht der Ergebnisse befindet sich in Anhang B. Diese Zusammenfassung dient als Grundlage für die Anforderungen an die typsichere generische Bibliothek zur Datenvisualisierung in Tycoon (vgl. Kapitel 7).

Die Ergebnisse sind nach den in Abschnitt 4.1.2 vorgestellten Schwerpunkten gegliedert. Dabei wird auf die beiden ersten Schwerpunkte, Hard- und Softwareplattformen sowie Systemkomponenten, nicht weiter eingegangen. Die Ergebnisse der Untersuchungen hinsichtlich dieser Schwerpunkte sind in der Tabelle B.3 bzw. den Tabellen B.4 bis B.7 zu finden.

Visualisierungsmöglichkeiten

Die Analyse der Visualisierungsmöglichkeiten verschiedener Systeme in diesem Kapitel hat einige Unterschiede bei den verfolgten Ansätzen ergeben. Grundsätzlich existieren die in Abbildung 4.15 dargestellten Ansätze.

Bildschirmmasken von datenintensiven Anwendungen werden in statisch erzeugte und dynamisch zur Laufzeit generierte Masken aufgeteilt. Statische Masken können entweder interaktiv aus den in einem Werkzeugkasten zur Verfügung gestellten Schnittstellenkomponenten oder durch Bibliotheksaufrufe erzeugt werden. Beide Ansätze lassen sich noch einmal

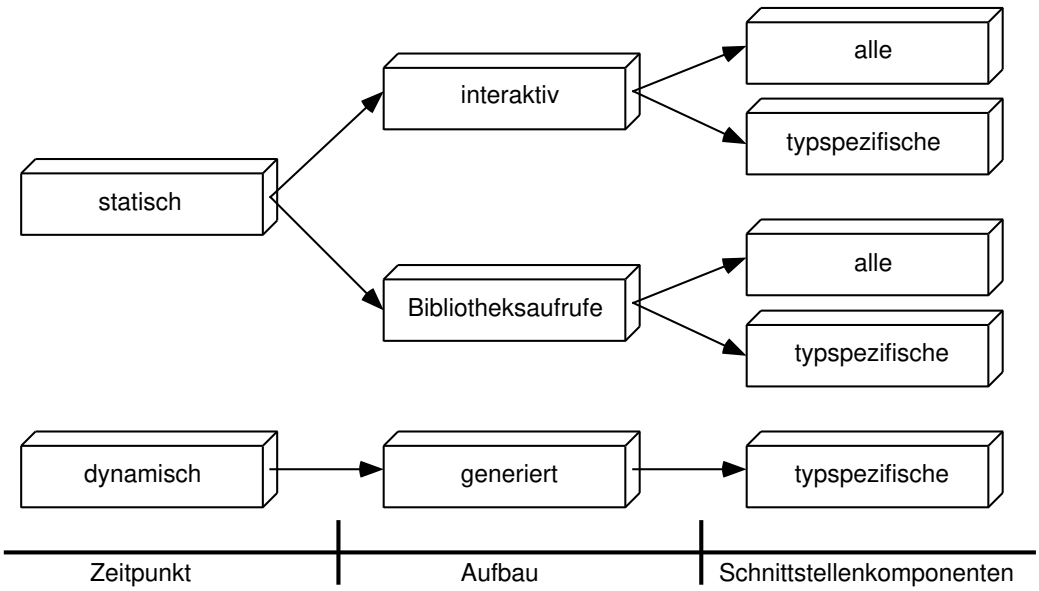


Abbildung 4.15: Ansätze zur Datenvisualisierung

aufteilen. Es kann jeweils zwischen der Bereitstellung aller Schnittstellenkomponenten des zugrundeliegenden Fenstersystems und dem Anbieten spezieller, den Typen der zugrundeliegenden Programmiersprache entsprechenden, Komponenten unterschieden werden.

Die dynamische Generierung läßt sich zur automatischen Generierung der den Typen der zu visualisierenden Daten entsprechenden Komponenten verfeinern. Dabei tritt ein generelles Visualisierungsproblem zutage. Vor allem in relationalen Datenbanksystemen kann die Semantik der Daten nicht mit abgespeichert werden. Unterstützt die Datenbank z.B. keine boole'schen Werte, so können solche Werte durch Anwendungslogik auf ganze Zahlen abgebildet werden. Der Transfer muß jedoch auch für die Darstellung an der Benutzerschnittstelle durchgeführt werden. Bei einer generischen Visualisierung ist dies nicht möglich, da die im Datenwörterbuch zur Verfügung stehenden Informationen nicht ausreichen, die Werte geeignet zu visualisieren.

Im O_2-System stehen dem Anwender die umfangreichsten Visualisierungsmöglichkeiten zur Verfügung. Bildschirmmasken können in diesem System sowohl durch Bibliotheksaufrufe erzeugt als auch dynamisch zur Laufzeit generiert und nachträglich verändert werden. Die Möglichkeit, in INGRES/Windows 4GL Schemadefinitionen der Datenbank zum Aufbau der Masken zu verwenden, ist auch in anderen Systemen, z.B. in O_2 und GemStone, zu finden. Während in INGRES/Windows 4GL die Informationen nur zur statischen Erzeugung von Masken verwendet werden können, verwenden O_2 und GemStone diese Informationen zum dynamischen Generieren von Masken.

Die Verwendung von VisualBASIC vereinfacht für den Entwickler nur die Gestaltung der Benutzerschnittstelle. Bezüglich Abstraktion, Wiederverwendung und Wartung ist VisualBASIC den anderen Systemen deutlich unterlegen. Ein weiterer Nachteil ist das nur statische Anlegen von Schnittstellenkomponenten.

DBPL ist mit den anderen Systemen nicht direkt vergleichbar, da es nur textuelle Masken unterstützt. In dieser Kategorie stehen aber mit den beiden Modulen *Form* und *SetForm* sowie mit dem generischen Relationenbrowser *Assist* umfangreiche und mächtige Datenvisualisierungsmöglichkeiten zur Verfügung.

Interoperabilität

DBPL-Programme besitzen textuelle Masken. Aus diesem Grund sind die Interoperabilitätsmöglichkeiten sehr beschränkt. Die anderen untersuchten Systeme sind graphisch orientiert und daher mit einer Maus bedienbar. Solche Systeme offerieren bessere Interoperabilitätsmöglichkeiten. Die umfangreichsten Datenaustauschmöglichkeiten sind in VisualBASIC zu finden. Sie sind jedoch nicht speziell VisualBASIC zuzuordnen, sondern haben ihren Ursprung in der Windows-Umgebung.

Anforderungen

Aus der Analyse der verschiedenen Systeme in diesem Kapitel ergeben sich die folgenden Anforderungen an die typsichere generische Datenvisualisierung in Tycoon:

- Typsicherer Aufbau von satzorientierten Bildschirmmasken durch Bibliotheksaufrufe. Dafür ist es notwendig, alle in TL vorhandenen Basisdatentypen und Typkonstruktoren an der Benutzerschnittstelle geeignet zu repräsentieren.
- Generische Visualisierung von Kollektionen.
- Realisierung eines anwendungsübergreifenden Datenaustauschs.
- Erzeugung beliebiger graphischer Benutzerschnittstellen für Tycoon-Programme durch Bibliotheksaufrufe.
- Anbindung von interaktiv erzeugten Benutzerschnittstellen.

In den Kapiteln 6 und 7 werden zwei Bibliotheken, vorgestellt die gemäß der aufgestellten Anforderungen entwickelt wurden.

5. Kommerzielle Dienste einer speziellen Anwendungsumgebung

In diesem Kapitel wird eine Anwendungsumgebung mit ihren kommerziellen Diensten vorgestellt. Die Umgebung stellt die Entwicklungsplattform für die in den nachfolgenden Kapiteln vorgestellten generischen Bibliotheken dar. Die Zusammensetzung der Entwicklungsplattform ergibt sich zum einen aus der am Arbeitsbereich DBIS vorhandenen Hard- und Software und zum anderen aus dem Ergebnis der Suche nach einem geeigneten Entwicklungswerkzeug für graphische Benutzerschnittstellen in Kapitel 2.

Die Erstellung graphischer Benutzerschnittstellen für die OpenWindows-Anwendungsumgebung wird durch verschiedene Werkzeuge und Dienste unterstützt. Unterschieden wird dabei zwischen X11- (*X Window System Version 11*) und NeWS- (*Network extensible Window System*) basierten Werkzeugen. Grundlage des netzwerkfähigen Fenstersystems NeWS ist die Seitenbeschreibungssprache PostScript und das damit verbundene, gegenüber X11 mächtigere Graphikmodell. Daraus ergeben sich eine Reihe von Vorteilen für NeWS-basierte Werkzeuge. Ein spezielles Werkzeug ist das NeWS Toolkit (*The NeWS Toolkit*, TNT). Es implementiert viele Schnittstellenkomponenten der OPEN LOOK-Spezifikation für graphische Benutzerschnittstellen. Dieses Kapitel stellt alle aufgeführten Dienste ausführlich vor und beschreibt deren Zusammenspiel.

5.1 Die Seitenbeschreibungssprache PostScript

Die Sprache PostScript [Adobe Systems 90] ist eine von Adobe Systems Inc. seit 1982 entwickelte, dynamische, kellerspeicherbasierte und interpretative Seitenbeschreibungssprache. Die Sprache ermöglicht die geräte- und auflösungsunabhängige Beschreibung von *Seiten* für Ausgabegeräte wie Drucker oder Bildschirme. Dazu enthält die Sprache neben graphikspezifischen Operatoren auch eine Reihe allgemeiner Operatoren, die PostScript zu einer vielseitig verwendbaren allgemeinen Programmiersprache (*general purpose programming language*) machen.

5.1.1 Graphikmodell

Die Seitenbeschreibungssprache PostScript basiert auf dem *stencil/paint imaging model*, einem graphischen Modell, das dem Malen mit Farbe auf Papier nachempfunden ist. Graphiken werden in diesem Modell durch Pfadkonstruktionen, Mal- und Transformationsfunktionen beschrieben. Durch die Beschreibung aller auf einer Seite vorhandenen Informationen als Folge von abstrakten graphischen Objekten wird die Unabhängigkeit vom Ausgabegerät und der Auflösung erreicht.

Graphische Objekte sind z.B. Buchstaben, Texte, geometrische Figuren und gerasterte Bilder. Außer den gerasterten Bildern sind alle graphischen Objekte, also auch die Buchstaben der Zeichensätze (*fonts*), aus mathematischen Pfaden (*paths*) in geräteunabhängigen Koordinatensystemen aufgebaut. Solche durch Pfade beschriebene Objekte können ohne Qualitätsverlust verschiedenen Transformationen unterzogen werden. Nach Ausführung aller Transformationen wird der Pfad des Objekts mit einer Linie beliebiger Stärke im Zielkoordinatensystem des Ausgabegeräts nachgezogen. Anschließend kann das Objekt noch wahlweise mit unterschiedlichen Mustern in beliebigen Farben gefüllt werden.

5.1.2 Programmierung

Wurde PostScript ehemals hauptsächlich als Seitenbeschreibungssprache konzipiert, so ermöglichen die in PostScript enthaltenen Merkmale von allgemeinen Programmiersprachen, wie Polymorphismus, Funktionen höherer Ordnung und die dynamische Erweiterbarkeit, den Einsatz von PostScript als vollwertige Programmiersprache. Daneben läßt sich PostScript als interaktives System zur Bedienung von Ausgabegeräten einsetzen und hat sich im Graphikbereich als Datenaustauschformat zwischen verschiedenen Anwendungsprogrammen etabliert.

Interaktionsmodelle

Aus der Anwendung von PostScript ergeben sich unterschiedliche Interaktionsmodelle. Zwei Modelle sind in Abbildung 5.1 dargestellt. Grundlegendes Modell ist das *Drucker-Modell* (*printer model*). Darauf aufbauend gibt es das *interaktive Modell* (*interactive programming language model*).

Im Drucker-Modell werden die Seitenbeschreibungen nicht von Anwendern, sondern von Anwendungsprogrammen, wie Text- oder Graphikprogrammen, automatisch erzeugt. Die Beschreibung einer Seite (*page description*) wird komplett an ein Ausgabegerät übermittelt. Der PostScript-Interpreter im Ausgabegerät erstellt dann anhand der Befehle zum *Konstruieren* und *Malen* von Graphiken ein *geräteabhängiges* Rasterbild, welches anschließend ausgegeben wird.

Das interaktive Modell unterscheidet sich vom Drucker-Modell dahingehend, daß Daten nicht einmalig vom Anwendungsprogramm zum Ausgabegerät, sondern in einer Sitzung (*session*) fortwährend in beiden Richtungen übertragen werden. Dafür muß zunächst eine

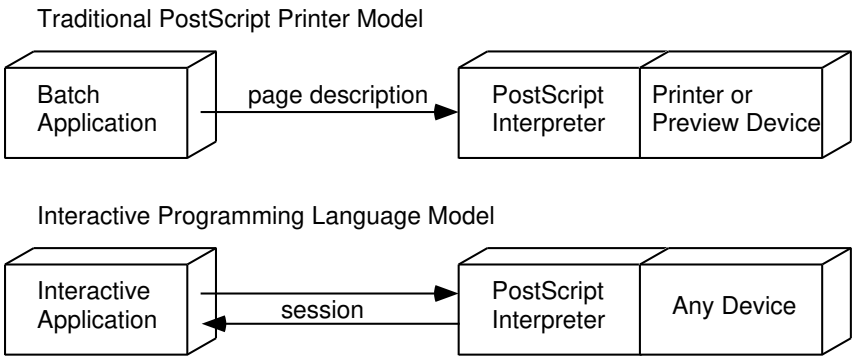


Abbildung 5.1: PostScript-Interaktionsmodelle [Adobe Systems 90]

Verbindung zwischen dem Anwendungsprogramm und dem PostScript-Interpreter im Ausgabegerät aufgebaut werden. In einer solchen Sitzung wird der PostScript-Interpreter direkt durch die von der Anwendung ausgegebenen Befehle angesprochen. Er führt sie sofort aus und gibt bei Bedarf entsprechende Rückmeldungen an das Anwendungsprogramm. Diese Vorgehensweise ist nicht nur für Test- und Experimentierarbeiten günstig. Sie ist die Voraussetzung für den Einsatz von PostScript als Programmiersprache für Benutzerschnittstellen.

Kellerspeicher und Postfixnotation

PostScript ist kellerspeicherbasiert, d.h. zur Übergabe und Speicherung von Objekten werden polymorphe Kellerspeicher (*stacks*) verwendet. Ein polymorpher Kellerspeicher ist ein Teil des Speichers, in dem Objekte verschiedener Typen abgelegt und in einer LIFO-Reihenfolge (*last-in first-out*) wieder abgerufen werden können. Daraus ergibt sich eine Notation (*postfix notation*), die von der in anderen Programmiersprachen verwendeten Infixnotation abweicht. Bei der Postfixnotation werden zunächst alle Argumente auf einem Kellerspeicher abgelegt. Erst anschließend wird der betreffende Operator aufgerufen. Ein einfaches Beispiel ist die Addition zweier Zahlen:

40 60 ***add***

Die beiden Objekte *40* und *60* werden auf dem Kellerspeicher abgelegt. Nachdem alle Argumente für den Operator **add** auf dem Kellerspeicher liegen, kann der Operator ausgeführt werden. Er holt sich die beiden Argumente vom Kellerspeicher, addiert sie und legt das Ergebnis wieder dort ab.

Konzepte

Alle Objekte in PostScript haben einen Typ. Dazu gibt es ein reiches Typsystem. Neben den von anderen Programmiersprachen her bekannten konventionellen Datentypen gibt es

in PostScript noch weitere spezielle Datentypen, wie z.B. Wörterbücher (*dictionaries*) und Dateien (*files*), sowie einen polymorphen Nulltyp.

Die Sprache PostScript stellt verschiedene Kontrollstrukturen zur Steuerung des Programmablaufs zur Verfügung. Es gibt Operatoren zur Programmierung von bedingter Ausführung, typabhängigem Verhalten, Schleifen und Ausnahmesituationen. Auch rekursive Programmierung ist möglich.

Eine Reihe von polymorphen Operatoren und Funktionen höherer Ordnung erlauben die Beschreibung typunabhängigen Verhaltens. Zum Beispiel kann mit dem Operator **forall** sowohl über Wörterbücher als auch über Felder iteriert werden und mit dem Operator **put** können Zeichen in einer Zeichenkette, Objekte in einem Feld sowie Paare aus Schlüssel und Wert in einem Wörterbuch gespeichert werden.

Weitere über den Umfang dieses Abschnitts hinausgehende Informationen sind in [Adobe Systems 90; Adobe Systems 91; Adobe Systems 88] und in [Holzgang 90; Kirch, Müßig 92] zu finden.

5.2 Das Fenstersystem NeWS

NeWS ist ein von Sun Microsystems ab 1984 entwickeltes netzwerkfähiges Fenstersystem [SunSoft 92a; Gosling et al. 89]. Es macht das mächtige Graphikmodell der Seitenbeschreibungssprache PostScript einem verteilten Fenstersystem zugänglich. NeWS ist unabhängig von der verwendeten Hardware und dem Betriebssystem. Im Gegensatz zum nur statisch vom Anbieter erweiterbaren Fenstersystem X11 [Jones 89; Davison et al. 92] kann die Funktionalität des NeWS von Anwendungsprogrammen durch Hinzufügen von Kode (*download*) dynamisch erweitert werden.

5.2.1 Grundbegriffe

In diesem Abschnitt werden die im Zusammenhang mit NeWS häufig verwendeten Begriffe *canvas*, *class*, *process*, *event* und *package* kurz erläutert.

Canvas

Ein Canvas ist eine nicht notwendigerweise rechteckige und zusammenhängende Fläche auf einem Bildschirm, in der Anwendungsprogramme Informationen darstellen können. Solche Flächen können sowohl ineinander verschachtelt sein, als sich auch überlappen. Durch eine hierarchische Struktur können die Beziehungen zwischen den Ausgabeflächen einer Anwendung abgebildet werden.

Class

NeWS stellt einen flexiblen objektorientierten Klassenmechanismus zur Verfügung. Er erlaubt die eigene Definition von Klassen und die Verwendung vorhandener Klassenbibliothe-

ken, wie z.B. des NeWS Toolkits (vgl. Abschnitt 5.4). Es werden sowohl Einfach- als auch Mehrfachvererbung unterstützt.

Process

NeWS-Prozesse sind leichte Prozesse (*lightweight processes*). Sie laufen im X11/NeWS Server ab. Die Erzeugung solcher Prozesse verbraucht wenig Ressourcen, wie CPU-Zeit und Hauptspeicher. Alle leichten Prozesse teilen sich einen gemeinsamen Adreßbereich.

Event

Ereignisse (*events*) repräsentieren Nachrichten zwischen NeWS-Prozessen. Ereignisse werden entweder durch die Interaktion eines Benutzers vom X11/NeWS Server oder durch andere NeWS-Prozesse generiert. Wie in Abbildung 5.6 auf Seite 67 zu sehen ist, werden Ereignisse z.B. von der Tastatur oder der Maus ausgelöst. Die Nachrichten beinhalten Informationen über die Art des Ereignisses, z.B. Tastendruck oder Mausklick, den Auslöser des Ereignisses, z.B. Taste a oder *rechte* Maustaste sowie die Auslöseposition und den Auslösezeitpunkt.

Ereignisse sind mit den Ausgabeflächen von Anwendungen in einer Liste (*interest list*) assoziiert. Alle anfallenden Ereignisse werden in eine globale Warteschlange (*event queue*) eingereiht. Von dieser werden sie auf die lokalen Warteschlangen der am Ereignis interessierten Anwendungen verteilt und dort ausgewertet.

Package

Der Paketmechanismus (*package mechanism*) dient der Modularisierung des Namensraums im X11/NeWS Server. Der Mechanismus ähnelt dem *shared library* Konzept des Betriebssystems SunOS. Auf dem NeWS aufbauende Werkzeuge, wie z.B. das NeWS Toolkit, sind in externe Pakete ausgelagert. Sie müssen bei Bedarf explizit hinzugeladen werden (vgl. Abschnitt 5.4).

5.2.2 Programmierung

Die Grundlage der NeWS-Sprache bildet die in Abschnitt 5.1 vorgestellte Sprache PostScript. Sie wird verwendet, um Texte und Bilder auf einem Bildschirm auszugeben. Daneben sind in der Sprache viele Erweiterungen zu finden, die der Interaktion (Bildschirmausgaben sowie Maus- und Tastatureingaben) und der Mehrprozeßfähigkeit in einem Fenstersystem Rechnung tragen. Die NeWS-Programme werden vom X11/NeWS Server interpretiert und ausgeführt.

Es gibt zwei Programmierschnittstellen zum NeWS. Die eine ist das C Client Interface, eine C-Bibliothek. Die andere ist das objektorientierte NeWS Toolkit, das im Abschnitt 5.4 ausführlich beschrieben wird. In diesem Abschnitt wird die Funktion einiger grundlegender Operatoren beschrieben, die notwendig sind, um Methoden aufzurufen und Daten zwischen einem Anwendungsprogramm und dem X11/NeWS Server auszutauschen.

Modularisierung

NeWS benutzt zur Modularisierung einen Paketmechanismus. Aus diesem Grund müssen nach einem Verbindungsaufbau zwischen Anwendungsprogramm und X11/NeWS Server zuerst die benötigten Pakete, z.B. *NeWS* geladen werden. In diesem Paket befinden sich alle NeWS-spezifischen Operatoren.

/NeWS 3 0 ***findpackage beginpackage***

Der Operator **findpackage** lädt das Paket mit der gewünschten Versionsnummer und **beginpackage** macht es zugänglich. Vor dem Abbau einer Verbindung zum X11/NeWS Server sollten die geladenen Pakete mit dem **endpackage** Operator wieder aus dem Speicher entfernt werden.

Methodenaufruf und Pseudovariablen

In objektorientierten Sprachen werden Unterprogramme (Methoden) nicht durch Prozeduraufrufe sondern durch das Versenden von Nachrichten an Objekte aufgerufen. Zum Versenden von Nachrichten gibt es den Operator **send**.

args meth obj ***send***

Diese Sequenz ruft die Methode *meth* in dem Objekt *obj* mit den Parametern *args* auf.

Es gibt zwei *Pseudovariablen*, die für spezielle Empfänger von Methoden benutzt werden. Die Empfänger solcher so versandten Methoden ergeben sich aus dem Aufrufkontext. Soll eine Methode aus dem gleichen Objekt aufgerufen werden, so wird die Pseudovariable **self** benutzt.

args meth ***self send***

Die andere Pseudovariable (**super**) dient dazu, eine in der Subklasse überschriebene Methode der Superklasse aufzurufen.

args meth ***super send***

Kommunikation

Für jede Verbindung eines Anwendungsprogramms zum X11/NeWS Server wird ein eigener NeWS-Prozeß erzeugt. Dieser liest die vom Anwendungsprogramm kommenden typisierten Daten als NeWS-Programme und führt sie aus. In der Gegenrichtung können beliebige Daten versendet werden. Dazu existieren die beiden Operatoren **typedprint** und **tagprint**.

20 ***typedprint***
(Peter) ***typedprint***

Der polymorphe Operator **typedprint** schickt dem eigentlichen Wert einen Typidentifizierungskode vorweg. Anhand dieses Kodes kann das Anwendungsprogramm den nachfolgenden Wert aus der Standardrepräsentation extrahieren. Zum besseren Verständnis sei noch angemerkt, daß Zeichenketten in PostScript nicht durch Anführungszeichen, sondern durch runde Klammern begrenzt werden.

t17 ***tagprint***

Mit dem Operator **tagprint** werden Assoziatoren (*tags*) zum Anwendungsprogramm übertragen. Assoziatoren dienen der Identitifizierung von Aktionen (*callbacks*) im Anwendungsprogramm (vgl. Abschnitt 5.4.2). Anwendungen der vorgestellten Mechanismen sind in Kapitel 6 zu finden.

5.3 Die OPEN LOOK Benutzerschnittstelle

Die OPEN LOOK-Benutzerschnittstelle entspricht den von Sun Microsystems und AT&T entwickelten Stilrichtlinien (*style guides*) [Sun Microsystems 90b; Sun Microsystems 90a] für die Gestaltung einer über verschiedene Hardwareplattformen, Betriebssysteme und Anwendungen hinweg konsistenten, effizienten und mit Maus und Tastatur einfach zu bedienenden graphischen Benutzerschnittstelle. Ziel einer solchen Spezifikation ist einerseits die vollständige Beschreibung des graphischen Erscheinungsbildes von Typen von Objekten (*look*), die ein Anwender auf dem Bildschirm sieht. Andererseits beschreibt eine Spezifikation auch das Verhalten des Systems (*feel*) auf Interaktionen des Anwenders, d.h. wie dieser mit den Objekten arbeiten kann.

Der Unterschied zu anderen Benutzerschnittstellen, z.B. OSF/Motif [Open Software Foundation 90], NeXTStep [NeXT 91] Microsoft Windows [Microsoft 92] oder dem Finder [Apple Computer 92] liegt im äußeren Erscheinungsbild und der Funktionalität der Fenster und Interaktionselemente [Hüskes, Shahrbabaki 93; Ammon 94].

Die OPEN LOOK-Schnittstellenkomponenten (*widgets*) lassen sich in drei Klassen unterteilen, den Bildschirmhintergrund (*framebuffer*), die Interaktionselemente (*controls*) und die Fenster (*windows*). In den nachfolgenden Abschnitten werden einige für die Visualisierung verwendete Klassen mit ihren Elementen genauer beschrieben.

5.3.1 Bildschirmhintergrund

Der Bildschirmhintergrund stellt den Untergrund dar, auf dem die Anwendungen gestartet werden. Die Anwendungen erscheinen auf dem Bildschirmhintergrund entweder als Fenster oder als Piktogramme (*icons*). Piktogramme symbolisieren gestartete Anwendungen, die z.Zt. nicht benötigt werden. Ihre Verwendung fördert die Überschaubarkeit des Bildschirmhintergrundes.

5.3.2 Interaktionselemente

Interaktionselemente sind vom Anwender manipulierbare Schnittstellenkomponenten. Durch die Manipulation, z.B. Drücken eines Knopfes oder Eingeben eines Textes, gibt der Anwender Eingaben an das System. Eine Übersicht über die wichtigsten Interaktionselemente der OPEN LOOK-Benutzerschnittstelle ist in Abbildung 5.2 zu finden, die dazugehörige Funktionalität wird in der nachfolgenden Aufstellung beschrieben.

Knöpfe: In der Darstellung von Knöpfen wird zwischen normalen Knöpfen (*buttons*) und Abkürzungsknöpfen (*abbreviated buttons*) unterschieden. Beide Arten lassen sich weiter unterteilen. Mit einfachen Knöpfen werden Aktionen (*actions*) aufgerufen und Menüknöpfe (*menu buttons*) dienen dem Anzeigen von Menüs. Sie unterscheiden sich von einfachen Knöpfen durch ein kleines Dreieck im Knopf, das die Richtung angibt, in der das Menü angezeigt wird. Schließlich gibt es noch Fensterknöpfe (*window buttons*). Sie dienen dem Aufruf von Aufklappfenstern.

Menüs: Menüs fassen mehrere ähnliche Aktionen zusammen. Es gibt drei Arten von Menüs. Am häufigsten werden Knopfmenüs (*button menus*) benutzt. Des weiteren gibt es noch Aufklapp- (*popup menus*) und Submenüs (*submenus*). Die Einträge in Menüs (*menu items*) dienen, genau wie einfache Knöpfe, dem Aufruf von Aktionen. Alle Menüs können mit einer Stecknadel (*push pin*) auf dem Bildschirmhintergrund angepinnt werden.

Auswahlfelder: Bei den Auswahlfeldern (*settings*) wird zwischen zwei Arten unterschieden. Aus Exklusivauswahlfeldern (*exclusive settings*) kann nur ein Eintrag gleichzeitig ausgewählt werden und aus Mehrfachauswahlfeldern (*nonexclusive settings*) können mehrere Einträge gleichzeitig ausgewählt werden.

Ja/Nein-Felder: Die Ja/Nein-Felder (*check boxes*) können nur zwei mögliche Werte annehmen, *ja* oder *nein* bzw. *wahr* oder *falsch*. Der Wert *wahr* wird durch einen Haken im Feld repräsentiert.

Eingabefelder: Eingabefelder (*input fields*) dienen der Dateneingabe. Dabei wird zwischen numerischen (*numeric fields*) und alphanumerischen (*text fields*) Eingabefeldern unterschieden. Numerische Eingabefelder haben optional zwei Abkürzungsknöpfe am rechten Ende des Feldes, mit denen der angezeigte Wert schrittweise in- bzw. dekrementiert werden kann.

Rollbalken: Ein Rollbalken (*scrollbar*) erlaubt es, den momentan sichtbaren Ausschnitt von Daten in einem speziellen Bereich (*pane*) zu verschieben. Am Rollbalken existieren dafür eine Vielzahl von Angriffspunkten. Der sichtbare Ausschnitt kann z.B. zeilen- oder seitenweise verschoben werden.

Festtexte: Festtexte (*labels*) sind nicht-veränderbare Informationstexte.

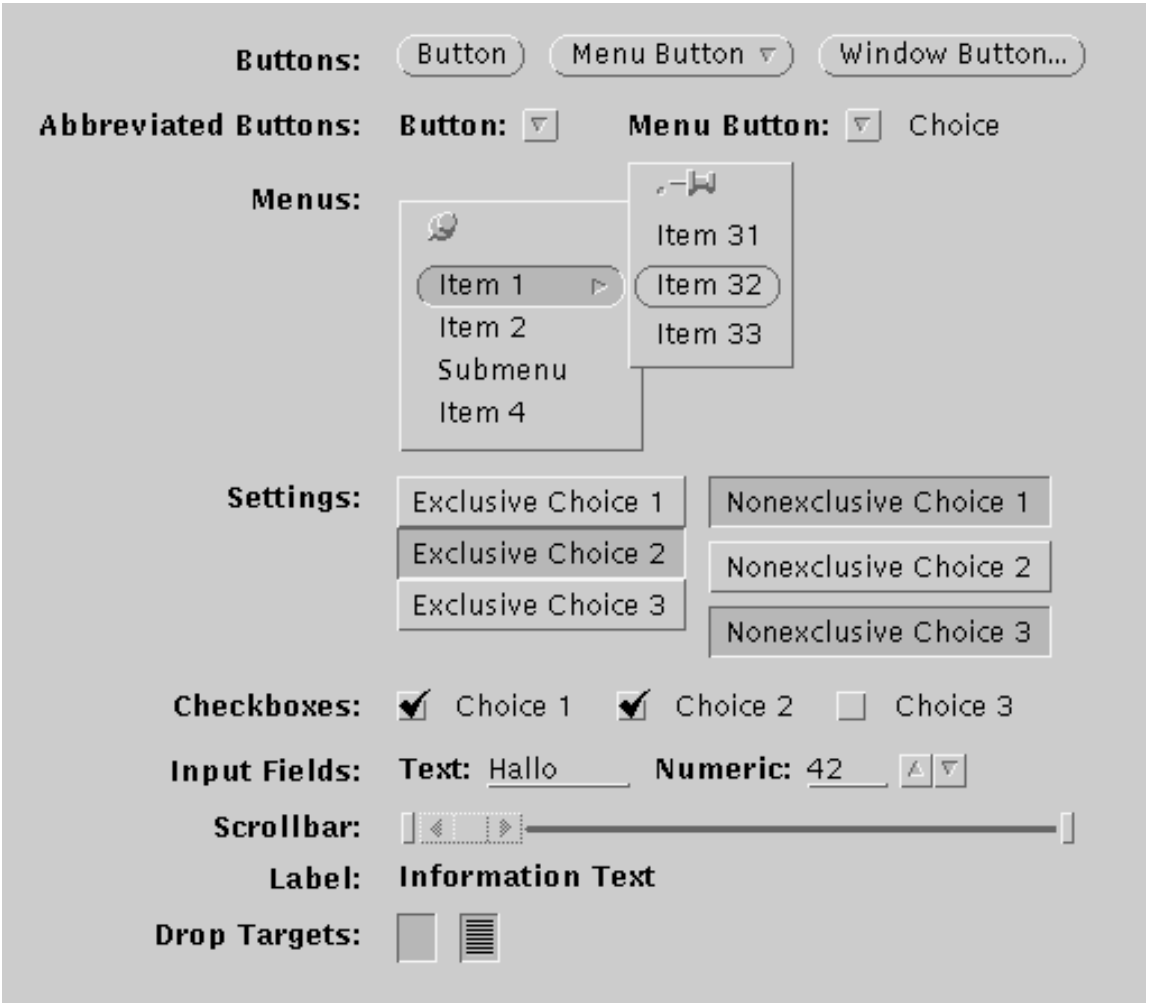


Abbildung 5.2: Wichtige OPEN LOOK Interaktionselemente

Datenaustauschfelder: Datenaustauschfelder (*drop targets*) dienen dem Austausch von Daten über Anwendungsgrenzen hinweg. Ein ausgefülltes Feld repräsentiert austauschfähige Daten in einer Anwendung.

Eine über diese Einführung hinausgehende Beschreibung der vorgestellten sowie eine Beschreibung weiterer Interaktionselemente ist in [Sun Microsystems 90b] zu finden.

5.3.3 Fenster und Fensterelemente

In verteilten Fenstersystemen geben Anwendungsprogramme ihre Informationen auf virtuellen Bildschirmen, sogenannten Fenstern aus. Jedes Anwendungsprogramm hat ein Grundfenster (*base window*). In Abbildung 5.3 ist das Grundfenster des OpenWindows-Dateimanagers (*file manager*) abgebildet. Wird das Fenster geschlossen, so wird es als Piktogramm auf dem Bildschirmhintergrund dargestellt. Weitere Fenster einer Anwendung sind Aufklappfenster (*popup windows*). Sie werden nur auf Anforderung des Anwenders oder der Anwendung geöffnet und verschwinden wieder, sobald die entsprechende Funktion ausgeführt ist.

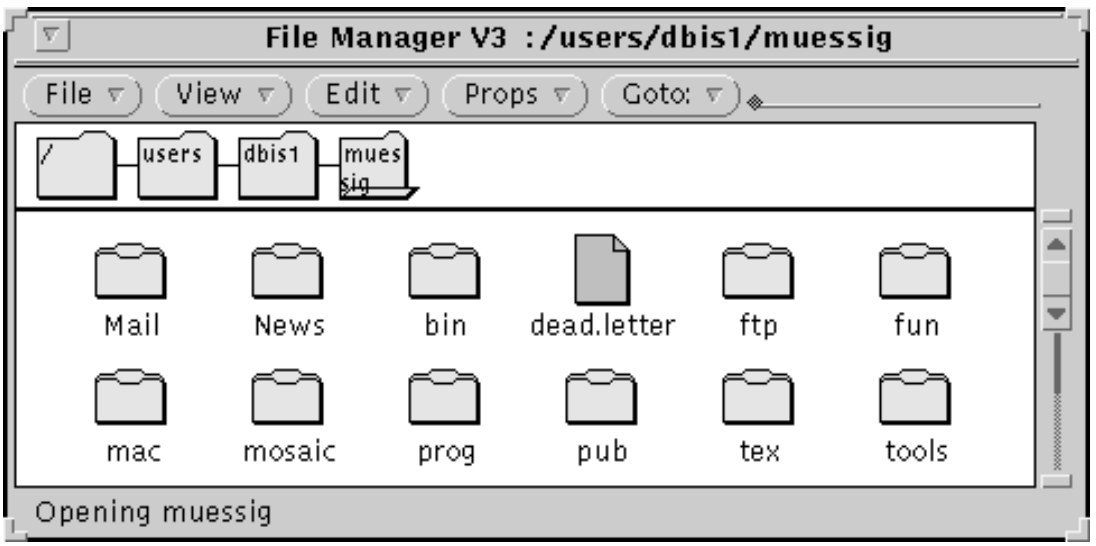


Abbildung 5.3: Grundfenster (OpenWindows Dateimanager)

Jedes Fenster hat eine Kopfzeile, die neben dem Titel des Fensters und dem Namen des bearbeiteten Objekts noch weitere, von der Art des Fensters abhängige, Komponenten enthält (vgl. Abbildung 5.4). Grundfenster enthalten z.B. einen Menüknopf (*window menu button*), über den ein Menü aufrufbar ist. In der Kopfzeile von Aufklappfenstern ist hingegen eine Stecknadel zu finden. Sie dient dem permanenten Anpinnen von Aufklappfenstern auf dem Bildschirmhintergrund. Wird die Stecknadel durch ein weiteres Anklicken herausgezogen, so verschwindet das Fenster wieder vom Bildschirmhintergrund.

Abbildung 5.4: Merkmale von Fenstern

Innerhalb der Fenster befindet sich in der Regel ein Kontrollbereich (*control area*) und ein eingerahmter Bereich (*pane*). Der Kontrollbereich dient zur Aufnahme von Interaktionselementen (vgl. Abschnitt 5.3.2). In dem eingerahmten Bereich werden die Anwendungsinformationen wie Texte oder Graphiken dargestellt. Hat der Bereich vertikale und horizontale Rollbalken, so kann der sichtbare Bildausschnitt verschoben werden. Dadurch ist es möglich, Bilder darzustellen, deren Ausmaße größer sind als der sichtbare Bildausschnitt.

Viele weitere Elemente der Fenster sind optional. Ein Fenster kann spezielle Ecken haben, mit denen seine Größe verändert werden kann (*resize corners*) oder eine Fußzeile (*footer*), in der Fehler oder Mitteilungen für den Anwender ausgegeben werden können.

5.3.4 Rückmeldungen

Neben den in den vorherigen Abschnitten beschriebenen Schnittstellenkomponenten wird in den OPEN LOOK-Stilrichtlinien auch die Form der graphischen Rückmeldung (*feedback*) über den aktuellen Zustand einer Anwendung festgelegt.

Zum Beispiel werden Voreinstellungen (*defaults*) in Menüs und Auswahlfeldern besonders gekennzeichnet. Ausgewählte Elemente werden hervorgehoben (*highlighted*) und nicht anwählbare Elemente werden als nicht aktiv (*inactive*) gekennzeichnet. Kann eine Anwendung momentan keine Eingaben entgegennehmen, so wird sie als belegt (*busy*) gekennzeichnet. Auch der Mauszeiger nimmt in Abhängigkeit vom Bearbeitungszustand einer Anwendung und von seiner Position spezielle Formen an.

5.4 Das NeWS Toolkit

Das NeWS Toolkit [SunSoft 92b; SunSoft 92a] ist ein auf der Sprache PostScript basierendes, objektorientiertes Werkzeug zur Erstellung von Benutzerschnittstellen für das Fenstersystem NeWS. Das Werkzeug implementiert viele der OPEN LOOK-Schnittstellenkomponenten. C-basierte Bibliotheken zur Kommunikation zwischen Anwendungsprogramm und Fenstersystem sowie zur Ereignisverwaltung ermöglichen den Einsatz des Werkzeuges unter Verwendung verschiedener Programmiersprachen. Zum NeWS Toolkit gehört auch ein einfacher Texteditor (*jot*). Auch hierfür steht eine C-Bibliothek zur Verfügung.

5.4.1 Organisation

Der Einsatz eines objektorientierten Werkzeuges zur Erstellung von Benutzerschnittstellen hat folgende Vorteile. Die grundlegenden Ziele der objektorientierten Programmierung, Vererbung (*inheritance*), Wiederverwendung (*polymorphism*) und Kapselung (*encapsulation*), decken sich mit den Ansprüchen von Werkzeugen zur Benutzerschnittstellenerstellung (vgl. Abschnitt 2.6).

Zur Erstellung von OPEN LOOK-konformen Benutzerschnittstellen sind im NeWS Toolkit über 30 Klassen vordefiniert. In Abbildung 5.5 sind alle Klassen des Werkzeuges in ihrer Hierarchie dargestellt. Von diesen Klassen können je nach Anwendung entweder Subklassen oder Instanzen erzeugt werden.

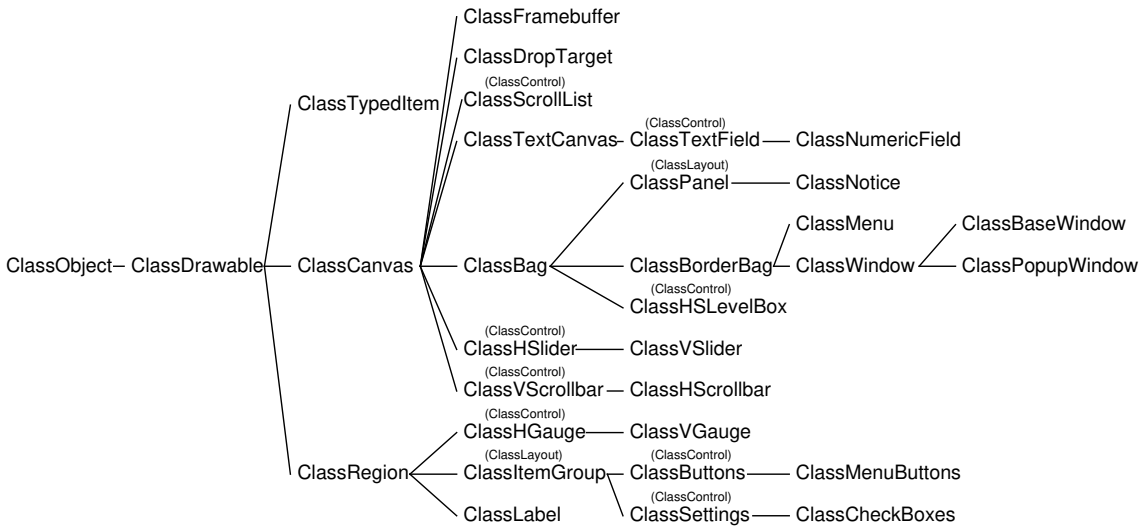


Abbildung 5.5: Hierarchie vordefinierter Klassen im NeWS Toolkit [SunSoft 92b]

Die folgende alphabetische Aufzählung stellt die wichtigsten im NeWS Toolkit definierten Klassen vor und beschreibt kurz ihren Verwendungszweck.

ClassObject: Die Klasse *ClassObject* ist die Wurzel der Klassenhierarchie. Sie beinhaltet die Basismethoden des NeWS (vgl. Abschnitt 5.2.1).

ClassCanvas: Die Klasse *ClassCanvas* stellt die Grundfunktionalität für ihre Subklassen zur Verfügung. Diese Funktionalität besteht z.B. darin, daß sie sich selbst auf dem Bildschirm anzeigen und restaurieren sowie Operationen wie Größenveränderungen selbst ausführen können. Das Aussehen eines Canvases ist durch eine nicht notwendigerweise rechteckige Fläche gekennzeichnet. Canvases beinhalten auch Fähigkeiten zur Ereignisbehandlung.

ClassBag: Ein *Bag* ist eine Art Behälter. Er wird verwendet, um mehrere Klienten, Canvases oder Regions, für Layoutzwecke oder Operationen zusammenzufassen.

ClassBorderBag: Ein *BorderBag* kann bis zu fünf Klienten an fest vordefinierten Positionen verwalten. Die Klienten können sich in der Mitte und an den vier Rändern befinden. Diese Positionen sind z.B. für den Aufbau von Fenstern sinnvoll.

ClassPanel: Ein *Panel* ist wie ein Bag eine Art Behälter. Im Gegensatz zum Bag stehen für ein Panel mehrere Layoutstrategien zur Verfügung. Durch die Layoutstrategien wird die Anordnung der Klienten im Panel bestimmt.

ClassItemGroup: Die Klasse *ClassItemGroup* stellt Methoden zur Verwaltung von mehreren gleichen Klienten zur Verfügung. Sie hat den Zweck, die Anzahl von Klienten, die in Behälter eingefügt werden, zu reduzieren. Genau wie für Panels gibt es mehrere Layoutstrategien.

ClassControl: Die Klasse *ClassControl* wird per Mehrfachvererbung an alle Klassen vererbt, die vom Benutzer manipulierbare Schnittstellenkomponenten definieren. Sie beinhaltet u.a. Methoden zur Definition von Benachrichtigungsprozeduren (*notify procedures*).

ClassLayout: Die Klasse *ClassLayout* wird per Mehrfachvererbung an alle Klassen vererbt, die Klienten mit Layoutstrategien plazieren. Es gibt vier Layoutstrategien. Beim *spaced placement* sind keine weiteren Layoutinformationen notwendig. Die Klienten werden automatisch so plaziert, daß der minimalste Platz belegt wird. Beim *grid placement* werden die Klienten zeilen- bzw. spaltenweise plaziert. Beim *absolute placement* werden die Klienten durch die zusätzliche Angabe von absoluten Koordinaten plaziert und beim *calculated placement* wird die Plazierung der Klienten durch Ausführung eines Kodefragmentes berechnet.

Aufbauend auf diesen vom Aussehen der Benutzerschnittstelle unabhängigen Klassen gibt es im NeWS Toolkit eine Vielzahl von Klassen, die die meisten der OPEN LOOK-Schnittstellenkomponenten implementieren. Stellvertretend für alle anderen Klassen werden hier die Klassen *ClassBaseWindow* und *ClassMenuButtons* aufgeführt.

ClassBaseWindow: Instanzen dieser Klasse sind OPEN LOOK-Grundfenster (vgl. Abschnitt 5.3.3).

ClassMenuButtons: Instanzen dieser Klasse sind OPEN LOOK-Menüknöpfe (vgl. Abschnitt 5.3.2). Eine Instanz stellt dabei eine Menge von Menüknöpfen zur Verfügung, deren Anordnung durch die gewählte Layoutstrategie bestimmt wird.

Eine genaue Beschreibung aller vom NeWS Toolkit definierten Klassen ist in [SunSoft 92b] zu finden.

5.4.2 Kommunikation und Ereignisverwaltung

Bestandteil des NeWS Toolkits ist der Wire Service, eine C-Bibliothek, die eine bidirektionale Kommunikation zwischen Anwendungsprogrammen und dem X11/NeWS Server ermöglicht. In der Bibliothek sind Funktionen zum Auf- und Abbau von Verbindungen zum X11/NeWS Server und Funktionen zum Assoziieren von Schnittstellenkomponenten und Ereignissen auf der NeWS-Seite mit Namen und Aktionen auf der Seite des Anwendungsprogramms zu finden. Darüber hinaus sind in der Bibliothek auch Funktionen zur Abhandlung von synchroner und asynchroner Kommunikation zu finden. Einzelne wichtige Funktionen werden in Abschnitt 6.1.1 noch ausführlich beschrieben.

In der Kommunikation zwischen einem Anwendungsprogramm und dem X11/NeWS Server wird zwischen drei Kommunikationsarten unterschieden. Im einfachsten Fall sendet nur das Anwendungsprogramm NeWS-Kodefragmente zum X11/NeWS Server und erwartet keine Rückgabewerte. Diese Art der Kommunikation wird z.B. verwendet, um Eigenschaften von Schnittstellenkomponenten zu verändern.

Bei der *synchronen* Kommunikation sendet das Anwendungsprogramm NeWS-Kodefragmente zum Server und wartet auf die Rückgabewerte. Dabei wird die Programmausführung so lange unterbrochen (*blocked*), bis die Rückgabewerte vom Server angekommen sind. Die synchrone Kommunikation wird z.B. verwendet, um Eigenschaften von Schnittstellenkomponenten abzufragen und die Ergebnisse dann im Anwendungsprogramm weiter zu verwenden.

Auch bei der *asynchronen* Kommunikation sendet das Anwendungsprogramm NeWS-Kodefragmente zum Server, wartet aber im Unterschied zur synchronen Kommunikation nicht auf die Rückgabewerte. Diese Art der Kommunikation wird verwendet, um Funktionen des Anwendungsprogramms mit Interaktionselementen der Benutzerschnittstelle zu assoziieren. Wird dann ein Interaktionselement manipuliert, z.B. ein Knopf gedrückt, so sendet der X11/NeWS Server eine Nachricht an das Anwendungsprogramm. Dieses kann anhand des Assoziators in der Nachricht die dazugehörige Aktion identifizieren und aufrufen. Die aufgerufene Funktion ist für die Behandlung evtl. vorhandener Rückgabewerte selbst zuständig.

5.4.3 Textbearbeitung

Mit dem NeWS Toolkit wird neben dem Wire Service noch ein weiteres Paket geliefert, das Jot-Textpaket. Es stellt Datenstrukturen und Funktionen zur interaktiven Behandlung von Texten zur Verfügung. Das Paket besteht im wesentlichen aus den drei Teilen JotText, JotView und Jot. In diesem Abschnitt werden nur die Grundkonzepte des Paketes beschrieben, weitere Informationen zum Jot-Textpaket sind in [SunSoft 92b] zu finden.

Die Funktionalität des Textpakets korrespondiert mit dem von Smalltalk bekannten *model-view-controller* Paradigma [Goldberg, Robson 83; Hüskes 93]. Das Paradigma unterteilt eine Anwendung in die Komponenten Datenobjekt (*model*), welches die Information der Anwendung enthält, die Sicht (*view*), die das mit ihr assoziierte Datenobjekt graphisch

repräsentiert und den Verwalter (*controller*), der als Schnittstelle zwischen dem Eingabegerät und dem Datenobjekt fungiert. Er ermöglicht dem Benutzer die Interaktion mit dem Datenobjekt über die Sicht.

Ein JotText entspricht dem Datenobjekt, das die Information der Anwendung, i.d.R. einen mehrzeiligen Text, enthält. Der JotText selbst ist nicht sichtbar. Er benötigt zu seiner Darstellung einen JotView. Der JotView repräsentiert die Daten des JotTextes graphisch und ermöglicht eine bestimmte Sicht auf sie. Da es sinnvoll sein kann, verschiedene Sichten auf ein Datenobjekt zu haben, sieht NeWS auch die Möglichkeit der Erzeugung mehrerer JotViews für denselben JotText vor.

Zusätzlich ist es möglich, kleine Regionen (*spans*) auf dem JotText zu definieren. Eine Region wird durch ihre Ursprungsposition und Länge beschrieben. Der Positionsparameter paßt sich automatisch an, wenn Zeichen vor der Region eingefügt oder gelöscht werden. Ebenso verändert sich der Längenparameter der Region automatisch, wenn innerhalb ihrer Begrenzungen Zeichen hinzugefügt oder entfernt werden. Jeder JotText verwaltet eine Liste, der mit ihm verknüpften Regionen.

5.4.4 Datenaustausch

Das NeWS Toolkit unterstüzt den Datenaustausch auf der Ebene der Benutzerschnittstelle. Einfache Daten, wie z.B. Zeichenketten und Zahlen, können selektiert und direkt mit der Maus aufgenommen (*drag*) werden. Aufgenommene Daten können über den Bildschirm bewegt und über einem anderen Feld fallengelassen (*drop*) werden. Die bewegten Daten werden dann in dieses Feld eingefügt. Komplexere Daten, wie z.B. längere Texte oder Bilder, können meist nicht direkt mit der Maus selektiert werden. Der symbolische Austausch solcher Daten ist über Datenaustauschfelder (vgl. Abbildung 5.2) möglich. Datenaustauschfelder sind Instanzen der Klasse *ClassDropTarget*.

Dabei ist das Anwendungsprogramm für den eigentlichen Datenaustausch selbst verantwortlich. Das NeWS Toolkit benachrichtigt das Programm nur über den Start, das Bewegen und das Ziel des Austauschs. Benutzerdefinierte Klassen müssen demnach bestimmte Methoden zur Verfügung stellen, die auf Ereignisse, wie Aufnehmen und Fallenlassen, reagieren können.

5.4.5 Anwendungsentwicklung

Die Anwendungsentwicklung mit dem NeWS Toolkit besteht aus zwei Teilen in unterschiedlichen Programmiersprachen. Der Server-Teil wird in der objektorientierten NeWS-Sprache geschrieben. Der Klienten-Teil, das Anwendungsprogramm, kann z.B. in C oder C++ geschrieben werden. Er enthält die NeWS-Aufrufe. Im Gegensatz zu X11-Anwendungen, können Teile der NeWS-Anwendungen auch im Server ablaufen. Die Erstellung des Server-Teils besteht im allgemeinen aus vier Schritten:

1. Definition der vom Anwendungsprogramm benutzen Subklassen.
2. Instanziierung der im ersten Schritt erzeugten Subklassen.
3. Zusammensetzen der instanziierten Objekte. Dazu werden die Objekte meistens in Behältern gruppiert. Die Behälter werden anschließend in Fenster eingefügt.
4. Festlegen der Größe von Fenstern und Plazieren der Fenster, Aktivieren der Ereignisverwaltung und schließlich Anzeigen der Fensters.

Im Klienten-Teil werden fünf Schritte unterschieden:

1. Aufbauen einer Verbindung mit dem X11/NeWS Server.
2. Definition von Aktionen (*callback functions*), die durch Nachrichten (*notifier*) von den Interaktionselementen angestoßen werden. Vielen der OPEN LOOK-Schnittstellenkomponenten können Benachrichtigungsprozeduren zugewiesen werden, die Nachrichten an das Anwendungsprogramm senden, wenn sie manipuliert werden.
3. Definition von Assoziatoren (*tags*) und Identifikatoren (*tokens*). Assoziatoren dienen der Assoziierung von Interaktionselementen mit Aktionen, während Identifikatoren Namen für NeWS-Objekte sind.
4. Initialisierung, Ausführen des Server-Teils.
5. Warten auf Nachrichten vom X11/NeWS Server.

Bei statisch übersetzten C-Programmen sind die hier aufgeführten Schritte der Anwendungsentwicklung in dieser Form zu finden. Für andere, dynamische Programmiersprachen, wie z.B. TL, können sich sowohl die Reihenfolge der Schritte ändern, als auch Schritte des Server-Teils im Anwendungsprogramm durchgeführt werden.

5.5 Die OpenWindows Anwendungsumgebung

OpenWindows ist genau wie NeWS ein verteiltes Fenstersystem. Die wesentlichen Komponenten der OpenWindows-Anwendungsumgebung sind der X11/NeWS Server und Werkzeuge zur Erstellung von OPEN LOOK-konformen Benutzerschnittstellen, sowohl für das X11- als auch für das NeWS-Protokoll. Daneben existieren diverse fensterbasierte Hilfsmittel (*deskset tools*) [Davison et al. 92; SunSoft 93a; SunSoft 93b], z.B. zum Editieren von Texten (*textedit*), zum Schreiben und Versenden von Briefen (*mailtool*) und zur Dateibehandlung (*file manager*, vgl. Abbildung 5.3), auf die in diesem Zusammenhang aber nicht weiter eingegangen werden soll.

5.5.1 Der X11/NeWS Server

Der X11/NeWS Server [Schaufler 89; Gosling et al. 89; Sun Microsystems 90d] bildet die Grundlage für die OpenWindows-Anwendungsumgebung. Der X11/NeWS Server ist ein Fenstersystemmanager, der das *Client/Server-Modell* [Tanenbaum 87; Sun Microsystems 90c] implementiert, d.h. es existiert eine klare Trennlinie zwischen Anwendungsprogramm (Klient) und Diensterbringer (Server). Klient und der Server können sich dabei auch auf unterschiedlichen Knoten eines Netzwerkes befinden.

Bildschirm, Tastatur und Maus werden von einem Prozeß, dem Server, verwaltet. Die drei Hauptaufgaben des Servers sind, wie in Abbildung 5.6 zu sehen ist, die Interpretation und die Ausführung von NeWS-Programmen, das Zuweisen von Bildschirmteilen und das Verteilen von Eingaben an die Anwendungsprogramme. Der Server verbindet die Semantik des X11- und des NeWS-Protokolls, so daß Anwendungsprogramme über beide Protokolle mit dem Server kommunizieren können.

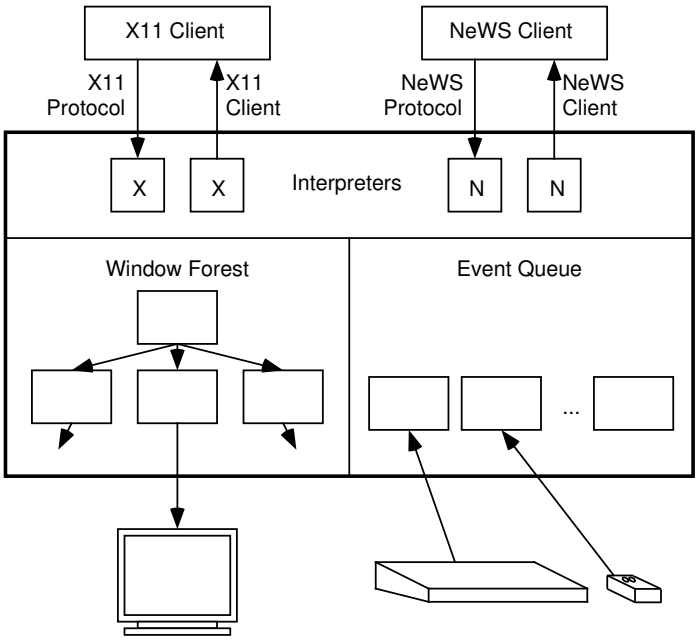


Abbildung 5.6: Struktur des X11/NeWS Server [Gosling et al. 89]

Der X11/NeWS Server besteht, wie in Abbildung 5.6 zu sehen ist, aus mehreren Komponenten, die im folgenden kurz beschrieben werden. Der Server enthält die schon in Abschnitt 5.2.1 beschriebenen leichten Prozesse. Der Scheduler verteilt die für den Server zur Verfügung stehende Rechenzeit auf diese Prozesse. Jeder Prozeß kann unterschiedliche Operationen ausführen und Nachrichten der Eingabegeräte und anderer Prozesse empfangen.

Der Fensterwald (*window forest*) ist eine Datenstruktur, in der alle Fenster eines Bildschirms verwaltet werden. Dabei ist es egal, ob ein Fenster durch das X11- oder durch das NeWS-Protokoll erzeugt worden ist.

Die Ereigniswarteschlange (*event queue*) hat die Aufgabe, alle von den Eingabegeräten ankommenden Nachrichten zu sammeln, mit einem Zeitstempel zu versehen und zu serialisieren. Die Verteilung wird anhand von Beispielnachrichten (*interests*) durchgeführt, die von den leichten Prozessen generiert werden. Sie werden als Muster für den Vergleich herangezogen. Jeder leichte Prozeß, dessen Beispielnachricht einer ankommenden Nachricht gleicht, bekommt eine Kopie der Nachricht. Er leitet sie gegebenenfalls an das Anwendungsprogramm weiter, mit dem er kommuniziert.

5.5.2 Der Developer's Guide

Mit Hilfe des Devguide (*Developer's Graphical User Interface Design Editor*, vgl. Abbildung 5.7) [Sun Microsystems 92] ist es möglich, interaktiv OPEN LOOK-konforme Benutzerschnittstellen für die OpenWindows-Anwendungsumgebung zu erstellen. Das Ergebnis wird in einer GIL-Datei (*Graphics Interface Language*) gespeichert. Diese Datei kann in C-Quelltextdateien sowie die dazugehörige *Make*-Datei umgewandelt werden. Anschließend kann mit einem C-Übersetzter ein unter OpenWindows lauffähiges Programm erzeugt werden (vgl. Abschnitt 2.4).

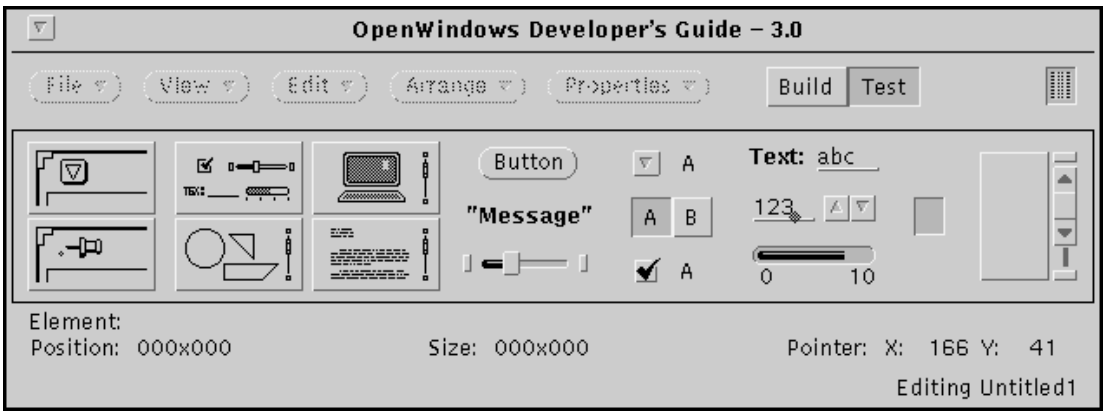


Abbildung 5.7: OpenWindows Developer's Guide

Schon während der Entwicklung mit dem Devguide kann das endgültige Aussehen und Verhalten der Benutzerschnittstelle in einem *Test-Modus* simuliert werden. Umfangreiche Dialogmanagementfunktionen (*connections*) ermöglichen die Ansteuerung vieler Schnittstellenkomponenten auf der Ebene der Benutzerschnittstelle, ohne daß dieses Verhalten explizit in C kodiert werden muß.

5.5.3 Zusammenspiel der kommerziellen Dienste

In Abbildung 5.8 ist das Zusammenspiel der in diesem Kapitel vorgestellten kommerziellen Dienste dargestellt. Alle Dienste sind unter dem Betriebssystem SunOS in der OpenWindows-Anwendungsumgebung lauffähig.

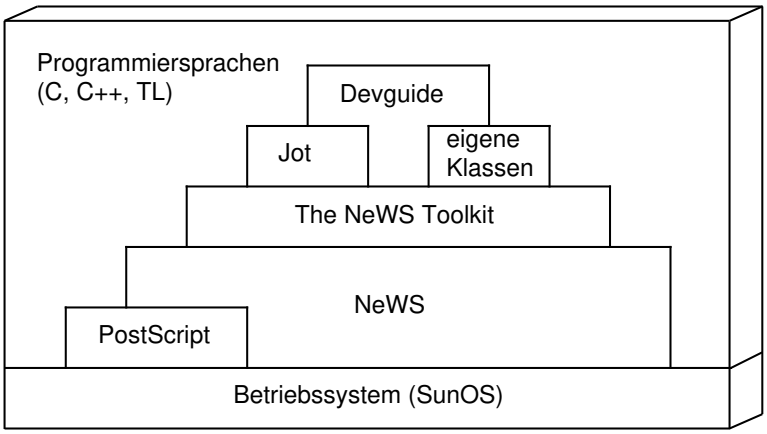


Abbildung 5.8: Zusammenspiel der kommerziellen Dienste

Für einen Anwendungsprogrammierer gibt es verschiedene Möglichkeiten, mit den vorgestellten Diensten Anwendungsprogramme zu erzeugen. Zum Beispiel ist es denkbar, ein Programm nur in PostScript (vgl. Abschnitt 5.1) zu schreiben, da die Sprache neben graphikspezifischen Operatoren auch eine Reihe allgemeiner Operatoren enthält und damit zu einer vielseitig verwendbaren allgemeinen Programmiersprache wird. Da auf dieser Ebene jedoch keine Schnittstelle zur Programmierung graphischer Benutzerschnittstellen verfügbar ist, müßte diese erst implementiert werden. Auch stehen für die Programmierung in PostScript keine allgemein verwendbaren Bibliotheken zur Verfügung. Ein weiterer Nachteil ist die erst zur Laufzeit erfolgende Typüberprüfung. Aus den genannten Gründen ist von einer vollen Implementation von größeren Anwendungen in PostScript abzuraten.

Als nächste Ebene bietet sich die Implementation in NeWS (vgl. Abschnitt 5.2) an. Da NeWS weitgehend auf PostScript basiert, hat diese Sprache auch ähnliche Nachteile wie PostScript. In der Sprache sind jedoch viele Erweiterungen zu finden, die der Interaktion (Bildschirmausgaben sowie Maus- und Tastatureingaben) und der Mehrprozeßfähigkeit in einem Fenstersystem Rechnung tragen. Dadurch sind die Grundlagen geschaffen, Benutzerschnittstellen effizient zu programmieren. In NeWS findet sich auch ein Bibliothekskonzept, das die Wiederverwendung unterstützt. Genau wie auf der Ebene von PostScript ist es auf der Ebene von NeWS nicht anzuraten, komplette Anwendungen zu schreiben. Zum einen fehlt auch hier die Unterstützung durch allgemeine Bibliotheken und zum anderen wird die Entwicklung und Wartung von Programmen auf dieser Ebene nur ungenügend unterstützt.

Trotzdem kann es aus Effizienzgründen sinnvoll sein, kleinere Unterprogramme direkt im X11/NeWS Server und nicht im Klient ablaufen zu lassen. Vorstellbar sind z.B. Prozeduren zur Manipulation von Schnittstellenkomponenten.

Auf dem Fenstersystem NeWS setzt mit dem NeWS Toolkit ein objektorientiertes Werkzeug auf (vgl. Abschnitt 5.4). Das Werkzeug implementiert eine Klassenhierarchie zur effizienten Programmierung graphischer Benutzerschnittstellen. Die Qualität der zugrundeliegenden Sprache wird jedoch dadurch nicht gesteigert. Aus diesem Grund gilt auch hier das eben für NeWS Gesagte.

Einerseits ist die fehlerträchtige direkte NeWS-Programmierung zu unsicher, andererseits ist die durch das NeWS Toolkit zur Verfügung gestellte mächtige Funktionalität unbestritten. Daher bietet es sich an, auf dieser Ebene eine Schnittstelle zu einer Hochsprache zu implementieren. Eine solche Schnittstelle hat folgende Vorteile für die Erstellung von Benutzerschnittstellen. Funktionsaufrufe in einer Hochsprache sind parametrisierbar und typisierbar. Sie ermöglichen weiterhin, von der kellerspeicherorientierten Syntax der NeWS-Methodenaufrufe zu abstrahieren und die sonst aufwendig zu programmierende Kommunikation, die zwischen einer Anwendung und dem X11/NeWS Server stattfindet, zu verbergen.

Eine Anbindung des Werkzeuges an einen Übersetzter für C oder C++ erlaubt die Ausnutzung der eben aufgezählten Vorteile. Es treten jedoch auch die typischen Nachteile von Übersetzern auf. Zum Beispiel ist es nicht möglich, in einem Übersetzer interaktiv Funktionen oder Programme auszuführen, da ein Programm zur Laufzeit komplett übersetzt und gebunden vorliegen muß. In der Entwicklungs- und Testphase ist es jedoch von Vorteil, auch dynamisch Funktionen ausführen und testen zu können. Dadurch ist es dann möglich, Benutzerschnittstellen oder Teile davon zu erzeugen und zu verändern (vgl. Abschnitt 6.1). Auch die anderen in diesem Kapitel vorgestellten Dienste der OpenWindows-Anwendungsumgebung, z.B. der Devguide und das Jot-Textpaket, lassen sich über eine Schnittstelle von einer Hochsprache aus nutzen. In Kapitel 6 wird die Implementation einer solchen Schnittstelle zwischen der Tycoon-Entwicklungsumgebung und dem NeWS Toolkit ausführlich vorgestellt.

6. Graphische Benutzerschnittstellen in Tycoon

In diesem und dem nächsten Kapitel werden die beiden im Rahmen dieser Arbeit implementierten Bibliotheken (*newsenv* und *editenv*) vorgestellt. Zum besseren Verständnis dieser Kapitel sind u.a. Kenntnisse in der NeWS- und der Tycoon-Programmierung sowie dem Aufbau des NeWS Toolkits erforderlich. Die vorherigen Kapitel sollten grundlegende Kenntnisse vermittelt haben.

Dieses Kapitel stellt ausführlich die Konzepte und die Implementation des *newsenv*, einer Bibliothek zur dynamischen Erzeugung und Manipulation graphischer Benutzerschnittstellen unter OPEN LOOK in Tycoon, vor. Die Bibliothek baut auf den Ergebnissen der Untersuchung in Kapitel 2 auf. Dort wurde untersucht, welcher Ansatz am besten geeignet ist, aus einer Programmiersprache heraus dynamisch graphische Benutzerschnittstellen zu erzeugen. Aus diesem Grund werden in der Bibliothek die Klassen des in Abschnitt 5.4 beschriebenen NeWS Toolkits in Tycoon verfügbar gemacht.

Die umfangreiche Bibliothek (ca. 40 Module) bietet Schnittstellen zum NeWS Toolkit, zum Wire Service und zum Jot. Sie ist unter anderem eine Voraussetzung für die Implementation der in Kapitel 7 vorgestellten typsicheren generischen Editoren. Im Anschluß an die Vorstellung der Bibliothek werden einige praktische Anwendungsgebiete des *newsenv* aufgezeigt. Ein detaillierter Überblick über die Funktionalität der Bibliotheksmodule ist auch in [Mathiske et al. 93] zu finden.

6.1 Dynamische Erzeugung graphischer Benutzerschnittstellen

Das *newsenv* ist das Ergebnis der Neuimplementation des in [Kirch, Müßig 92] vorgestellten Moduls *news*. Im Zuge der Erweiterung dieses Moduls sind die Funktionen auf mehrere Module verteilt worden. Abbildung 6.1 zeigt, wie das *newsenv* in sechs jeweils inhaltlich zusammenhängende Teile gegliedert werden kann. Jeder Teil beinhaltet i.d.R. mehrere Module.

Die Grundlage bildet ein Teil mit Modulen zur Kommunikation und Ereignisverwaltung. Alle weiteren Module der Bibliothek nutzen die von diesen Modulen zur Verfügung gestellte Funktionalität. Der weitaus größte Teil der Module implementiert die Schnittstelle

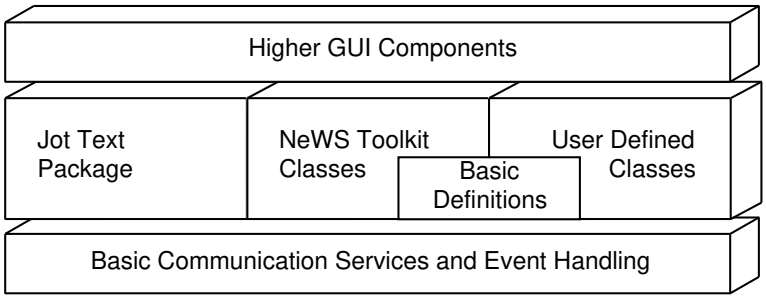


Abbildung 6.1: Grobstruktur des *newsenv*

zu dem in Abschnitt 5.4 vorgestellten NeWS Toolkit. Auf dieser Ebene sind noch weitere Module angesiedelt, die Schnittstellen zu NeWS-Klassendefinitionen von Anwendern bieten. Einen Teil für sich bilden die Jot-Editoren. In der höchsten Schicht des *newsenv* befinden sich funktional speziellere Schnittstellenkomponenten, wie z.B. Dateiauswahlboxen, die auf den Basismodulen der Bibliothek aufbauen. In den nachfolgenden Abschnitten werden die einzelnen Teile ausführlich beschrieben.

Die Benutzung von Funktionen aus der Bibliothek hat Vorteile für die Erstellung von Benutzerschnittstellen. Die Funktionen des *newsenv* gestatten es, eine Typisierung der Eingabeparameter durchzuführen, von der kellerspeicher-orientierten Syntax der NeWS-Methodenaufrufe zu abstrahieren und die sonst aufwendig zu programmierende Kommunikation, die zwischen einer Tycoon-Anwendung und dem X11/NeWS Server stattfindet, zu verbergen (vgl. Abschnitt 5.5.3).

6.1.1 Kommunikation und Ereignisverwaltung

Ein Programm, das Schnittstellenkomponenten auf einem Bildschirm darstellen und über Manipulationen an den Komponenten informiert werden will, muß mit dem X11/NeWS Server kommunizieren. Die Verbindung zwischen dem Tycoon-System und dem X11/NeWS Server wurde in einer älteren Version des Servers über benannte Dateien des Unix-Dateisystems (*named pipes*) abgewickelt. In der aktuellen Version wird die Verbindung durch eine bidirektionale Leitung (*wire*) realisiert. Die Leitung kann mit einer Warteschlange verglichen werden, in der die Wartenden je nach verfügbarer Zeit des jeweiligen Empfängers abgefertigt werden. Optional ist es jedoch möglich, die Abfertigung programmtechnisch zu erzwingen.

In der Tycoon-Umgebung wird die Kommunikation zwischen einem Tycoon-Programm und dem X11/NeWS Server über das Modul *wire* abgewickelt. Darauf aufbauend implementiert das Modul *action* eine dynamische Ereignisverwaltung.

Kommunikation

Das Modul *wire* stellt im wesentlichen eine Schnittstelle zum Wire Service des NeWS Toolkits dar. Es enthält Funktionen zum Auf- und Abbau sowie zur Initialisierung von Verbindungen zwischen der Tycoon-Umgebung und dem X11/NeWS Server. Weitere Funktionen erlauben das dynamische Laden von PostScript-Fragmenten in den Server sowie das Senden und Empfangen von typisierten Werten.

Die Schnittstelle zum Wire Service ist über die dynamische Anbindung von externen C-Bibliotheken realisiert. Die Funktionen der C-Bibliothek *libwire.so.1.0* werden über die generische *bind* Funktion in Tycoon-Modulen bekannt gemacht. Zum Beispiel wird die C-Funktion zum Verbindungsaufbau zwischen Tycoon und dem X11/NeWS Server (*Open*) wie folgt eingeführt:

*let Open = bind(:**Fun**(:String) :Int "/usr/openwin/lib/libwire.so.1.0" "wire_Open" "si")*

Der erste Parameter beschreibt die Signatur (***Fun**(:String) :Int*) der neuen Tycoon-Funktion *Open*. Die nächsten beiden Parameter geben den Namen der C-Bibliothek (*libwire.so.1.0*) und den Namen der C-Funktion (*wire_Open*), die importiert werden soll, an. Der letzte Parameter beschreibt die erwarteten Parameter der C-Funktion. In diesem Fall ist der Eingabeparameter eine Zeichenkette (s) und der Rückgabeparameter eine ganze Zahl (i). Alle in Tycoon bekanntgemachten Funktionen der C-Bibliothek werden in der gleichen Weise angebunden. Eine ausführliche Beschreibung der Anbindung externer C-Funktionen ist in [Mathiske et al. 93] zu finden.

Zwecks Vereinfachung der Benutzung wird beim Import des Moduls *wire* automatisch eine Verbindung zum Server aufgebaut. Dies geschieht durch den internen Aufruf der Funktion *new* mit dem *defaultServer*. Der Variable *defaultServer* wird der Name des Servers zugewiesen, auf dem das Tycoon-System läuft.

new(defaultServer)

Nach einem Verbindungsaufbau werden zuerst die benötigten Pakete, hier *NeWS*, *TNTCore* und *TNT* (vgl. Abschnitt 5.2.1), geladen. Anschließend können benutzerdefinierte Klassendefinitionen in den Server geladen werden.

Für die im nächsten Abschnitt beschriebene Ereignisverwaltung ist die Funktion *allocateTags* wichtig. Sie erlaubt es, Assoziatoren (*tags*) zu alloziieren, anhand derer später aufzurufende Tycoon-Funktionen identifiziert werden (asynchrone Kommunikation). Die Funktion *new* initialisiert außerdem einen speziellen Assoziator, das *scratchTag*, welches zur Abwicklung der synchronen Kommunikation (vgl. Abschnitt 5.4.2) benötigt wird. In diesem Zusammenhang sei noch auf die Funktion *scratchSync* hingewiesen. Die Funktion wird eingesetzt, um die synchrone Kommunikation sicher abzuwickeln (vgl. Abschnitt 6.1.2).

Eine Reihe von Funktionen des Wire Service erlauben das typisierte Schreiben (*write*) in und Lesen (*read*) aus der Leitung. Die von Tycoon aus in die Leitung geschriebenen Objekte werden in FIFO-Reihenfolge (*first in, first out*) auf dem Kellerspeicher des Servers

abgelegt. Dort stehen sie anschließend für weitere Manipulationen bzw. als Parameter für aufgerufene Methoden zur Verfügung. Umgekehrt werden die Objekte vom Kellerspeicher mittels **tagprint** bzw. **typedprint** (vgl. Abschnitt 5.2.2) in die Leitung geschrieben. Sie können dann von Tycoon aus ausgelesen werden.

An einigen Stellen in Tycoon-Programmen, z.B. bei der Installation von Aktionen oder dem Bildschirmaufbau, kann es wünschenswert sein, die Abarbeitung der in die Leitung geschriebenen Operatoren, Methoden und Daten zu erzwingen. Aus diesem Grund steht die *flush*-Funktion des Wire Service auch in Tycoon zur Verfügung.

Ereignisverwaltung

Die Ereignisverwaltung wird durch das Modul *action* implementiert. Aufgabe des Moduls ist zum einen die Installation (Assoziation) von Aktionen an Schnittstellenkomponenten und zum anderen die Behandlung auflaufender asynchroner Nachrichten (*events*).

Vielen OPEN LOOK Schnittstellenkomponenten kann eine Benachrichtigungsprozedur (*notify procedure*) zugewiesen werden. Diese Prozedur wird ausgeführt, wenn die Schnittstellenkomponente manipuliert, z.B. ein Knopf gedrückt wird. Solche Prozeduren können Aktionen (*callbacks*) in Tycoon-Programmen anstoßen. Damit die richtige Aktion angestoßen wird, muß die Benachrichtigungsprozedur der Schnittstellenkomponente mit der anzustoßenden Aktion assoziiert werden. Dies geschieht mit der Funktion *action.new*, die beide Partner über einen Assoziator (*tag*) verbindet.

```
let new(action :Fun() :Ok) :Wire.Tag =
  begin
    let index = wire.allocateTags(1)
    dynArray.set(events index action)
    index
  end
```

Wie in der Funktion zu sehen ist, werden die Aktionen (*action*) mit ihrem Assoziator (*index*) für den Anwender transparent in einem dynamischen Feld (*events*) gehalten. Der Vorteil dieser Datenstruktur gegenüber dem von Tycoon angebotenen Datentyp **Array** ist, daß ein dynamisches Feld zur Laufzeit wachsen kann, während bei dem von Tycoon angebotenen Typ bei der Erzeugung des Feldes die Größe festgelegt werden muß.

Die Benachrichtigungsprozedur besteht auf der NeWS-Seite im einfachsten Fall nur aus folgender Sequenz:

```
{ tag tagprint }
```

Wenn eine Schnittstellenkomponente manipuliert wird, so wird die Prozedur ausgeführt und durch *tag* ***tagprint*** der Assoziator über die Leitung zum Tycoon-Programm gesendet. Ist in der Nachricht ein gültiger Assoziator, so wird die durch den Assoziator identifizierte Aktion

ausgeführt. Eine solche Vorgehensweise bietet eine große Flexibilität, denn es können zur Programmlaufzeit Aktionen auf einfache Weise hinzugefügt und entfernt werden.

Ankommende Ereignisse können neben dem sie identifizierenden Assoziator auch noch weitere Daten beinhalten. Deshalb gibt es für einige Basistypen von Tycoon (*Int*, *Real* und *String*) eigene Installationsfunktionen. Die angestoßene Aktion muß dann das Datum nicht selbst aus der Leitung lesen, sondern wird automatisch mit dem entsprechenden Parameter angestoßen.

Zur Laufzeit muß ein Tycoon-Programm in einer Nachrichtenschleife (*event loop*) auf ankommende Ereignisse warten. Zum Aufbau dieser Schleife kann die nachfolgend beschriebene Funktion *waitAndDispatch* verwendet werden.

```
let waitAndDispatch() :Ok =
  begin
    let tag = wire.peekTag()
    if tag == 0 then
      raise error
    else
      let action = dynArray.get(events tag)
      if action != unused then
        wire.readTag()  action()
      else
        wire.notify()  ok
      end
    end
  end
```

Wird die Funktion aufgerufen, so wird in der Leitung nachgesehen (*wire.peekTag*), ob ein Assoziator vorliegt. Liegt dort ein gültiger Assoziator vor, so wird aus dem dynamischen Feld die dazugehörige Aktion extrahiert (*dynArray.get*). Anschließend wird der Assoziator aus der Leitung entfernt (*wire.readTag*) und die Aktion aufgerufen (*action*). Ansonsten wird die Kontrolle wieder an andere Prozesse übergeben (*wire.notify*).

Sitzungen

Das Tycoon-System besitzt eine interaktive Programmierumgebung. Diese ermöglicht, zusammen mit dem Konzept der Persistenz, ein inkrementelles Arbeiten über mehrere Benutzersitzungen hinweg (vgl. Abschnitt 3.3). Daraus ergeben sich aber auch verschiedene Probleme. Zum Beispiel besteht nach dem Beenden einer Benutzersitzung keine Verbindung mehr zum X11/NeWS Server. Auch sind alle im Server instanziierten Objekte und alloziierten Bezeichner von Tycoon aus nicht mehr zugreifbar.

In einer neuen Benutzersitzung ist es daher notwendig, eine neue Verbindung zum Server aufzubauen und dort alle benötigten Objekte neu zu instanziieren und Bezeichner neu zu

alloziieren. Dies muß jedoch manuell erfolgen, da sich das Modul *wire* bereits im Objektspeicher befindet und nicht neu importiert wird. Aus diesem Grund gibt es in der Tycoon-Bibliothek ein Modul namens *session*. Es ermöglicht das Anmelden von Funktionen, die bei dem Start einer Benutzersitzung ausgeführt werden sollen. Im Falle des Moduls *wire* wird die folgende Sequenz verwendet:

```
session.appendInitialization(fun() new(defaultServer))
```

Diese Sequenz fügt den Aufruf der *new*-Funktion in die Initialisierung einer Benutzersitzung ein.

Auch beim Import des Moduls *action* werden Daten initialisiert, z.B. das Feld, in dem die Assoziatoren mit den dazugehörigen Aktionen gehalten werden. In einer neuen Benutzersitzung sind die Daten im Feld nicht mehr gültig. Das Feld muß neu initialisiert werden. Die folgende Sequenz führt dies durch:

```
session.appendInitialization(fun() events := dynArray.new(:T unused))
```

6.1.2 Schnittstelle zum NeWS Toolkit

Zu jeder Klasse des NeWS Toolkits gibt es im *newsenv* ein korrespondierendes Modul, dessen Name sich aus dem Klassennamen ohne das Prefix *Class* ergibt. In den Modulen sind jeweils die wichtigsten Methoden der Klasse durch Funktionen implementiert. Daraus ergeben sich einige Vorteile gegenüber dem direkten Methodenaufruf mit *wire.writeString*. Die Eingabeparameter der Methoden können durch Funktionsparameter ersetzt werden. Dadurch wird die Syntax der Aufrufe vereinfacht. Die Typisierung der Funktionsparameter erlaubt eine Typüberprüfung zur Übersetzungszeit und fördert damit die Fehlersicherheit. Durch die Definition der Funktionen wird eine leichtere Wiederverwendung ermöglicht.

Intern werden die Tycoon-Funktionen direkt auf die jeweiligen Methoden der zu dem Modul gehörenden Klasse abgebildet. Dies soll anhand von einfachen Beispielen verdeutlicht werden. Die Beispiele entstammen dem Modul *buttons*. In diesem Modul sind alle wesentlichen Operationsarten (vgl. Abbildung 6.2 auf Seite 77) zu finden, auf die die meisten Funktionen aus den Modulen des *newsenv* zurückzuführen sind. Die Operationsarten, auf die die Funktionen zurückzuführen sind, beinhalten das Erzeugen von Objekten (*create*), das Setzen (*set*) und Abfragen (*get*) von Eigenschaften von Objekten sowie das Installieren von Aktionen (*set notifier*) in Objekten.

In Abbildung 6.2 ist jeweils der auszuführende Tycoon- dem auszuführenden NeWS-Kode gegenübergestellt. Zusätzlich ist zu sehen, welche Daten durch die Leitung zum Server und umgekehrt gesendet werden. Bevor nun die Beispiele beschrieben werden, stellt der nächste Abschnitt zwei grundlegende Dienste des Moduls *object* vor, die für alle weiteren Module von Bedeutung sind.

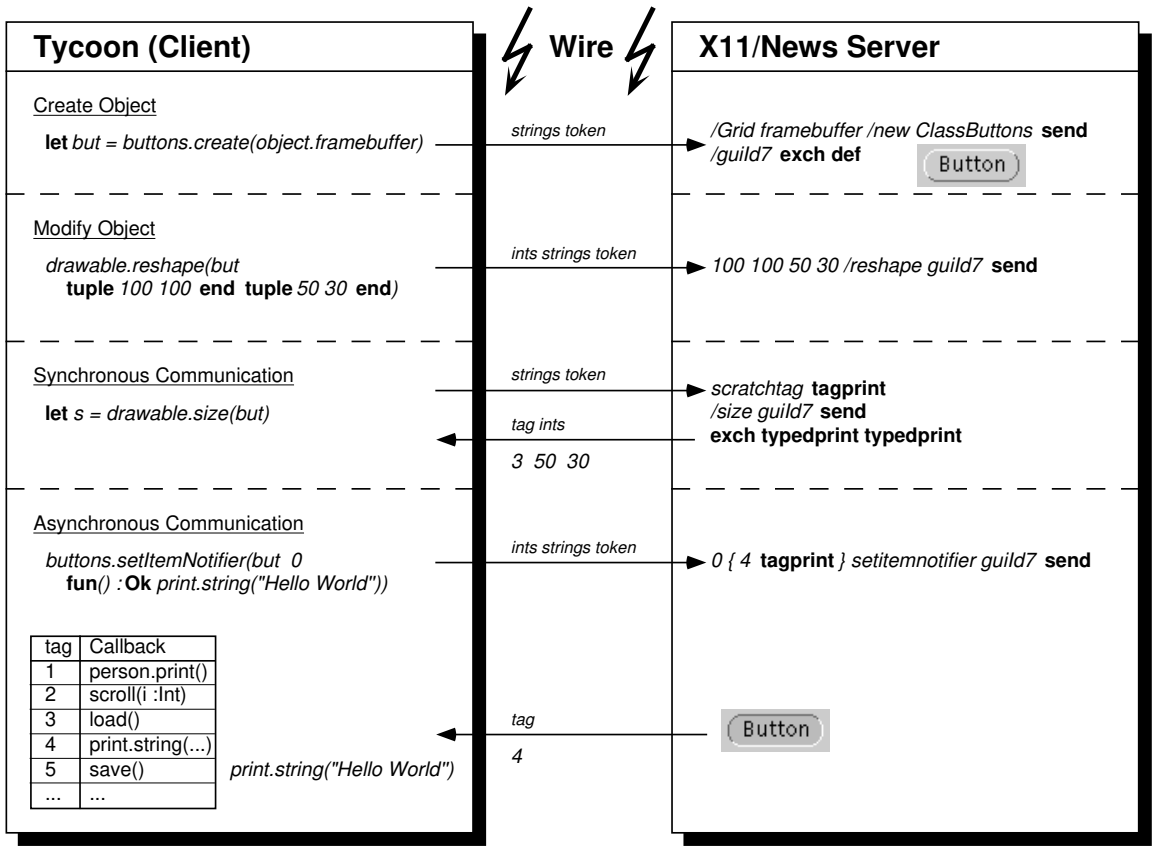


Abbildung 6.2: Kommunikation zwischen Tycoon und NeWS über die Leitung (*wire*)

Grundfunktionalität

Ein oft benötigter Dienst ist das Senden einer Nachricht an ein benanntes Objekt. Daraus ergibt sich auch der zweite wichtige Dienst des Moduls: Das Instanziieren und Benennen eines Objekts einer bestimmten Klasse. Für diese beiden Dienste stehen die Funktionen *object.send* und *object.new* zur Verfügung.

Der in Abschnitt 5.2.2 vorgestellte Methodenaufruf ist durch die Funktion *send* nachgebildet.

```
let send(obj :Object.T  message :String) :Ok =
    wire.writeString(message <> " " <> obj <> " send")
```

Der Funktion werden beim Aufruf Nachricht (*message*) und Empfangsobjekt (*obj*) übergeben. Intern wird aus den beiden Parametern eine Zeichenkette der Art "*message obj* **send**" konstruiert und mit der Funktion *writeString* in die Leitung geschrieben. Der Operator **send** sendet dann die Nachricht im Server an das Empfangsobjekt.

Um aber Nachrichten an Objekte senden zu können, müssen sie einen Namen bekommen, der sie eindeutig identifiziert. Unter diesem Namen sind die Objekte dann sowohl auf der NeWS- als auch auf der Tycoon-Seite identifizierbar. Die Zuweisung an einen Identifikator wird von Tycoon aus bei der Instanziierung des Objekts vorgenommen.

```
let new(class :String) :T =
  begin
    send(class "/new")
    let self = newToken()
    wire.writeString("/" <> self <> " exch def")
    self
  end
```

Nachdem durch *send(class "/new")* ein neues Objekt der Klasse *class* instanziiert worden ist, wird mit der Funktion *newToken* ein neuer Identifikator (*token*) generiert. Da nun aber der Identifikator über dem Objekt auf dem Kellerspeicher liegt, müssen vor der Definition durch den Operator **def** die beiden obersten Elemente des Kellerspeichers mit dem Operator **exch** vertauscht werden.

Erzeugen von Objekten

In jedem Modul des *newsenv*, das mit einer Klasse des NeWS Toolkits korrespondiert, gibt es eine *create*-Funktion. Sie wird benutzt, um Objekte zu instanziieren.

```
let create(parent :Canvas.T) :Buttons.T =
  begin
    wire.writeString("/Grid")
    wire.writeString(parent)
    object.new("ClassButtons")
  end
```

Mit dieser Funktion wird eine Gruppe von netzartig (*grid placement*) angeordneten Knöpfen erzeugt. Zu beachten ist, daß jedem Objekt bei der Instanziierung ein Vater (*parent*) zuzuordnen ist. Steht dafür z.Zt. kein Objekt zur Verfügung, so kann als Vater der Bildschirmhintergrund (*framebuffer*) angegeben werden.

Methodenaufruf ohne Rückgabewerte

Um die Eigenschaften von Schnittstellenkomponenten dynamisch ändern zu können, werden spezielle Funktionen benötigt. Da diese Funktionen keine Rückgabewerte erwarten, bestehen sie im allgemeinen nur aus einem Teil, in dem die benötigten Parameter in die Leitung geschrieben werden und dem eigentlichem Methodenaufruf mittels *object.send*.

An dem folgenden Beispiel ist auch ein weiterer Vorteil der Bibliothek zu sehen. In der Klassenhierarchie des NeWS Toolkits müssen gleichlautende Methoden nur einmal implementiert

werden. In der Hierarchie tiefer angeordnete Klassen erben die Implementation von ihren Superklassen. Auch in Tycoon brauchen solche Funktionen aufgrund des Subtyppolymorphismus, gleiche Parameterliste vorausgesetzt, nur einmal implementiert zu werden. Die Funktionen sind mit Instanzen unterschiedlicher Klassen parametrisierbar.

```
let reshape(d :Drawable.T  location :Location  size :Size) :Ok =
  begin
    wire.writeInt(location.x)
    wire.writeInt(location.y)
    wire.writeInt(size.width)
    wire.writeInt(size.height)
    object.send(d "/reshape")
  end
```

Im ersten Teil der Funktion *reshape* werden die gewünschte Position und Größe typisiert in die Leitung geschrieben. Anschließend wird die Methode *reshape* im Objekt *d* aufgerufen. Die Funktion erlaubt die Positionierung (*location*) und Definition der Größe (*size*) aller Objekte, die aus Subklassen der Klasse *ClassDrawable* instanziiert worden sind. Wie in Abbildung 5.5 auf Seite 62 zu sehen ist, sind fast alle Klassen direkte oder indirekte Subklassen der Klasse *ClassDrawable*. Aus diesem Beispiel läßt sich auch der geschickte Aufbau der Klassenhierarchie des NeWS Toolkits ableiten. Generell verwendbare allgemeine Methoden sind in in der Hierarchie weit oben anzutreffenden Klassen definiert, während spezielle, nur für die jeweilige Klasse benötigte Methoden neu definiert bzw. redefiniert werden müssen.

Methodenaufruf mit Rückgabewerten

Am Beispiel der *size*-Funktion wird die synchrone Kommunikation vorgestellt. Alle Funktionen im *newsenv*, die auf Methoden abgebildet werden, welche Rückgabewerte auf dem Kellerspeicher ablegen, sind in der gleichen Weise implementiert.

```
let size(d :Drawable.T) :Size =
  begin
    wire.writeString("scratchtag tagprint")
    object.send(d "/size")
    wire.writeString("exch typedprint typedprint")
    wire.scratchSync()
    wire.readTag()
    tuple wire.readInt()  wire.readInt() end
  end
```

Mit dieser Funktion kann die Größe aller Objekte des NeWS Toolkits abgefragt werden. Dazu wird zunächst das vorher definierte *scratchtag* vom Server aus in die Leitung geschrieben. Anschließend wird vom Objekt *d* die Größe bestimmt und auf dem Kellerspeicher abgelegt.

Damit die x-Koordinate zuerst gelesen werden kann, müssen die beiden Koordinaten auf dem Kellerspeicher vertauscht (**exch**) werden. Die Übertragung wird mit dem **typedprint** Operator vorgenommen. Auf der Tycoon-Seite wird mit *wire.scratchSync* auf das zuerst in die Leitung geschriebene *scratchtag* gewartet. Liegt es vor, kann es herausgelesen werden. Anschließend müssen nur noch die beiden Koordinaten gelesen und in ein Tupel verpackt werden.

Installation von Aktionen

Die letzte wichtige Operationsart ist die Installation von Aktionen an instanziierte Schnittstellenkomponenten (asynchrone Kommunikation). Dazu werden der Installationsfunktion i.d.R. neben dem Identifikator für die Schnittstellenkomponente noch die auszuführende Aktion übergeben. In diesem Beispiel wird zusätzlich noch der Index des Knopfes in der Gruppe, an dem die Aktion installiert werden soll, mit übergeben.

```
let setItemNotifier(but :Buttons.T  index :Int  act :Action.T) :Ok =
  begin
    wire.writeInt(index)
    wire.writeString("{ " <> fmt.int(action.new(act)) <> " tagprint } ")
    object.send(but "/setitemnotifier")
  end
```

Auch hier werden zunächst die benötigten Parameter auf dem Kellerspeicher abgelegt, bevor die Methode *setitemnotifier* aufgerufen wird. Sie installiert die übergebene Aktion an dem durch den Index ausgewählten Knopf. Die Benachrichtigungsprozedur besteht in diesem Fall aus der einfachsten, schon in Abschnitt 6.1.1 vorgestellten und beschriebenen Sequenz. Aus diesem Grund wird hier nicht weiter darauf eingegangen.

6.1.3 Schnittstelle zum Devguide

In Abschnitt 5.5.2 ist beschrieben worden, wie mit dem Devguide interaktiv Benutzerschnittstellen erzeugt werden können. Für einige Anwendungsklassen sind statische Benutzerschnittstellen ausreichend. Aus diesem Grund existiert in der Bibliothek auch ein Modul, dessen Funktionen es erlauben, solche Benutzerschnittstellen von Tycoon-Programmen aus anzusprechen und zu verwenden.

6.1.4 Schnittstelle zu benutzerdefinierten Klassen

Schnittstellen zu benutzerdefinierten Klassen lassen sich in der gleichen Weise implementieren, wie die Schnittstellen zu Klassen des NeWS Toolkits. Zu erwähnen ist nur, daß sich die Namensgebung der Methoden in den benutzerdefinierten Klassen in jedem Fall an die durch die Klassenhierarchie vorgegebene anlehnen sollte. Dies hat den Vorteil, daß auch in

benutzerdefinierten Klassen redefinierte Methoden über vorhandene Methoden des *newsenv* aufgerufen werden können.

Die neuen Klassendefinitionen müssen dem X11/NeWS Server vor der ersten Instanziierung bekannt gemacht werden. Am einfachsten und wartungsfreundlichsten ist es, die Klassendefinition beim Import des Moduls automatisch in den Server zu laden. Zu beachten ist dabei jedoch, daß die Definitionen nach Abbau der Verbindung verloren gehen (vgl. Abschnitt 6.1.1).

6.1.5 Texteditoren

Da die Funktionalität der Jot-Texteditoren als C-Bibliothek vorliegt, ist die Schnittstelle genau wie beim Wire Service über eine dynamische Anbindung von externen C-Bibliotheken an Tycoon realisiert. Mit den drei Modulen *jotText*, *jotView* und *jot* stehen in Tycoon alle wichtigen Funktionen des Jot-Textpakets zur Verfügung. Auf die einzelnen Funktionen soll in diesem Zusammenhang aber nicht weiter eingegangen werden.

6.2 Graphische Anwendungsprogrammierung

In diesem Abschnitt wird ein Projekt vorgestellt, in dem die Bibliothek zur dynamischen Erzeugung graphischer Benutzerschnittstellen produktiv eingesetzt worden ist. Darüber hinaus werden auch die Probleme skizziert, die beim Einsatz der Bibliothek aufgetreten sind.

Bei der Entwicklung und Implementation der graphischen Benutzerschnittstelle von Werkzeugen der STYLE-Umgebung (*Systematics of TYped Language Environments*) [Wetzel 94] ist das *newsenv* auf unterschiedliche Weise eingesetzt worden. Die STYLE-Umgebung stellt eine interaktive Entwurfsumgebung zum systematischen Design datenintensiver Anwendungen dar. Eingesetzt wurde die Bibliothek z.B. in einem Klasseneditor zur Entwurfsunterstützung [Kaß 94] und in einem Graphikeditor zur objektorientierten Datenmodellierung [Löst 94]. Im folgenden werden jeweils die Motivation und die Konzepte vorgestellt, die zur Entwicklung des Werkzeuges führten. Anschließend zeigt eine Abbildung den Aufbau der Benutzerschnittstelle des betrachteten Werkzeuges.

Ziel der folgenden Beispiele ist es nicht, in die STYLE-Umgebung oder die Werkzeuge vertiefend einzuführen, sondern vielmehr, einen kurzen Überblick über mögliche Einsatzgebiete der Bibliothek, deren Anforderungen und die gewählten Lösungen zu geben. Für einen vertiefenden Einblick in die STYLE-Umgebung und die Werkzeuge sei auf die angegebene Literatur verwiesen.

6.2.1 Ein Klasseneditor zur Entwurfsunterstützung

Die Aufgabe des multifunktionalen Texteditors ist die zielgerichtete Entwurfsunterstützung bei der Modellierung von Klassendefinitionen in einem objektorientierten Datenbanksystem.

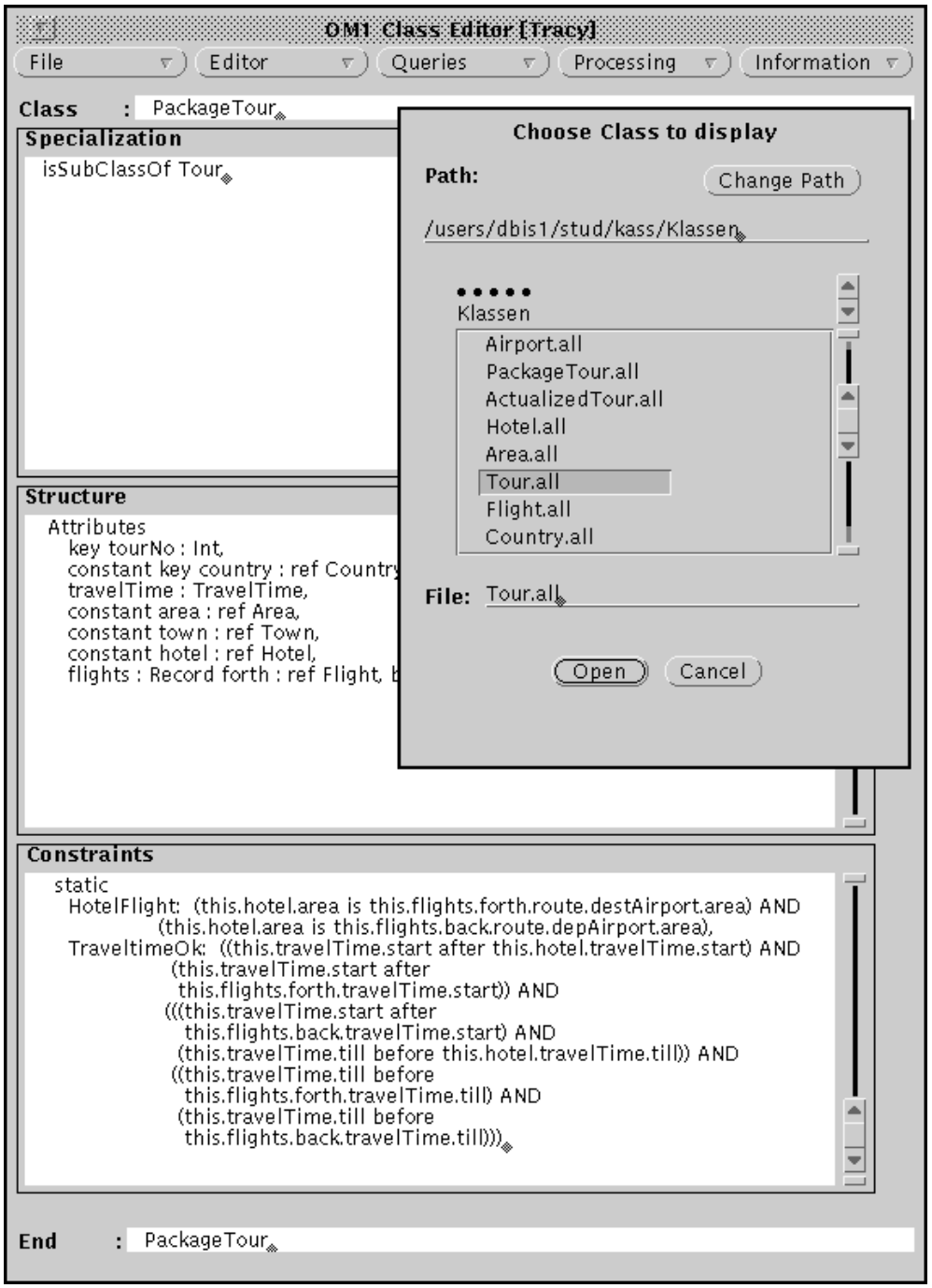


Abbildung 6.3: Multifunktionaler Klasseneditor zur Entwurfsunterstützung [Kaß 94]

Die graphische Benutzerschnittstelle des multifunktionalen Texteditors ist aus unterschiedlichen Schnittstellenkomponenten unter Benutzung des *newsenv* zusammengesetzt worden (vgl. Abbildung 6.3). Die wichtigsten dabei verwendeten Komponenten sind Fenster, Behälter, Rollbalken, Festtexte, Menüs, Knöpfe, Notizen, Listen sowie eine Dateiauswahlbox.

Durch die Benutzung der Bibliothek hat sich die Konstruktion des Editors wesentlich vereinfacht, da die gewünschte graphische Benutzerschnittstelle mit relativ geringem Aufwand nach dem Baukastenprinzip erstellt werden kann. Demgegenüber steht die mühsame und fehlerträchtige Programmierung der graphischen Benutzerschnittstelle in NeWS, die immer dann erforderlich wird, wenn eine benötigte Funktion nicht im Umfang der Bibliothek enthalten ist [Kaß 94].

Auch wird in der Arbeit auf einige Probleme des NeWS Toolkits hingewiesen. Zum Beispiel können einige Klassen nicht ohne weiteres miteinander kombiniert werden. Beispielsweise wird bei der Änderung der Größe eines Fensters dessen Inhalt nicht angepaßt, d.h.er bleibt in der ursprünglichen Größe bestehen. Ein weiteres Problem ergibt sich bei einer starken Belastung des Servers mit der Leitung zum X11/NeWS Server. In solchen Situationen stehen Befehlssequenzen teilweise noch zur Hälfte im Puffer, während die andere Hälfte schon abgearbeitet wurde. Bei nicht rechtzeitiger Freigabe des Puffers wird daher vergeblich auf die fehlenden Informationen gewartet. Durch eine Freigabe des Puffers nach zeitkritischen Funktionen ist dieses Problem jedoch in den Griff zu bekommen. Darüber hinaus ist eine gelegentliche Restauration des Bildschirms notwendig unumgänglich [Kaß 94].

6.2.2 Ein Graphikeditor zur objektorientierten Datenmodellierung

Der Graphikeditor ist ein Werkzeug zur objektorientierten Datenmodellierung in der STYLE-Umgebung. Er dient der Erstellung von Diagrammen eines im Rahmen einer Studienarbeit [Löst, Möller 92] entworfenen semantischen Datenmodells. Dabei werden, wie in Abbildung 6.4 zu sehen ist, in einem Referenzdiagramm die Klassen eines Schemas als Knoten mit ausblendbaren Werteattributen modelliert. Die referentiellen Beziehungen werden als Kanten zwischen den Klassen dargestellt. In einem Vererbungsdiagramm werden Vererbungsbeziehungen zwischen Klassen als Pfeile zwischen den Klassenknoten angezeigt.

Bei der Implementation der Graphikfunktionalität ergibt sich ein Problem aus der Tatsache, das die oben genannten Diagramme anwendungsspezifische graphische Bausteine beinhalten, die nicht in den OPEN LOOK-Stilrichtlinien spezifiziert werden. Zur Lösung dieses Problems bieten sich grundsätzlich zwei Strategien an. Zum einen kann das *newsenv* verwendet werden, um die neuen graphischen Bausteine zu erzeugen und zu verwalten. Diese Vorgehensweise ist zwar portabel, macht aber intensiven Gebrauch von den bidirektionalen Kommunikationsmöglichkeiten des Wire Service. Zum anderen können neue NeWS-Klassen implementiert werden, die die benötigten graphischen Bausteine und Operationen zur Verfügung stellen. Diese Vorgehensweise nutzt auch die Möglichkeit, komplexe Programme direkt im X11/NeWS Server ausführen zu können. Dadurch werden auch die im vorherigen Abschnitt aufgeführten Probleme des NeWS Toolkits zum Teil umgangen. Die neuen Klassen erhalten

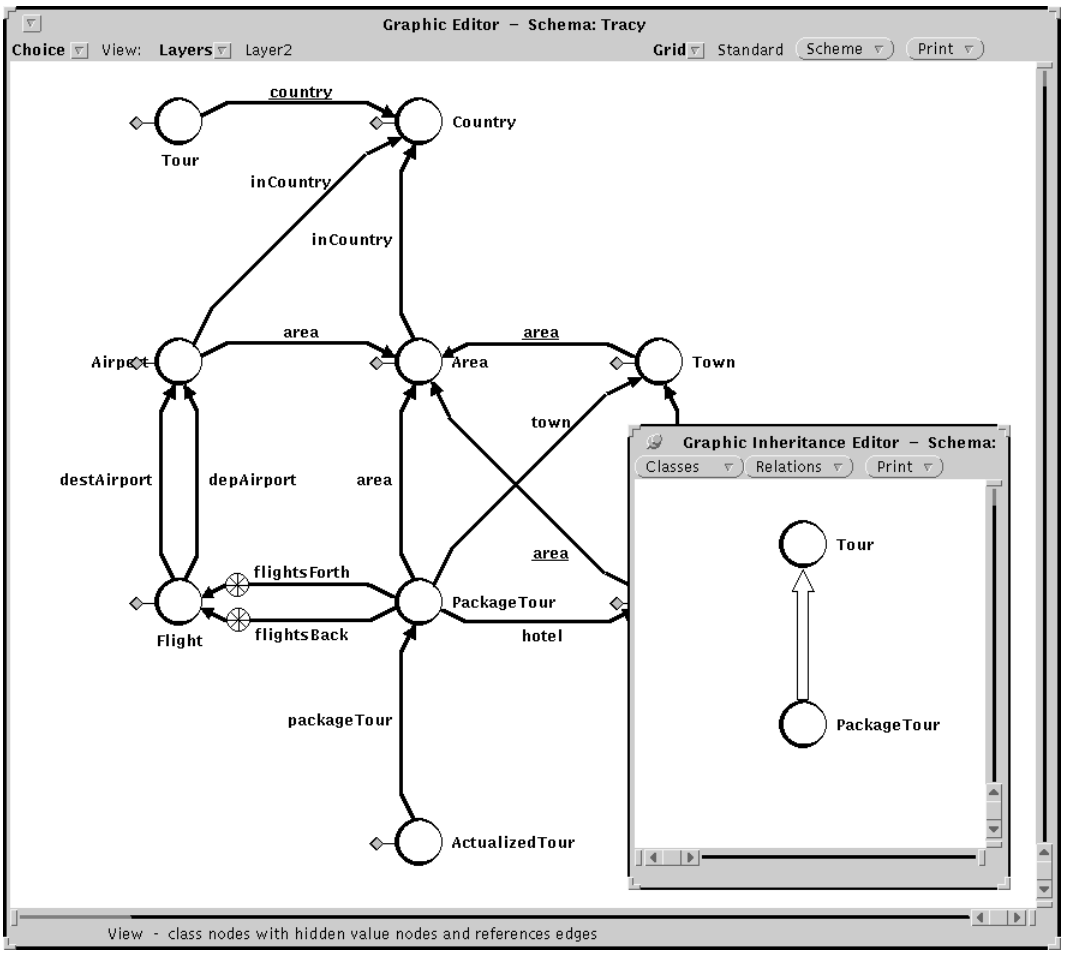


Abbildung 6.4: Graphikeditor zur Visualisierung [Löst 94]

bei Bedarf, genau wie die Klassen des *newsenv*, eine Schnittstelle in Tycoon. Die gewählte Strategie wird im folgenden kurz beschrieben.

Für den Aufbau der graphischen Benutzerschnittstelle aus OPEN LOOK-Schnittstellenkomponenten, wie Grund- und Aufklappfenstern, Menüs, Knöpfen und Rollbalken, sowie für das Anhängen von Aktionen an Menüpunkte werden Funktionen aus Modulen des *newsenv* verwendet. Um die von OPEN LOOK nicht spezifizierte Funktionalität zur Erzeugung, zur Manipulation und zur Anzeige der Diagramme zu erhalten, werden neue Schnittstellen in Tycoon implementiert. Durch Nutzung der Basisdienste *wire* und *action* aus dem *newsenv* wird der Implementationsaufwand für die Funktionen, die die Methoden der neu definierten NeWS-Klassen aufrufen, verringert [Löst 94].

7. Datenvisualisierung in Tycoon

Ausgangspunkt für die typsichere generische Datenvisualisierung in Tycoon ist das Konzept der *beliebig kombinierbaren Editoren*. Editoren sind Objekte, die eine einheitliche Präsentation (*look*) und Dialogsteuerung (*feel*) gemeinsam haben. Editoren dienen der typsicheren Visualisierung atomarer und hierarchisch strukturierter Werte des Tycoon-Objektspeichers. Ermöglicht wird das durch eine für den jeweiligen Datentyp geeignete *Bildschirmrepräsentation*. Die Daten in der Bildschirmrepräsentation können vom Anwender interaktiv verändert werden.

Die Editoren lassen sich in drei Klassen unterteilen. Die einfachsten Editoren finden sich in der Klasse der *Basiseditoren*. Sie entsprechen weitgehend den in Tycoon vorhandenen Basistypen. Die Klasse der *Verbundeditoren* beinhaltet einfache hierarchische Editoren. Sie entsprechen weitgehend den in Tycoon vorhandenen Typkonstruktoren. Die Klasse der *Massendateneditoren* implementiert generische Editoren zur Visualisierung von Massendaten.

In diesem Kapitel werden die Konzepte und die Implementation der typsicheren generischen Editoren ausführlich vorgestellt. Die Vorstellung der Editoren beginnt mit der Beschreibung der Funktionalität sowie mit Abbildungen über die Gestaltung der graphischen Benutzerschnittstelle. Anschließend wird anhand des schon aus Kapitel 4 bekannten Programmbeispiels aus Anhang A die Anwendungsprogrammierung mit der Editorbibliothek vorgestellt. In einem weiteren Abschnitt werden ausgewählte Beispiele aus den zur Bibliothek gehörenden Schnittstellen und Modulen erläutert. Es folgt abschließend ein Abschnitt über Möglichkeiten bei der Erweiterung der Bibliothek durch Anwender am Beispiel von abstrakten Datentypen.

7.1 Gestaltung der graphischen Benutzerschnittstelle

Die graphische Benutzerschnittstelle der einzelnen Editoren ist durch die Verwendung des NeWS Toolkits (vgl. Abschnitt 5.4) und OPEN LOOK (vgl. Abschnitt 5.3) in ihrem Aussehen und Verhalten geprägt. Die Auswahl der speziellen Visualisierungen erfolgte anhand praktischer Überlegungen. Der erfolgreiche praktische Einsatz des Prototyps (vgl. [Kirch, Müßig 92]) sowie der Vergleich mit anderen Systemen (vgl. Kapitel 4) hat die vorgenommene Gestaltung der Visualisierungen zusätzlich bekräftigt.

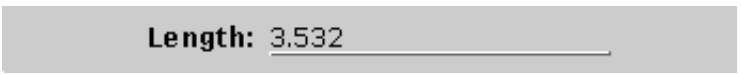
Um ein Gefühl für das Aussehen und das Verhalten der Editoren zu vermitteln, werden in den folgenden Abschnitten alle relevanten Komponenten des Systems anhand von Abbildungen vorgestellt. Zu jeder Abbildung wird die von der jeweiligen Visualisierung abhängige Funktionalität ausführlich beschrieben.

7.1.1 Basiseditoren

In diesem Abschnitt werden die Visualisierungen der in Tycoon vorhandenen Basistypen sowie von Aufzählungen und Funktionen vorgestellt. Alle Editoren verfügen über einen Grundwert, auf den sie sich zurücksetzen lassen. Der Grundwert kann über die Auswahl *New* im *Edit*-Menü angesprochen werden (vgl. Abschnitt 7.1.4).

Ganzzahlen: Werte des Datentyps *Int* werden durch ein numerisches Eingabefeld repräsentiert. Mit Hilfe der beiden Knöpfe auf der rechten Seite des Feldes ist eine schrittweise In- bzw. Dekrementierung des angezeigten Wertes möglich. Ein solcher Editor kann auf seinen Grundwert, in diesem Fall *0*, zurückgesetzt werden.

Fließkommazahlen: Werte des Datentyps *Real* werden in der gleichen Weise repräsentiert wie ganzzahlige Werte. Es gibt jedoch keine Knöpfe zum In- bzw. Dekrementieren. Der Grundwert ist *0.0*.

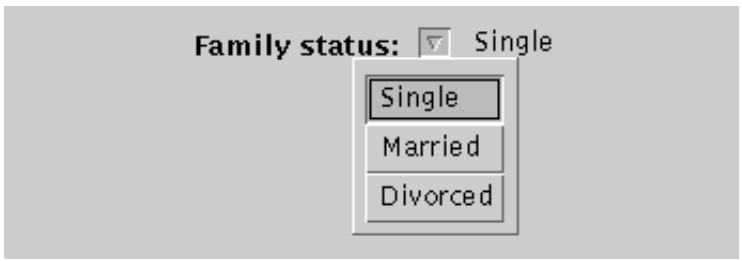


Zeichen: Einzelne Zeichen (*Char*) werden durch ein alphanumerisches Eingabefeld der Länge eins repräsentiert. Der Grundwert ist ein leeres Zeichen (''').

Zeichenketten: Werte des Datentyps *String* werden durch ein alphanumerisches Eingabefeld repräsentiert. Reicht der zur Verfügung stehende Platz nicht aus, so erscheinen an beiden Rändern Knöpfe, mit denen es möglich ist, durch den Text zu navigieren. Zwei weitere Zeichenketteneditoren ermöglichen die Visualisierung von nur lesbaren (*read-only*) und festen Werten. Der Grundwert ist eine leere Zeichenkette ("").

Boole'sche-Werte: Boole'sche Werte (*Bool*) werden durch ein Ja/Nein-Feld repräsentiert. Der Wert *true* entspricht dabei einem Häkchen im Feld. Der Grundwert ist *false*.

Aufzählungen: Auf der Typebene von Tycoon wird ein Aufzählungstyp als degeneriertes Tuple mit Varianten (***Tuple case*** ... ***end***) dargestellt. Werte von Aufzählungstypen werden durch eine Exklusivauswahl an einem Abkürzungsmenüknopf visualisiert. Die Werte der Aufzählung sind die Einträge in der Auswahl. Der aktuelle Wert ist rechts neben dem Menüknopf zu finden. Als Grundwert ist jeweils die erste Variante gewählt.



Funktionen: Funktionswerte der Form ***fun*** *() :**Ok*** werden durch einen Knopf repräsentiert. Die parameterlose Funktion wird ausgeführt, wenn der Knopf gedrückt wird. Der Grundwert für Funktionen ist **fun**() **ok**.

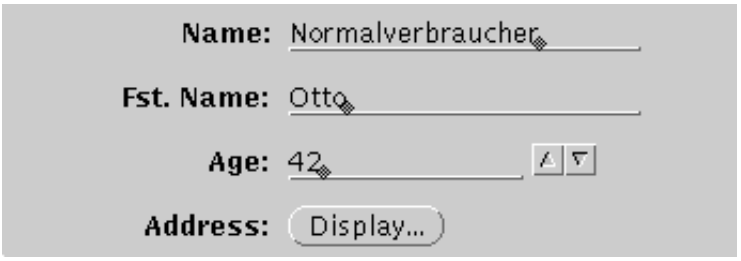
7.1.2 Verbundeditoren

In der gleichen Weise wie auf der Typebene von Tycoon werden auch Editoren für strukturierte Werte systematisch durch Aggregation ihrer Komponenten erzeugt. Als Komponenten von Verbundeditoren kommen nicht nur die im letzten Abschnitt vorgestellten Basiseditoren in Betracht, sondern alle definierten Editoren können als Komponenten verwendet werden.

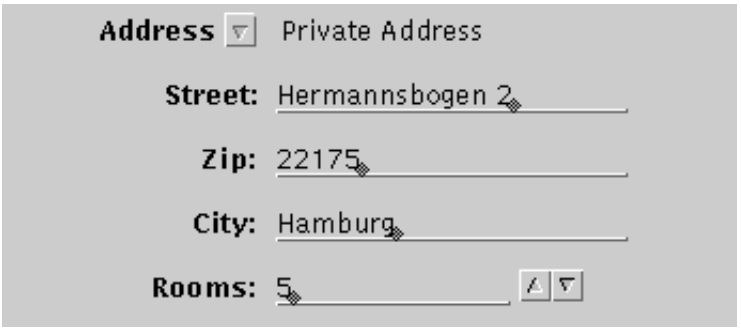
Dabei sei zunächst darauf hingewiesen, daß Verbundeditoren typischerweise zwei Visualisierungen haben. Die erste Visualisierung ist für alle Verbundeditoren unabhängig von ihren Komponenten gleich. Sie wird benutzt, wenn ein Verbundeditor als Komponente in einem anderen Editor eingeschachtelt ist.

Nach einem Druck auf den Knopf öffnet sich ein weiteres Fenster, in dem die eigentliche Visualisierung des Editors dargestellt wird. Eine genaue Beschreibung der Funktionalität der beiden Verbundeditoren folgt in der folgenden Aufzählung.

Verbunde: Werte von Verbundtypen (***Tuple*** ... ***end***) werden durch ihre Komponenten dargestellt. Alle Komponenten werden linksbündig, die Namen der Komponenten rechtsbündig dargestellt. Der Grundwert ergibt sich aus den Grundwerten der Komponenten.



Varianten: Werte von Variantentypen werden durch die Kombination von Aufzählungstyp und Verbund visualisiert. Die verschiedenen Varianten sind über einen Abkürzungsmenüknopf auswählbar. Jede Variante beinhaltet einen Verbund zur Visualisierung der Komponenten. Die jeweils ausgewählte Variante wird rechts neben dem Menüknopf dargestellt. Der Grundwert ergibt sich aus den Grundwerten der Komponenten der ersten Variante.

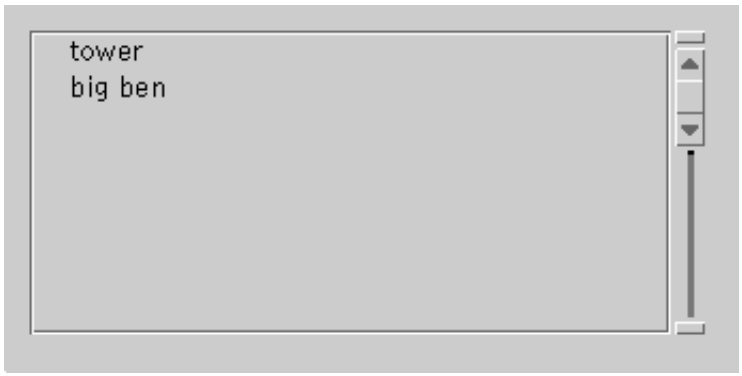


7.1.3 Massendateneditoren

Die nächste Stufe nach den Verbundeditoren bilden die Massendateneditoren. Massendatentypen bestehen meist aus der Kombination eines Basis- oder Verbundtyps mit einem Kollektionstyp wie Liste oder Menge. Entsprechend sind die Visualisierungen der Massendatentypen aufgebaut. Wie schon im vorherigen Abschnitt erwähnt, haben hierarchische Editoren typischerweise zwei Visualisierungen, eine für eingeschachtelte Editoren und eine für den eigentlichen Editor. Grundwerte ist jeweils eine leere Kollektion.

Iterationen: Iterationen werden im Prinzip wie Verbunde visualisiert. Sie enthalten jedoch zusätzlich einen Rollbalken, mit dem innerhalb der Datenstruktur navigiert werden kann. Außerdem kennt ein Iterationseditor die Anzahl seiner Elemente und die Position des angezeigten Elements.

Tabellen: Tabellen bilden die zweite Darstellungsmöglichkeit für Massendaten. Im Gegensatz zum Iterationseditor, in dem immer nur ein Datensatz z.Zt. angezeigt werden kann, können in einem Tabelleneditor mehrere Datensätze gleichzeitig angezeigt werden. Die Komponenten werden spaltenweise dargestellt, wobei die Namen der Komponenten die Spaltenüberschriften bilden. Ein Rollbalken am rechten Rand ermöglicht die Navigation in der Tabelle. Auch ein Tabelleneditor kennt die Anzahl seiner Elemente. In der Tabelle selbst können die Daten nicht manipuliert werden. Durch die Selektion einer Zeile wird ein Verbundeditor geöffnet, in dem der entsprechende Datensatz angezeigt und manipuliert werden kann.



7.1.4 Rahmen für Editoren

Alle in den letzten drei Abschnitten vorgestellten Visualisierungen für Editoren können in dieser Form nicht in einem Fenstersystem verwendet werden. Um solche Visualisierungen sinnvoll einsetzen zu können, bedarf es eines Rahmens, der eine Grundfunktionalität zur Verfügung stellt. Ein solcher Rahmen ist in Abbildung 7.1 dargestellt. Bei dem Rahmen

handelt es sich, je nach Hierarchiestufe, entweder um ein Grundfenster oder um ein Aufklappfenster. Die Funktionalität ist jedoch für beide Rahmen weitgehend gleich.

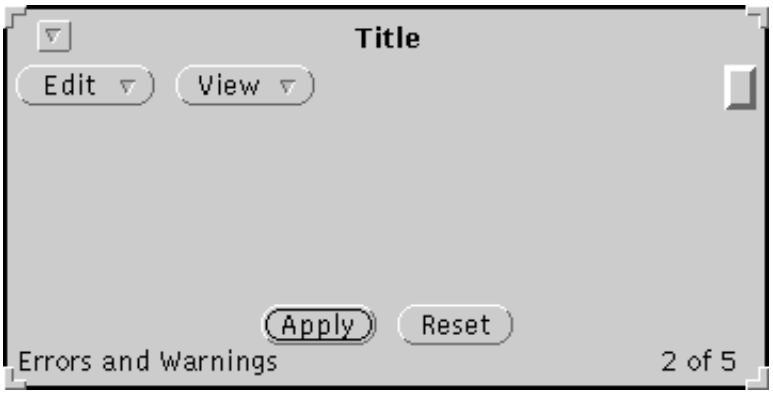


Abbildung 7.1: Grundrahmen für Editoren

Unterschieden werden die Rahmen lediglich beim Schließen. Während Grundfenster ausschließlich durch Auswahl des Menüpunktes *Quit* aus dem Menü der Kopfzeile des Fensters geschlossen werden können, können Aufklappfenster sowohl durch Herausziehen der Stecknadel als auch durch Auswahl des Menüpunktes *Dismiss* geschlossen werden.

Beim Schließen eines Rahmens werden intern zwei Fälle unterschieden. Sind die Werte im Editor nicht geändert worden, so wird der Rahmen sofort ohne Nachfrage geschlossen. Sind die Werte in einem Editor jedoch geändert worden und noch nicht gesichert, so erscheint eine Abfrage. Dadurch ist es dem Anwender möglich, den Editor mit oder ohne Sicherung der Daten zu verlassen oder in den denselben zurückzukehren.

Unter der Kopfzeile des Rahmens befindet sich die Menüzeile. Diese besteht aus zwei Teilen, dem eigentlichen Menü auf der linken Seite und einem Datenaustauschfeld auf der rechten Seite. Direkt unter der Menüzeile wird die Visualisierung des eigentlichen Editors eingesetzt. Als Visualisierungen kommen alle in den vorherigen Abschnitten vorgestellten Editoren in Frage. Unter dem Editor befinden sich noch zwei Knöpfe und die Fußzeile.

Kopfzeile

In der Kopfzeile des Rahmens wird der Name des direkt integrierten Editors angezeigt.

Menüs

Unter der Kopfzeile des Rahmens befindet sich die Menüzeile. Auf der linken Seite der Menüzeile sind zwei Menüknöpfe zu finden. Unter den beiden Menüknöpfen (*Edit* und *View*) befinden sich mehrere Menüeinträge, mit denen entsprechend assoziierte Tycoon-Funktionen aufgerufen werden können. In Tabelle 7.1 sind alle Menüeinträge der beiden Menüpunkte mit ihren Submenüeinträgen und der dazugehörigen Funktionalität aufgeführt und erläutert.

Menü	Eintrag	Subeintrag	Funktionalität
Edit	*Edit*		Öffnet Editor mit ausgewähltem Element.
	New	*Here*	Fügt Element an aktueller Position ein.
		Behind	Fügt Element hinter aktueller Position ein.
	Cut	*This*	Löscht aktuelles Element bzw. setzt Grundwert.
		All	Löscht alle Elemente.
		Selected	Löscht ausgewählte Elemente.
	Copy		Kopiert in Puffer.
	Paste		Kopiert aus Puffer.
View	*Properties*		Grundeinstellungen setzen.
	Show Path		Zeigt Pfad zum Editor an.

Tabelle 7.1: Funktionalität der Menüzeile im Rahmen

Nicht alle Menüeinträge stehen in allen Editoren zur Verfügung. Zum Beispiel macht es keinen Sinn, in einen Basiseditor ein neues Element einzufügen (*new*) oder einen neuen Editor mit dem gleichen Element zu öffnen (*edit*). Alle in einem Editor nicht zur Verfügung stehende Menüeinträge sind als inaktiv markiert. Sie können daher vom Anwender auch nicht angewählt werden.

Datenaustauschfeld

Auf der rechten Seite der Menüzeile befindet sich ein Datenaustauschfeld. Das Feld ermöglicht das symbolische Austauschen von Daten (vgl. Abschnitt 5.4.4).

Beim Datenaustausch können zwei Arten unterschieden werden, zum einen der Austausch von Daten zwischen Editoren (*internal data exchange*) und zum anderen der Austausch von Daten zwischen einem Editor und einem anderen Programm (*external data exchange*). Der externe Datenaustausch kann z.B. als Import- bzw. Exportfunktion verwendet werden. Für Anwender besteht jedoch hinsichtlich des Ablaufs kein Unterschied. Die Daten werden in einem Fenster symbolisch über das Datenaustauschfeld oder direkt mit der Maus aufgenommen. So aufgenommene Daten können mit der Maus über den Bildschirm bewegt und über einem anderen Fenster oder Datenaustauschfeld fallengelassen werden. Die beiden am Datenaustausch beteiligten Anwendungsprogramme sind durch das Aufnehmen und Fallenlassen der Daten informiert und regeln den eigentlichen Austausch selbständig.

Knöpfe

Unter dem Editor befinden sich zwei Knöpfe. Sie dienen dem Datenaustausch zwischen Anwendung und Editor. Der *Reset*-Knopf liest den aktuellen Wert aus der zugrundeliegenden Datenstruktur bzw. Variable aus und zeigt ihn auf dem Bildschirm an und der *Apply*-Knopf schreibt den aktuellen Wert in die Datenstruktur bzw. Variable unter Beachtung eventueller Integritätsbedingungen zurück.

Fußzeile

In der Fußzeile des Rahmens werden Rückmeldungen für Anwender ausgegeben. Die linke Seite ist dabei für Warn- und Fehlermeldungen, wie verletzte Integritätsbedingungen, vorgesehen. Auf der rechten Seite werden Zustandsmeldungen, wie z.B. die Nummer des aktuellen Elements und die Gesamtzahl aller Elemente im Editor, angezeigt.

7.2 Anwendungsprogrammierung am Beispiel

Nachdem in den vorherigen Abschnitten die Bildschirmrepräsentationen der einzelnen Editoren und des Rahmens vorgestellt worden sind, werden nachfolgend anhand von einfachen Beispielen einige Verwendungsmöglichkeiten der generischen Editoren aufgezeigt. Zuvor wird anhand einer schematischen Abbildung der Datenfluß zwischen den Programmvariablen und dem Bildschirm dargestellt.

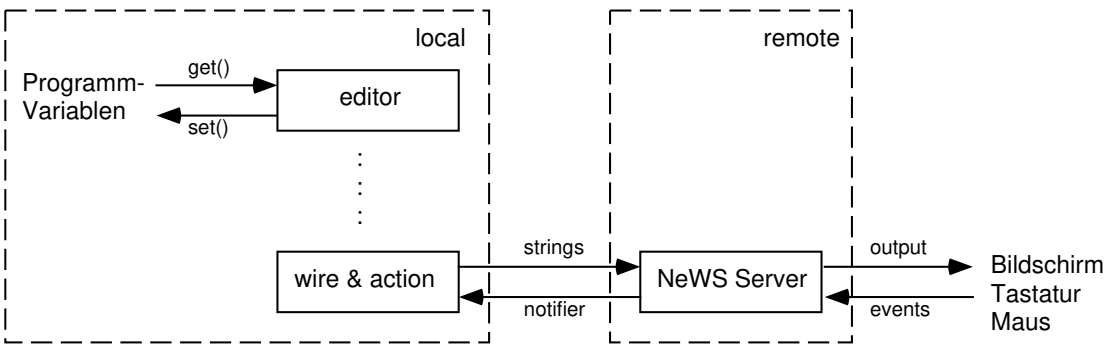


Abbildung 7.2: Datenfluß bei der Visualisierung

Gesteuert wird der Datenfluß von den einzelnen verwendeten Editoren. In Abbildung 7.2 ist zu sehen, wie über die Funktion *get* die Programmvariablen ausgelesen werden. Die Editoren bereiten die Werte, je nach Typ, entsprechend auf und reichen sie an tiefere Schichten weiter. Erst über das Modul *wire* aus dem *newsenv* findet die Kommunikation mit dem X11/NeWS Server statt. Der Server ist für den Bildschirmaufbau und die Kommunikation mit dem Anwender verantwortlich. Umgekehrt werden die Aktionen vom Anwender über das Modul *wire* an die Editoren weitergeleitet. Sie sind dann dafür verantwortlich, die evtl. veränderten Werte unter Beachtung vorhandener Integritätsbedingungen über die *set*-Funktion in die Programmvariablen zurückzuschreiben.

7.2.1 Visualisierung unstrukturierter Werte

Das nachfolgende Beispiel zeigt, wie auf einfache Weise ein Editor für Zeichenketten erzeugt werden kann. Dafür muß das nachfolgende Tycoon-Kodefragment ausgeführt werden.

Der Anwender bekommt dann den abgebildeten Editor (vgl. Abbildung 7.3) zur Verfügung gestellt.

```
import
  editor  action

let var name = "Meyer"
let var continue = true
let quit :Action.T = fun() continue := false
let dispatch = fun() continue

let stringEditor = editor.newString("My First Editor"
  let get = fun() name
  let set = fun(new :String) name := new)

begin
  editor.display(stringEditor  quit  editor.somewhere)
  editor.waitAndDispatch(dispatch)
  name
end
```

Abbildung 7.3: Einfacher Editor zum Anzeigen einer Zeichenkette

Die gesamte Flexibilität der Editoren liegt in den Funktionen *get* und *set*. Mit Hilfe der Funktion *get* kann nicht nur eine einfache Variable ausgelesen werden, sondern es ist auch möglich, Werte aus beliebig komplizierten Strukturen auszulesen bzw. zu berechnen.

```
let get = fun() person.address.street
let get = fun() data.netto ** data.tax
```

Die Funktion *set* dient dem Zurückschreiben des Wertes in die zugrundeliegende Variable der Datenstruktur. Dadurch können an dieser Stelle beliebige Integritätsbedingungen getestet werden und es ist möglich, auf unzulässige Werte hinzuweisen.

```
let set = fun(new :String)
  if string.empty(new) then
    raise editor.illegal with "No name specified" end
  else
    name := new
  end
```

In diesem Beispiel wird geprüft, ob der Anwender einen Namen eingegeben hat. Ist dies nicht der Fall, so wird eine Ausnahme ausgelöst. In der Fußzeile des Editors erscheint die Meldung *"No name specified"*. Der Anwender kann den Wert des Editors ändern. Der Editor kann erst verlassen werden, wenn keine Integritätsbedingung mehr verletzt oder der neue Wert nicht in die Datenstruktur übernommen wird.

In dem Beispiel kommen noch weitere Funktionen vor. Ihre Aufgabe wird im folgenden erläutert. Mit der Funktion *waitAndDispatch* (vgl. Seite 101) wartet das Anwendungsprogramm in einer Schleife auf ankommende Ereignisse. Nach jedem Ereignis wird über die *dispatch*-Funktion geprüft, ob die Nachrichtenschleife verlassen werden soll. Dieser Zustand tritt ein, wenn ein Anwender *Quit* aus dem Menü des Grundrahmens auswählt. In diesem Fall setzt die Funktion *quit* die Variable *continue* auf *false* und die Funktion *waitAndDispatch* wird verlassen. Der Editor kann durch einen erneuten Aufruf der Funktion *display* wieder angezeigt werden. Vor dem Einstieg in die Nachrichtenschleife mit *waitAndDispatch* muß die Variable *continue* unbedingt wieder auf *true* gesetzt werden, da ansonsten die Schleife sofort wieder verlassen wird.

7.2.2 Visualisierung strukturierter Werte

Das zweite Beispiel soll verdeutlichen, wie aus einem Anwendungsprogramm heraus ganze Datensätze angezeigt und geändert werden können. Das Modul *world* mit den Datenstrukturen und Editordefinitionen, auf die im folgenden Bezug genommen wird, befindet sich im Anhang A.1. Das Ergebnis eines Aufrufs der Funktion *go* des Moduls *world* ist in Abbildung 7.4 zu finden.

Das Modul *world* importiert zunächst andere Module, deren Funktionen im weiteren gebraucht werden. Anschließend werden die Datenstrukturen definiert. Bevor die Editoren implementiert werden können, müssen noch einige Hilfsfunktionen definiert werden. Zum Beispiel wird für den Aufruf eines Iterationseditors eine Funktion zum Erzeugen eines neuen Elements (*newPerson*) benötigt, da es in Tycoon keine uninitialisierten Variablen gibt.

Für den Editor zum Visualisieren von Werten des Aufzählungstyps *FamilyStatus* müssen zusätzlich zwei weitere Funktionen definiert werden. Die eine Funktion dient dem Berechnen der Ordinalität eines Wertes und die andere erlaubt das Abbilden von Aufzählungswerten auf Zeichenketten. Die Abbildungsfunktion erzeugt eine Iteration von Paaren. Die Paare bestehen jeweils aus einem Wert des Typs *FamilyStatus* und der Zeichenkette, auf die der Wert abgebildet werden soll. Die Zeichenkette wird zum Visualisieren der Werte von Aufzählungstypen verwendet.

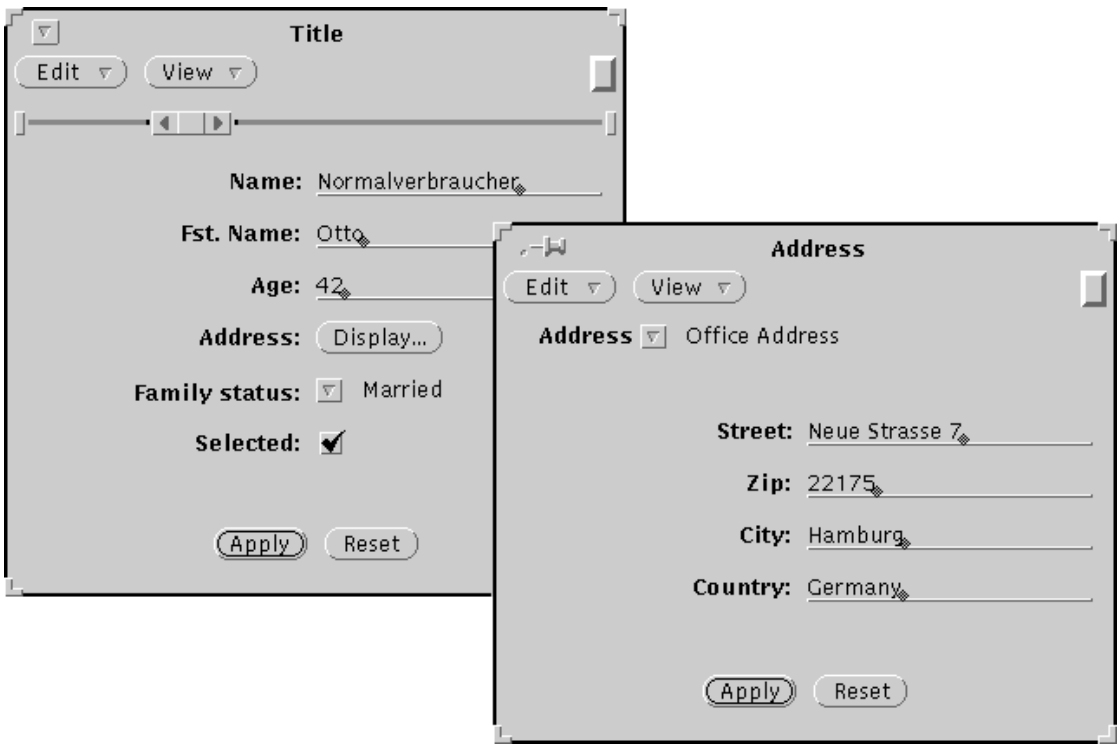


Abbildung 7.4: Editor für das Programmbeispiel aus Anhang A

Die eigentlichen Editoren können nun definiert werden. Die Definition läuft dabei in der gleichen Weise ab wie schon bei dem einfachen Editor im vorherigen Abschnitt. Auch die Integritätsbedingungen werden in der gleichen Weise aufgeführt. Einmal definierte Editoren können dabei in die Definition von komplexeren Editoren eingefügt werden. Ein Beispiel dafür ist der *addressEditor* im *personEditor*.

Zum Anzeigen des Editors wird die Funktion *display* benutzt. Anschließend wird wieder in einer Schleife auf ankommende Ereignisse gewartet.

7.3 Module und Schnittstellen der Editorbibliothek

Die Editorbibliothek (*editenv*) implementiert die in der Einleitung zu diesem Kapitel vorgestellten typsicheren generischen Editoren. Die Bibliothek greift dabei in vielfältiger Weise auf andere Bibliotheken, insbesondere auf das *stdenv* und das *newsenv* zurück. Im einzelnen besteht die Editorbibliothek, wie auch in Abbildung 7.5 dargestellt ist, aus elf Modulen, die sich in vier Schichten anordnen lassen. Angemerkt sei jedoch, daß Abbildung 7.5 nicht die komplexen Importbeziehungen, die innerhalb der Bibliothek bestehen, wiedergibt.

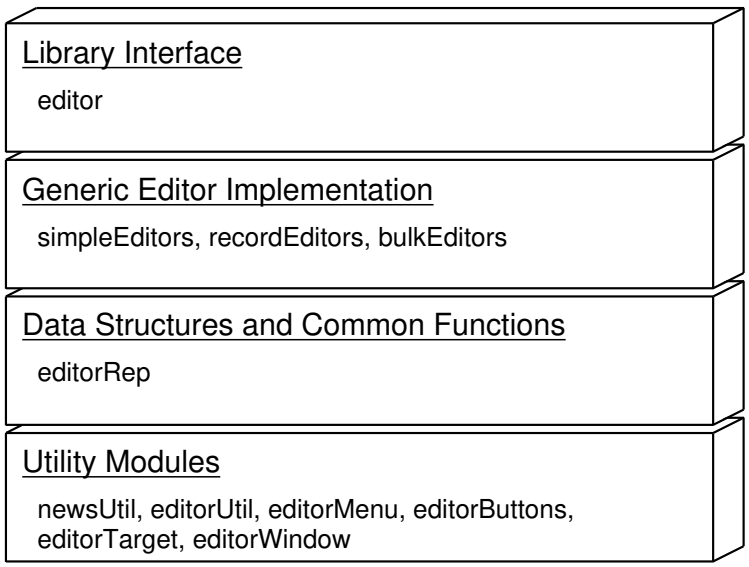


Abbildung 7.5: Struktur der Editorbibliothek

Die unterste Schicht wird von den Hilfsmodulen *newsUtil*, *editorUtil*, *editorMenu*, *editorButtons*, *editorTarget* und *editorWindow* gebildet. Das Modul *editorRep* aus der zweiten Schicht definiert im wesentlichen das abstrakte Protokoll, über das die Editoren angesprochen werden. In Schicht drei befinden sich die eigentlichen Editorimplementationen (*simpleEditors*, *recordEditors* und *bulkEditors*). Das Modul *editor* der obersten Schicht bildet die einzige nach außen hin sichtbare Schnittstelle. Alle wichtigen Definitionen werden an dieses Modul durchgereicht. In den folgenden Abschnitten werden die Konzepte und wichtigsten Implementationsideen der einzelnen Module ausführlich vorgestellt.

7.3.1 Abstraktes Datenstrukturmodell

Im Modul *editorRep* wird das Protokoll definiert, über das alle Editoren angesprochen werden. Darüber hinaus sind in diesem Modul alle Funktionen zu finden, die auf beliebige Editoren angewendet werden können.

Protokoll

Das Protokoll für Editoren ist eine Schnittstelle, die es ermöglicht, verschiedene Arten von Editoren in eine gemeinsame Umgebung zu integrieren und in objektorientierter Form anzusprechen [Kilberth et al. 93]. Alle definierten Editoren müssen mindestens die folgenden Komponenten haben, damit sie sich in die Umgebung nahtlos einfügen:

Im folgenden werden die einzelnen Komponenten des Protokolls ausführlich beschrieben. Die erste Komponente (*hidden*) dient der Zusammenfassung interner Daten, die zum Aufbau

```
Let Rec T <:Ok =
  Tuple
    hidden :Hidden(T)
    label :DisplayItem
    separateWindow :Bool
    menuActions :MenuActions
    create() :Ok
    destroy() :Ok
    apply() :Bool
    refresh() :Ok
    show() :Ok
    hide() :Ok
    getValue() :ValueRep
    setValue(:ValueRep) :Ok
    toString() :Iter.T(String)
  end
```

und zur Verwaltung einer hierarchischen Struktur benötigt werden. Alle Komponenten dieser Substruktur sind modifizierbar, da sich ihre Werte dynamisch ändern können. In der jetzigen Implementation können alle Komponenten des Tupels durch Anwender verändert werden. Da dieser Effekt nicht gewünscht ist, sollte die Komponente eigentlich nicht im Protokoll erscheinen. Mit Hilfe des Konzepts der Subtypisierung ließe sich die Komponente aus der Schnittstelle des Moduls entfernen. Der Typ *Hidden* hat die folgende Struktur:

```
Let Hidden(EditorT <:Ok) <:Ok =
  Tuple
    var guiId :Object.T
    var windowId :Window.T
    var menuBar :MenuBar
    var target :DropTarget.T
    var buttons :Buttons.T
    state :set.T(State)
    var key :Bool
    var modified :Bool
    var parent :optional.T(EditorT)
  end
```

In der *guiId* wird der Identifikator für die Bildschirmrepräsentation des Editors gehalten. Dies ist notwendig, da an die Objekte Nachrichten geschickt werden müssen, z.B. um einen neuen Wert darzustellen oder weitere Rahmen zu öffnen. In den vier folgenden Komponenten sind nur dann Einträge, wenn sich der Editor direkt in einem Rahmen befindet. In der

windowId steht der Identifikator für den Rahmen, im *menuBar* für die Menüzeile, im *target* für das Datenaustauschfeld und in *buttons* für die Knöpfe.

Die Variable *state* repräsentiert den Zustand des Editors. Wie am Typ der Komponente (*set.T(State)*) zu sehen ist, kann ein Editor mehrere Zustände gleichzeitig haben. In Tabelle 7.2 werden alle zulässigen Zustände erklärt.

Zustand	Bedeutung
new	Während der Initialisierung mit *newEditor*.
exists	Nach Erzeugung mit *create* gesetzt.
sub	Markiert Subeditoren.
visible	Gesetzt, solange sich der Editor auf dem Bildschirm befindet.
windowed	Gesetzt, wenn sich ein Editor direkt in einem Rahmen befindet.

Tabelle 7.2: Zulässige Zustände für Editoren

Die beiden folgenden Komponenten fungieren als Schalter. Mit dem Schalter *key* kann ein Basiseditor als Schlüsselfeld ausgezeichnet werden und der Schalter *modified* gibt an, ob der Wert in der Bildschirmrepräsentation geändert worden ist. Ist der Wert nicht geändert worden, so müssen der Wert und evtl. vorhandene Subwerte bei Aufruf der Methode *apply* nicht zurückgeschrieben werden. Aufgrund von nicht abzuprüfenden Integritätsbedingungen ergeben sich Geschwindigkeitsvorteile. Beim Verlassen eines nicht geänderten Editors kann auf die Sicherheitsnachfrage verzichtet werden. Die *apply*-Methode setzt den Schalter nach erfolgreicher Ausführung zurück.

Mit der letzten Komponente (*parent*) wird die hierarchische Struktur der Editoren aufgebaut. Da Tycoon keine uninitialisierten Variablen zuläßt, muß diese Komponente zunächst mit einem Nullwert initialisiert werden.

Nachdem alle Komponenten des Typs *Hidden* beschrieben worden sind, können nun die weiteren Komponenten des Protokolls beschrieben werden. Die Komponente *label* repräsentiert den Namen des Editors. Er wird für Basiseditoren links neben dem Editor angezeigt. Hierarchische Editoren haben den Namen in der Kopfzeile ihres Rahmens.

Die Komponente *separateWindow* wird bei der Erzeugung des Editors gesetzt. Sie bestimmt, ob der Editor in einem eigenen Rahmen dargestellt werden soll. Dabei ist zunächst eine Unterscheidung zwischen Grund- und Aufklappfenster nicht wichtig. Die Variable bekommt bisher für Verbund-, Varianten- und Iterationseditoren den Wert *true*.

In der Komponente *menuActions* sind die Aktionen vermerkt, die ausgeführt werden sollen, wenn Menüpunkte im Rahmen des Editors angewählt werden. Eine genaue Beschreibung der Menüpunkte ist in Tabelle 7.1 auf Seite 91 zu finden.

Alle weiteren Komponenten des Protokolls sind funktionalen Typs. Sie müssen von allen Editoren ausgeführt werden können. Die funktionalen Komponenten dienen dem Erzeugen

und Löschen von Editoren, dem Zurückschreiben von Werten, dem erneuten Lesen von noch nicht zurückgeschriebenen Werten, dem Anzeigen auf und dem Entfernen von Editoren vom Bildschirm. Weitere funktionale Komponenten ermöglichen den Datenaustausch mit anderen Editoren. Die Funktionen können als Nachrichten aufgefaßt werden, die an die einzelnen Editoren geschickt werden können. In jedem Editor muß deshalb implementiert werden, wie er auf die verschiedenen Nachrichten reagiert.

Die erste Nachricht, die an einen Editor geschickt werden muß, bevor er auf andere Nachrichten reagieren kann, ist *create*. In dieser Methode ist implementiert, wie die Bildschirmrepräsentation auf der NeWS-Seite aufgebaut und initialisiert wird. Die Datenstrukturen zur Verwaltung der Tycoon-Seite werden schon beim Erzeugen des Editors mit *newEditor* angelegt und initialisiert.

Die Nachricht *destroy* bewirkt genau das Gegenteil. Die Bildschirmrepräsentation auf der NeWS-Seite wird gelöscht und alle Informationen der Tycoon-Seite werden entsprechend verändert. Dies gilt auch für Editoren, die direkt in einen Rahmen integriert sind. Weitere Nachrichten kann ein Editor erst nach erneutem Erhalt der Nachricht *create* verarbeiten.

Die folgenden beiden Nachrichten dienen dem Datenaustausch zwischen der Datenstruktur und der Bildschirmrepräsentation. Auf die Nachricht *refresh* muß ein Editor den Wert aus der Variablen der Datenstruktur auslesen und ihn in seine Bildschirmrepräsentation schreiben und auf die Nachricht *apply* muß er den Wert aus seiner Bildschirmrepräsentation auslesen und in die zugrundeliegende Variable der Datenstruktur zurückschreiben.

Zwei weitere Methoden implementieren das Anzeigen und Entfernen des Editors auf bzw. von dem Bildschirm. Die Methode *show* zeigt einen Editor auf dem Bildschirm an bzw. aktiviert seine Bildschirmrepräsentation und die Methode *hide* entfernt einen Editor vom Bildschirm bzw. deaktiviert seine Bildschirmrepräsentation.

Um den Datenaustausch zwischen Editoren verschiedenen Typs zu vereinfachen, ist das nachfolgend beschriebene Protokoll vorgesehen. Jeder Editor kann über die beiden Methoden *getValue* und *setValue* mit anderen Editoren Daten austauschen.

```
Let Rec ValueRep <:Ok =
  Tuple
    case simpleV with value :String
    case executableV with function :Action.T
    case recordV with fields :Iter.T(ValueRep)
    case bulkV with data :Iter.T(ValueRep)
    case multiV with value :String
  end
```

Einfache Editoren (*simpleV*) haben ihren Wert im Feld *value*. Auch Funktionen (*executableV*) werden durch ihren Wert repräsentiert (*function*). Verbunde (*recordV*) werden durch

eine Iteration über ihre Komponenten repräsentiert. Massendaten (*bulkV*) werden über eine Iteration ihrer Werte repräsentiert.

Die letzte im Protokoll der Editoren definierte Nachricht (*toString*) wandelt den Wert einer Bildschirmrepräsentation in einem Editor in eine Zeichenkette um und gibt ihn zurück. Die Funktion findet im Zusammenhang mit dem Datenaustausch zwischen Editoren Verwendung.

Allgemeine Funktionen

Neben dem Protokoll für einen Editor sind in dem Modul *editorRep* auch Funktionen zu finden, die auf alle Editoren angewendet werden können. Das sind z.B. Funktionen, die feststellen, ob ein Editor sichtbar ist, oder die aus einem Editor einen Subeditor machen. Die Konzepte der wichtigsten Funktionen werden nachfolgend beschrieben.

Zu jeder funktionalen Komponente des Protokolls existiert eine korrespondierende Funktion, die den objektorientierten Programmierstil der hierarchisch strukturierten Editoren unterstützt. Die Funktionen können jeweils in zwei Teile aufgeteilt werden. Der eine Teil ist für alle Editoren gleich. In ihm werden die Zustandsinformation ausgewertet bzw. gesetzt. Im anderen Teil wird die Nachricht an den übergebenen Editor propagiert. Die Funktionen werden in der gleichen Reihenfolge beschrieben wie die Komponenten des Protokolls. Grundsätzlich testen alle Funktionen vorher den Zustand des Editors ab und geben, falls notwendig, entsprechende Meldungen aus. In Abbildung 7.6 sind alle möglichen Zustandsübergänge dargestellt. In den Klammern stehen jeweils die zusätzlichen Zustände von Editoren, die direkt in einem Rahmen integriert sind. Sie werden in Abschnitt 7.3.2 noch ausführlich beschrieben.

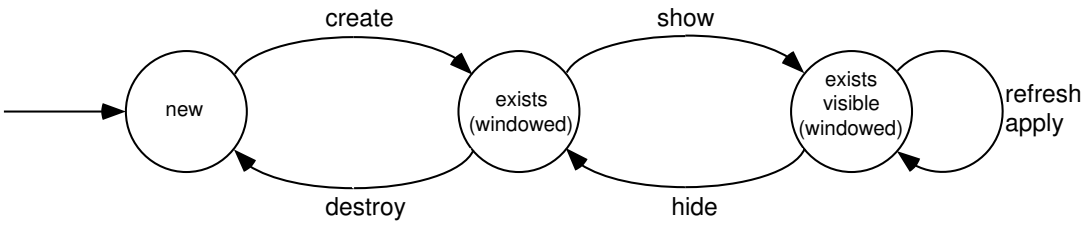


Abbildung 7.6: Mögliche Zustandsübergänge der Editoren

Beim Erzeugen der NeWS-Repräsentation mit *create* wird im Editor der Zustand *new* gelöscht. Als neue Zustandsinformation wird im Editor *exists* gesetzt. Umgekehrt setzt die *destroy*-Funktion den Zustand des Editors wieder auf *new*.

Die *show*-Funktion kann nur schon erzeugte Editoren anzeigen. Aus diesem Grund wird dies vor der Weiterpropagierung abgeprüft. Andersherum kann die *hide*-Funktion nur Editoren vom Bildschirm entfernen, die sichtbar sind. Wird ein Rahmen geschlossen, so sind alle

darin dargestellten Editoren nicht mehr sichtbar. Ein geschlossener, nicht mehr sichtbarer Rahmen wird nicht gelöscht. Er läßt sich wieder öffnen, ohne ihn neu zu erzeugen.

Die Funktion *refresh* dient dem Lesen von Werten aus der zugrundeliegenden Datenstruktur. Die Funktion wird implizit durch das System aufgerufen, wenn ein Rahmen zum ersten Mal geöffnet und wenn ein geschlossener Rahmen wieder geöffnet wird. Die Funktion kann auch explizit vom Anwender aufgerufen werden, um von ihm veränderte, aber noch nicht zurückgeschriebene Werte erneut aus der Datenstruktur zu lesen.

Schreibt ein beliebiger Editor seine Werte in die Variable der zugrundeliegenden Datenstruktur zurück, so können dabei Integritätsbedingungen verletzt werden. In der Funktion *apply* wird das Zurückschreiben deshalb über das **try**-Konstrukt von Tycoon aufgerufen. Tritt beim Zurückschreiben ein Fehler auf, so wird eine Ausnahmebehandlung ausgelöst. Sie gibt den Text des Fehlers in der Fußzeile des Editors aus, in dem der Fehler aufgetreten ist. Aus Geschwindigkeitsgründen arbeitet diese Funktion, wie auch die *refresh*-Funktion, nur auf sichtbaren Editoren.

Neben den Protokollfunktionen ist in diesem Modul eine weitere wichtige Funktion implementiert.

```
let waitAndDispatch(f() :Bool) :Ok =
  while f() do
    wire.flush()
    action.waitAndDispatch()
  end
```

Die Funktion kann als Hauptnachrichtenschleife in Anwenderprogrammen verwendet werden. Der Funktion wird eine boole'sche Funktion als Parameter übergeben. Intern wird in einer Schleife die boole'sche Funktion ausgewertet und auf ankommende Nachrichten von X11/NeWS Server gewartet. Nachrichten werden an ihre Empfänger weitergeleitet. Ist das Ergebnis der übergebenen Funktion nicht mehr *true*, so wird die Schleife und damit auch die Funktion verlassen.

Eine Vielzahl weiterer Funktionen ermöglicht das Abtesten und Erkennen definierter Zustände von Editoren. Da diese Funktionen für das weitere Verständnis nicht von Bedeutung sind, werden sie in diesem Zusammenhang nicht weiter behandelt.

7.3.2 Hilfsmodule

In diesem Abschnitt werden die Hilfsmodule der Editorbibliothek beschrieben. Im einzelnen sind das die Module *newsUtil*, *editorUtil*, *editorMenu*, *editorButtons*, *editorTarget* und *editorWindow*. Die Hilfsmodule bilden die unterste Schicht des in Abbildung 7.5 auf Seite 96 gezeigten Schichtenmodells.

newsUtil

Das Modul *newsUtil* faßt im wesentlichen Sequenzen mit Funktionsaufrufen in Module des *newsenv* zusammen. Dadurch können die Module der Editorbibliothek auf abstrakte Schnittstellenkomponenten zurückgreifen. Zum Beispiel werden ein Grund- und ein Aufklappfenster inklusive der Ereignisbehandlung zur Verfügung gestellt. Die Fenster sind die Grundlage der Rahmen für die Editoren. Daneben sind auch verschiedene Funktionen zur Behandlung von Rollbalken und Rollisten zu finden. Ein Rollbalken wird im Iterationseditor zum Navigieren benötigt und die Rolliste nimmt die Elemente des Tabelleneditors auf.

Erwähnenswert ist noch die durch die Funktion *setChildPos* implementierte intelligente Positionsberechnung für das Anzeigen von Subrahmen. In der Funktion wird in Abhängigkeit von der Position und der Größe des Vaterrahmens die neue Position berechnet. Die neue Position wird so bestimmt, daß der neue Rahmen vollständig auf dem Bildschirm sichtbar ist.

editorUtil

Im Modul *editorUtil* befinden sich Funktionen zum Konvertieren verschiedener Datentypen. Die Funktionen werden für den anwendungsübergreifenden Datenaustausch benötigt.

editorMenu

Das Modul *editorMenu* definiert eine für alle Editoren einheitliche Menüzeile. Wie schon in Abbildung 7.1 auf Seite 90 dargestellt, besteht die Menüzeile aus den beiden Menüpunkten *Edit* und *View*.

Die einzelnen Menüeinträge können beim Erzeugen des Menüs mit Tycoon-Funktionen assoziiert werden. Darüber hinaus ist es möglich, die benannten Menüeinträge einzeln je nach Zustand der Anwendung zu aktivieren bzw. zu deaktivieren.

editorButtons

Die einzige Funktion dieses Moduls erzeugt die beiden Knöpfe, die sich unter dem eigentlichen Editor im Rahmen befinden. Dabei können Name und Funktionalität der Knöpfe bestimmt werden. Der linke Knopf wird als vorausgewählter Knopf eingerichtet. Im allgemeinen wird der linke Knopf zum Speichern der Daten im Editor und der rechte zum Zurückschreiben verwendet.

editorTarget

Das Modul *editorTarget* exportiert Funktionen zur Erzeugung und Manipulation des Datenaustauschfeldes. Das Datenaustauschfeld befindet sich ganz rechts in der Menüzeile des Rahmens in Abbildung 7.1 auf Seite 90. Es dient dem anwendungsübergreifenden Datenaustausch.

Wie schon in Abschnitt 7.1.4 erwähnt, können beim Datenaustausch zwei Arten unterschieden werden. Der Austausch von Daten zwischen Editoren kann dabei auf typisierte Weise erfolgen. Die Daten werden über ein definiertes Protokoll (vgl. Seite 99) zwischen Quell- und

Zieleditor ausgetauscht. Hingegen ist für den Austausch zwischen einem Editor und einem externen Programm eine geeignete Repräsentation zu finden.

Der Austausch von Daten zwischen Editoren erfolgt über die beiden Protokollfunktionen *getValue* und *setValue*. Dabei wird der Wert des Editors in eine austauschfähige Struktur (*ValueRep*) verpackt. Der Zieleditor versucht, die Daten aus der Struktur der Reihe nach auf seine Komponenten zu verteilen.

Für den Austausch zwischen einem Editor und einem externen Programm, z.B. dem OpenWindows-Dateimanager, muß eine geeignete Austauschrepräsentation gefunden werden. Als einfache Repräsentation kommen mit Tabulator oder Komma separierte Listen in Frage. Pro Zeile würde jeweils ein Datensatz ausgegeben werden. Die Werte der Felder sind durch einen Tabulator oder ein Komma voneinander getrennt. In solchen Listen könnten allerdings in den Zeilen nur Basiswerte ausgegeben werden. Für NF^2-Daten müßten geeignetere Repräsentationen gefunden werden.

editorWindow

Das Modul *editorWindow* ist das wichtigste der Hilfsmodule. Es stellt die Rahmen für die Editoren zur Verfügung. Wie schon erwähnt, wird bei den Rahmen zwischen Grund- und Aufklappfenstern unterschieden. Daher gibt es auch zwei Funktionen, die die unterschiedlichen Anforderungen berücksichtigen. Mit dem Rahmen werden auch die Menüzeile, das Datenaustauschfeld und die beiden Knöpfe erzeugt, in den Rahmen eingefügt und initialisiert. Beide Funktionen treffen, wenn sie aufgerufen werden, auf eine der drei folgenden Situationen:

- Der Rahmen existiert noch nicht oder
- er existiert bereits, wird aber momentan nicht angezeigt, oder
- er existiert und wird bereits angezeigt.

Im ersten Fall wird zunächst der einzusetzende Editor mit seinen direkten Komponenten und ein entsprechender Rahmen erzeugt. Anschließend werden die Menüzeile, das Datenaustauschfeld, der Editor und die beiden Knöpfe im Rahmen plaziert. Der fertige Editor kann auf dem Bildschirm angezeigt werden, nachdem sein Zustand auf *windowed* geändert worden ist. Beim Initialisieren eines Subeditors wird zusätzlich der Vater vermerkt.

Im zweiten Fall müssen Editor und Rahmen nicht neu erzeugt werden, da sie noch existieren und momentan nur nicht sichtbar sind. Deswegen wird der Editor nur wieder angezeigt und die Werte werden neu ausgelesen. Aus internen Gründen muß die Ereignisbehandlung neu initialisiert und aktiviert werden.

Der dritte Fall ist der einfachste, da der Rahmen nicht nur existiert, sondern auch angezeigt wird. Daher wird hier nur der Rahmen in den Vordergrund geholt.

Das Modul exportiert neben den beiden Funktionen noch eine feste Positionsangabe. Die Position kann verwendet werden um den Basisrahmen anzuzeigen. Alle weiteren Subrahmen werden relativ zum Vaterrahmen durch die intelligente Positionsberechnung plaziert.

7.3.3 Editoren

Die Implementation der Editoren gestaltet sich unter Benutzung des Protokolls und der im vorherigen Abschnitt beschriebenen Hilfsmodule sehr übersichtlich. Trotzdem werden die einzelnen Editoren in separaten Modulen implementiert. Erst das Modul *editor* (vgl. Abschnitt 7.3.4) bildet eine zusammenfassende Schnittstelle für den Anwender. Nur über diese Schnittstelle kann auf die Editorbibliothek zugegriffen werden. In diesem Abschnitt werden die einzelnen Module der Reihe nach vorgestellt.

Jedes Modul faßt die Implementationen von unterschiedlichen Editoren einer der vier Klassen zusammen. Die einfachsten Editoren finden sich in der Klasse der *Basiseditoren*. Sie entsprechen weitgehend den in Tycoon vorhandenen Basistypen. Die Klasse der *Verbundeditoren* beinhaltet einfache hierarchische Editoren. Sie entsprechen weitgehend den in Tycoon vorhandenen Typkonstruktoren. Die dritte Klasse (*Massendateneditoren*) implementiert Editoren zur Verwaltung von Massendaten.

Wie in Abbildung 7.2 auf Seite 92 dargestellt, kommunizieren die Editoren mit den Programmvariablen über zwei bei der Instanziierung übergebene Funktionen (*get* und *set*). Die *get*-Funktion dient dem Auslesen des Wertes aus der zugrundeliegenden Datenstruktur. Die *set*-Funktion dient dagegen dem Zurückschreiben des Wertes in die Datenstruktur. An dieser Stelle können vom Anwender Integritätsbedingungen formuliert werden. Durch die Instanziierung werden die einzelnen Komponenten des im Abschnitt 7.3.1 beschriebenen Protokolls mit den für den Editor wichtigen Informationen belegt.

7.3.3.1 Basiseditoren

Das Modul *simpleEditors* definiert Funktionen zur Erzeugung von Basiseditoren. Das sind Editoren zur Visualisierung von Werten der Tycoon-Basistypen sowie von Aufzählungstypen und parameterlosen Funktionen ohne Rückgabewert. In Tabelle 7.3 sind alle in diesem Modul implementierten Editoren mit den entsprechenden Typen aufgeführt.

Am Beispiel eines Editors für ganze Zahlen soll die Vorgehensweise bei der Implementation beschrieben werden.

*newInt(l :DisplayItem get() :Int set(:Int) :***Ok***) :T*

An der Signatur ist zu erkennen, daß ein Editor neben seinem Namen (*l*), der in diesem Fall in der Kopfzeile des Rahmens angezeigt wird, zwei funktionale Parameter (*get* und *set*) übergeben bekommt.

let *hidden = editorRep.newHidden()*
let *label = l*
let *separateWindow = false*

Editor	Name	Typ
Ganzzahlen	*newInt*	**var** *Int*
Fließkommazahlen	*newReal*	**var** *Real*
Zeichen	*newChar*	**var** *Char*
Zeichenketten	*newString*	**var** *String*
Boole'sche Werte	*newBool*	**var** *Bool*
Aufzählungen	*newEnum*	**var Tuple case ... end**
Konstanten	*newStringConst*	*String*
Konstanten	*newStringVal*	**var** *String*
Funktionen	*newFun*	***Fun*() :*Ok***
Nullwerte	*newEmpty*	

Tabelle 7.3: Basiseditoren

Die Komponente *hidden* erhält mit der Funktion *newHidden* die Standardvorbelegungen. Der in *l* übergebene Name des Editors wird in der Komponente *label* gespeichert. Alle einfachen Editoren werden in dem Rahmen angezeigt, in dem sie erzeugt worden sind. Aus diesem Grund wird die Komponente *separateWindow* auf *false* gesetzt.

```
let menuActions :MenuActions =
  iter.enum of
    tuple editorMenu.new fun() numericField.setValue(hidden.guiId 0) end
  end
```

Die Menüzeile wird nur mit einer einzigen Funktion, für den Menüeintrag *new*, belegt, d.h. alle anderen Menüeinträge sind nicht selektierbar. Die Funktion hinter dem Menüeintrag *new* setzt den Wert des Editors auf den Grundwert *0*.

Auch die funktionalen Komponenten eines Editors für ganze Zahlen sind auf einfache Weise zu beschreiben.

```
let create() :Ok =
  begin
    hidden.guiId := numericField.create(object.framebuffer)
    textField.setMinimumVisible(hidden.guiId 10)
  end
```

Die *create*-Funktion erzeugt ein auf zehn Zeichen beschränktes numerisches Eingabefeld. Der Identifikator für die erzeugte Schnittstellenkomponente wird in der Komponente *hidden.guiId* vermerkt. Er wird von anderen funktionalen Komponenten noch benötigt.

```
let apply() :Bool =
  begin
    set(numericField.getValue(hidden.guiId))
    true
  end
```

Die Funktion *apply* verwendet die übergebene Funktion *set* zum Zurückschreiben der Werte aus dem Editor in die zugrundeliegende Datenstruktur. Um den Wert aus dem numerischen Eingabefeld zu bekommen, wird die Funktion *getValue* aus dem entsprechenden Modul aufgerufen.

```
let refresh() :Ok =
  numericField.setValue(hidden.guiId get())
```

Die *refresh*-Funktion beschreibt den umgekehrten Weg. Unter Verwendung der übergebenen Funktion *get* wird der Wert aus der Datenstruktur ausgelesen und als Parameter der Funktion *setValue* benutzt. Im Editor wird mit dieser Methode ein Wert gesetzt.

```
let show() :Ok =
  canvas.setVisualState(hidden.guiId layout.active)

let hide() :Ok =
  canvas.setVisualState(hidden.guiId layout.inactive)
```

Die Funktionen *show* und *hide* aktivieren bzw. deaktivieren das numerische Eingabefeld über die Funktion *canvas.setVisualState*.

```
let getValue() :ValueRep =
  tuple case simpleV of ValueRep with
    let value = fmt.int(numericField.getValue(hidden.guiId))
  end
```

Die Komponente *getValue* liest den Wert auf die gleiche Weise aus dem Editor wie die *apply*-Funktion. Die Funktion setzt den gelesenen Wert in die Datenaustauschrepräsentation für einfache Editoren (*simpleV*) ein. Dazu wird der Wert in eine Zeichenkette konvertiert.

```
let setValue(v :ValueRep) :Ok =
  case v
  when simpleV with v then
    numericField.setValue(hidden.guiId editorUtil.stringToInt(v.value))
  else
    ok
  end
```

Den umgekehrten Weg geht die *setValue*-Funktion. Sie testet die Datenaustauschrepräsentation auf *simpleV* und schreibt im Erfolgsfall den umgewandelten Wert in den Editor.

```
let toString() :Iter.T(String) =
  iter.singleton(fmt.int(get()))
```

Die letzte Komponente gibt den Wert des Editors als Iteration mit einem Element zurück.

Die Instanziierungen für die anderen einfachen Editoren laufen alle in der gleichen Weise ab. Lediglich der Aufzählungseditor (*newEnum*), der Funktionseditor (*newFunction*) und der leere Editor (*newEmpty*) haben eine von den anderen einfachen Editoren abweichende Signatur.

Einem Aufzählungseditor werden zusätzlich zu den beiden Funktionen *get* und *set* eine Ordnungsfunktion (*ord*) und eine Funktion zum Erzeugen einer Iteration von Paaren (*pairs*) übergeben.

```
newEnum(O <:Ok  l :DisplayItem  get() :O  set(:O) :Ok
    ord(:O) :Int  pairs() :Iter.T(Tuple :O :String end)) :T
```

Ohne die beiden zusätzlichen Parameter wäre es für die beiden Funktionen *set* und *get* nicht möglich, Werte zu lesen oder zu schreiben. Die Funktion *ord* gibt für einen Wert eines Typs O die Ordinalität zurück. Die Funktion *pairs* erzeugt eine Iteration von Paaren über alle Werte des Typs O. Ein Paar besteht jeweils aus einem Wert des Typs O und einer Zeichenkette. Mit Hilfe der Funktion können Werte eines beliebig komplexen Typs angezeigt werden. Der Anwender ist verantwortlich dafür, daß aus einem Wert des Typs O eine sinnvolle Zeichenkette gebildet wird. Dieses Konzept ermöglicht es, aus Tupeln nur bestimmte Komponenten für die Bildung der Zeichenkette zu benutzen. Nach der Beschreibung der beiden Funktionen *ord* und *pairs* soll nun die Vorgehensweise der Funktionen *apply* und *refresh* beschrieben werden. In der Funktion *apply* wird die erste Zeichenkette in der Iteration gesucht, die der ausgewählten entspricht. Der in diesem Paar vermerkte Wert des Typs O wird in die Datenstruktur zurückgeschrieben. In der Funktion *refresh* wird die übergebene Funktion *ord* benötigt. Mit Hilfe dieser Funktion wird in der Iteration der Wert gesucht, der dem Inhalt der Datenstruktur an dieser Stelle entspricht. Die diesem Wert zugeordnete Zeichenkette wird im Editor ausgegeben.

Die Signatur eines Editors zur Ausführung von Funktionen hat die folgende Form:

```
newFunction(l :DisplayItem  name :DisplayItem  f :Action.T) :T
```

Wie an der Signatur zu sehen ist, werden einem Funktionseditor keine *set* und *get*-Funktionen übergeben. Neben dem Namen des Editors beschreiben die weiteren Parameter den Namen (*name*), der auf dem Knopf zum Anstoßen der Funktion steht, sowie die anzustoßende Funktion (f) selbst. Der interne Aufbau des Editors gestaltet sich aus diesem Grund auch sehr einfach, denn die *apply*- und die *refresh*-Funktionen entfallen.

Noch einfacher ist der Aufbau des leeren Editors. Als Parameter wird nur der Name des Editors übergeben.

newEmpty(l :DisplayItem) :T

Bevor im weiteren auf die komplexeren Editoren eingegangen wird, sollen hier noch einige Bemerkungen zu zwei einfachen Editoren aufgeführt werden. Werte in den beiden Editoren *newStringVal* und *newStringConst* sind nicht veränderbar. Dabei wird dem *newStringVal*-Editor nur eine *get*-Funktion übergeben. Die Werte werden hier in der gleichen Weise erzeugt wie bei einem *newString*-Editor. Dem *newStringConst*-Editor wird bei der Instanziierung hingegen eine Zeichenkettenkonstante übergeben, die für alle Werte einer Datenstruktur gleich ist.

7.3.3.2 Verbundeditoren

Verbundeditoren sind einfache hierarchische Editoren. Die beiden im Modul *recordEditors* implementierten Editoren entsprechen zwei in Tycoon vorhandenen Typkonstruktoren (vgl. Tabelle 7.4).

Editor	**Name**	**Typ**
Verbunde	*newRecord*	***Tuple ... end***
Varianten	*newVariant*	***Tuple case ... end***

Tabelle 7.4: Verbundeditoren

Als erster hierarchischer Editor wird der Verbundeditor (*newRecord*) vorgestellt. Ein solcher Editor kann seinerseits einfache, strukturierte und multimediale Editoren enthalten. Diese werden in einem Feld bei der Instanziierung mit übergeben. Die Signatur eines Verbundeditors hat die folgende Form:

newRecord(l :DisplayItem.T fields :Iter.T(Editor.T)) :Editor.T

Wie an der Signatur zu sehen ist, erwartet der Verbundeditor eine Iteration von Editoren beliebiger Typen. Einem Verbundeditor müssen keine *get*- und *set*-Funktionen übergeben werden, da alle in einem Verbund integrierten Editoren diese Funktionen selbst beinhalten.

Das Protokoll eines solchen Editors wird wie folgt definiert: die Komponente *hidden* wird genau wie bei den einfachen Editoren mit dem Ergebnis der Funktion *newHidden* vorbelegt. Da ein geöffneter Verbundeditor in einem eigenen Fenster dargestellt werden soll, wird die Komponente *separateWindow* anders als bei den einfachen Editoren belegt.

Komplexer gestaltet sich die Implementation der *create*-Funktion. Ein Verbundeditor wird durch einen speziellen Behälter (*calculated panel*) visualisiert. Die einzelnen in der Iteration übergebenen Editoren müssen an geeigneten Stellen im Behälter plaziert werden. Wie schon früher erwähnt, werden die Namen der Editoren rechtsbündig und die Editoren selbst linksbündig in den Behälter eingefügt. Bei den Editoren muß zusätzlich zwischen einfachen und hierarchischen Editoren unterschieden werden. Für jeden einfachen Editor wird nach seinem Namen die durch die jeweilige *guiId* identifizierte Bildschirmrepräsentation in den Behälter eingefügt. Für hierarchische Editoren wird nach dem Namen statt der Bildschirmrepräsentation des Subeditors ein Knopf in den Behälter eingefügt. An den Knopf ist eine Aktion gebunden, die, wenn der Knopf gedrückt wird, einen Subrahmen öffnet, in dem der hierarchische Subeditor dargestellt wird.

Die anderen funktionalen Komponenten des Verbundeditors propagieren die entsprechende Nachricht an die in den Verbund integrierten Editoren.

Der zweite in diesem Modul implementierte Editor ist der Varianteneditor. Ein solcher Editor kann als Kombination von einem Aufzählungs- und einem Verbundeditor begriffen werden. Diese Tatsache ist auch an der Signatur des Varianteneditors zu erkennen.

*newVariant(O <:***Ok** *l :DisplayItem get() :O set(:O) :***Ok**
*ord(:O) :Int variants :Iter.T(***Tuple** *:O :T* **end**)) *:T*

Wie an der Signatur zu sehen ist, erwartet der Varianteneditor im Gegensatz zum Verbundeditor wieder eine *get*- und *set*-Funktion. Daneben werden eine Ordnungsfunktion (*ord*) und eine Funktion zum Erzeugen einer Iteration von Paaren (*variants*) übergeben. Die Funktion *ord* berechnet die Ordinalität der Varianten. Die Funktion *variants* erzeugt eine Iteration von Paaren aus Variante und dazugehörigem Editor.

Varianteneditoren werden wie Verbundeditoren durch einen speziellen Behälter repräsentiert. In den Behälter werden ein Abkürzungsmenüknopf zum Auswählen der Variante und der zu der Variante gehörende Editor eingefügt. Erschwerend kommt hinzu, daß die Variante und damit auch der Editor dynamisch geändert werden können. Aus diesem Grund ist beim Varianteneditor auch die *refresh*-Funktion komplexer.

Die intern zum Ändern der Variante verwendete Funktion *change* und die *refresh*-Funktion machen intensiven Gebrauch von den übergebenen Parametern. Während die *change*-Funktion die neue Variante durch die interaktive Auswahl des Anwenders erhält, bekommt die *refresh*-Funktion die neue Variante durch die *get*-Funktion. Beide Funktionen müssen den zur alten Variante gehörenden Editor aus dem Behälter entfernen, den neuen Editor einfügen und, da der neue Editor mehr oder weniger Komponenten als der alte enthalten kann, die Größe des Behälters neu bestimmen.

Die anderen funktionalen Komponenten des Varianteneditors propagieren die entsprechende Nachricht an den zur aktuellen Variante gehörenden Editor.

7.3.3.3 Massendateneditoren

In diesem Abschnitt werden Editoren beschrieben, die das Anzeigen von Massendaten ermöglichen. Alle Module, die einen Kollektionstyp exportieren, wie z.B. *set* oder *list*, verfügen über eine *elements*-Funktion. Mit Hilfe dieser Funktion ist es möglich, beliebige Kollektionen auf die von den Massendateneditoren verwendete Datenstruktur *LinkedList* abzubilden. Diese Datenstruktur eignet sich besonders deswegen als Grundlage für die Datenvisualisierung, weil ihre Elemente veränderbar und an eine Position gebunden sind. Mit den Massendateneditoren kann daher eine große Bandbreite von Datenstrukturen visualisiert werden. Wie in Tabelle 7.5 zu sehen ist, gibt es zwei Massendateneditoren. Einen Iterationseditor zum satzweisen direkten Editieren von Massendaten und einen Tabelleneditor zum gleichzeitigen Anzeigen von mehreren Datensätzen.

Editor	**Name**	**Typ**
Iterationen	*newIter*	*LinkedList.T(...)*
Tabellen	*newTable*	*LinkedList.T(...)*

Tabelle 7.5: Massendateneditoren

Die Signaturen der beiden Massendateneditoren unterscheiden sich nicht. Sie haben die folgende Form:

*newIter, newTable(E <:***Ok** *l :DisplayItem get() :LinkedList.T(E)*
newElement() :E elementEditor(:Cursor(E)) :T) :T

Die beiden Funktionen *get* und *set* haben die gleiche Aufgabe wie bei den einfachen Editoren. Der Anwender muß für das Einfügen von Elementen in die Datenstruktur eine Funktion übergeben, die ein neues Element erzeugt (*newElement*). Dies ist notwendig, da in Tycoon keine uninitialisierten Variablen möglich sind. Als Alternative wäre es denkbar, statt der Funktion einfach ein Element zu übergeben. Dies ist aber nicht möglich, da in diesem Fall alle neuen Elemente eine Referenz auf dieses Element enthielten. Änderungen an einem neuen Element wären in allen neuen Elementen sichtbar. Als letztes erwartet dieser Editor einen Subeditor, der zur Änderung eines Elements der Iteration dient. Die Elemente werden über einen *Cursor* verändert. Der Cursor ist als Tupel mit einer modifizierbaren Komponente implementiert.

Let *Cursor(E <:***Ok***) <:***Ok** *=*
Tuple var *x :E* ***end***

Zusätzlich müssen die Daten aus der Datenstruktur im Editor gehalten werden, denn sie werden erst nach einer *apply*-Nachricht zurückgeschrieben. Intern werden die Elemente in einer von dem Modul *linkedList* angebotenen Struktur gehalten.

Das Protokoll eines solchen Editors wird wie folgt definiert: die Komponente *hidden* wird genau wie bei den einfachen Editoren mit dem Ergebnis der Funktion *newHidden* vorbelegt. Da ein geöffneter Iterationseditor in einem eigenen Fenster dargestellt werden soll, wird die Komponente *separateWindow* anders als bei den einfachen Editoren belegt.

Die Implementation der *create*-Funktion ist wie bei allen hierarchischen Editoren komplexer als bei den Basiseditoren. Ein Iterationseditor wird durch einen speziellen Behälter (*border bag*) repräsentiert. Ein solcher Behälter kann bis zu fünf Klienten an fest vordefinierten Positionen aufnehmen. Der zur Navigation benötigte Rollbalken wird oben in den Behälter eingefügt. In die Mitte des Behälters wird der Elementeditor eingefügt.

Auch die *refresh*-Funktion beinhaltet wieder zusätzliche Logik, denn es muß sichergestellt werden, daß nur auf den vorhandenen Elementen der Iteration navigiert wird. Die Funktion muß zusätzlich auch auf eine leere Iteration richtig reagieren können. Ist die Iteration nicht leer, so wird der Cursor mit dem neuen Element gefüllt und der Wert auf dem Bildschirm ausgegeben. Alle anderen funktionalen Komponenten propagieren die Nachrichten an den Elementeditor weiter.

Auf einem Iterationseditor sind eine Vielzahl von Funktionen ausführbar. Die Funktionen können durch Auswählen der entsprechenden Menüeinträge angestoßen werden. Zum Beispiel kann mit *New* ein neues Element an der aktuellen Position eingefügt oder durch *Cut All* alle Elemente der Iteration gelöscht werden.

Mit einem Tabelleneditor ist es möglich, mehrere Datensätze gleichzeitig anzuzeigen. Aus diesem Grund ist er auch anders aufgebaut. Das zentrale graphische Element des Editors ist eine Tabelle. Die einzelnen Felder werden in Form von Spalten in der Tabelle dargestellt. Rechts neben der Tabelle befindet sich ein Rollbalken. Er ermöglicht die Navigation in der Tabelle.

Das eigentliche Problem beim Tabelleneditor ist die Übergabe der Datensätze an die Tabelle. Dafür stehen grundsätzlich zwei Strategien zur Verfügung. Zum einen können alle Datensätze zu Beginn an die Tabelle übergeben werden. In diesem Fall kann die gesamte Behandlung der Navigation von der Tabelle selbst übernommen werden. Die Übergabe von sehr vielen Datensätzen an die Tabelle kann jedoch viel Zeit in Anspruch nehmen. Deswegen ist von dieser Vorgehensweise abzuraten.

Die zweite Strategie sieht vor, immer nur so viele Datensätze an die Tabelle zu übergeben, wie gleichzeitig sichtbar sind. In diesem Fall muß die Navigation durch eine optimierende Funktion selbst implementiert werden. Der Vorteil ist jedoch der wesentlich geringere Anfangsaufwand. Genau wie beim Iterationseditor muß auch hier zwischen einer leeren und einer gefüllten Iteration unterschieden werden. Eine leere Iteration wird durch eine leere Tabelle dargestellt. Bei einer gefüllten Iteration gibt es fünf unterschiedliche Navigationsmöglichkeiten

Wenn eine Zeile abwärts navigiert wird, dann wird ein neues Element unten angehängt und das oben herausfallende Element wird gelöscht. Wenn mehrere Zeilen abwärts navigiert wird, werden abwechselnd so viele neue Elemente unten angehängt und oben gelöscht werden, wie

die Sicht verschoben worden ist. Entsprechendes gilt für die beiden Fälle beim aufwärts Navigieren. Optimierend wird nur dann eingegriffen, wenn um mehr Zeilen navigiert wird, als in der Tabelle gleichzeitig sichtbar sind. In diesem Fall werden alle Elemente der Tabelle gelöscht. Anschließend werden so viele Elemente ab der aktuellen Position in die Tabelle eingefügt, wie maximal gleichzeitig sichtbar sind. Dieser Fall tritt auch auf, wenn noch gar keine Elemente in der Tabelle sind.

In einem Tabelleneditor können die Datensätze nicht direkt verändert werden. Um einen Datensatz zu verändern, kann durch Selektion einer Zeile ein Elementeditor geöffnet werden. Nach der Selektion wird das ausgewählte Element in den Cursor geladen und der Editor mit diesem Element geöffnet. Als Problem ergibt sich, daß die Bildschirmrepräsentation des Tabelleneditors geändert werden muß, nachdem eine sichtbare Zeile geändert worden ist.

7.3.4 Schnittstelle zum Anwendungsprogramm

Das Modul *editor* dient als Schnittstelle zum Anwender der Bibliothek. Nur über dieses Modul ist der Zugriff auf die Editorbibliothek möglich. Es reicht die in den anderen Modulen definierten Editoren zum Anwender durch. Dazu importiert es die Module *simpleEditors*, *recordEditors* und *bulkEditors*. Auch das Protokoll wird über dieses Modul dem Anwender zugänglich gemacht. Die Funktion zum Anzeigen von Editoren wird vom Modul *editorWindow* importiert und zum Anwender exportiert.

7.4 Benutzerdefinierte Editoren

Der Anwender der Editorbibliothek hat die Möglichkeit, eigene Editoren in die Umgebung zu integrieren. Dazu ist es lediglich notwendig, alle vom Protokoll (vgl. Abschnitt 7.3.1) vorgegebenen Anforderungen zu erfüllen. In diesem Abschnitt wird die Vorgehensweise bei der Implementation eines Editors zum Abspielen von Audiodaten beschrieben.

In der Tycoon-Bibliothek *machineenv* befindet sich das Modul *audio* [Mathiske et al. 93]. Dieses Modul ermöglicht Tycoon-Programmen das Aufnehmen, Editieren und Abspielen von Audiodaten. Dazu sind in dem Modul neben dem abstrakten Datentyp *audio.T* verschiedene Funktionen implementiert, die auf Werten dieses Typs arbeiten.

Der Editor zum Abspielen von Audiodaten wird an der Benutzerschnittstelle durch einen Knopf repräsentiert. Wird der Knopf gedrückt, so werden die Daten an den Audioausgang übergeben und dort abgespielt.

Die Signatur des Editors hat die folgende Form:

newAudio(l :DisplayItem get() :audio.T) :Editor.T

Die Implementation gestaltet sich sehr einfach. Importiert werden muß neben anwendungsspezifischen Modulen nur das Modul *editor*. Alle zur Implementation benötigten Typen und Funktionen werden von diesem Modul exportiert. Anschließend können der Reihe nach alle durch das Protokoll der Editoren definierten Komponenten und Funktionen implementiert werden. Zum Beispiel wird durch die *create*-Funktion ein Knopf erzeugt. Der Knopf wird mit der folgenden Tycoon-Funktion assoziiert:

***let** play() :**Ok** = audio.play(get())*

Durch Drücken des Knopfes werden die Audiodaten über einen Audioausgang abgespielt. Da der Wert in der derzeitigen Version nicht änderbar ist, entfällt die Implementation der *apply*-Funktion. Die anderen Funktionen des Protokolls können analog zu denen des Editors für ganze Zahlen definiert werden (vgl. Abschnitt 7.3.3.1).

8. Bewertung und Ausblick

Dieses Kapitel faßt alle in dieser Arbeit gewonnenen Ergebnisse zusammen. Dazu wird in einem ersten Abschnitt die Vorgehensweise bei der Entwicklung der Editorbibliothek unter software-ergonomischen und -technischen Gesichtspunkten beschrieben. Anschließend werden die beim Vergleich der Visualisierungskomponenten bestehender Systeme in Kapitel 4 gewonnenen Erkenntnisse, die als Grundlage für die Implementation der Bibliothek dienten, mit den in der Implementation tatsächlich erreichten Zielen verglichen. Mit einer kurzen Übersicht über offene Probleme und mögliche Lösungsalternativen endet diese Arbeit.

8.1 Zusammenfassung

In dieser Arbeit sind alle für die typsichere generische Datenvisualisierung in Tycoon notwendigen Dienste ausführlich behandelt worden. Begonnen wurde in Kapitel 2 mit der Suche nach einem geeigneten Ansatz zur dynamischen Erzeugung von graphischen Benutzerschnittstellen aus einer Programmiersprache heraus. Mit Klassenbibliotheken wurde ein geeigneter Ansatz gefunden. Anschließend wurden in Kapitel 3 die Vorteile der Integration eines solchen Dienstes in die Tycoon-Entwicklungsumgebung gegenüber einer Integration in andere Umgebungen und Programmiersprachen vorgestellt.

Die Analyse verschiedener neuerer kommerzieller Systeme und Forschungssysteme an einem praktischen Anwendungsbeispiel hinsichtlich ihrer Möglichkeiten zur Datenvisualisierung in Kapitel 4 ergaben Anforderungen an und Ideen für die Bibliothek der typsicheren generischen Editoren in Tycoon. Die Ergebnisse dieser Untersuchung haben sowohl die Gestaltung der graphischen Benutzerschnittstelle der Editoren als auch die Funktionalität der erstellten Bibliotheken beeinflußt.

Schließlich wurden in Kapitel 5 die für die Implementation in der vorliegenden Anwendungsumgebung benutzten Dienste und Werkzeuge vorgestellt. In dieser Umgebung wurde im Rahmen dieser Arbeit zunächst eine Anbindung des NeWS Toolkits an das Tycoon-System in Form einer Bibliothek vorgenommen. Die Bibliothek ist eine Voraussetzung für die Implementation der typsicheren generischen Editoren, da hierbei vielfältig von der Möglichkeit Gebrauch gemacht wurde, typsicher dynamisch Schnittstellenkomponenten zu erzeugen und zu manipulieren.

8.2 Bewertung

In diesem Abschnitt werden die gestellten Anforderungen (vgl. Abschnitt 4.3) an die Datenvisualisierung mit den im Rahmen dieser Arbeit erarbeiteten Lösungen verglichen.

Graphische Benutzerschnittstellen

Die in Kapitel 6 vorgestellte Schnittstelle zum NeWS Toolkit (*newsenv*) ermöglicht die Erzeugung beliebiger graphischer Benutzerschnittstellen für Tycoon-Programme. Zusätzlich können mit dem Devguide interaktiv erzeugte Benutzerschnittstellen angebunden werden.

Die Benutzung von Funktionen aus dieser Bibliothek hat viele Vorteile für die Erstellung von graphischen Benutzerschnittstellen. Die Funktionen des *newsenv* gestatten es, eine Typisierung der Eingabeparameter durchzuführen, von der kellerspeicher-orientierten Syntax der NeWS-Methodenaufrufe zu abstrahieren und die sonst aufwendig zu programmierende Kommunikation, die zwischen einer Tycoon-Anwendung und dem X11/NeWS Server stattfindet, vor dem Anwender der Bibliothek zu verbergen.

Typsichere generische Datenvisualisierung

Die vorliegende Implementation der Editorbibliothek ist das Ergebnis eines iterativen Entwicklungsprozesses. Aufgrund von Erkenntnissen der Software-Ergonomie und -Technik hat eine iterative Vorgehensweise entscheidende Vorteile (vgl. z.B. [Oberquelle 92; Budde et al. 92]).

Eine erste Version der Bibliothek [Kirch, Müßig 92] wurde in unterschiedlichen Projekten und Demonstrationen produktiv eingesetzt. Die Projekte waren alle im Bereich der Entwicklung und Nutzung datenintensiver Anwendungen angesiedelt. Verwendet wurden jeweils verschiedene Editoren zur Visualisierung bzw. Eingabe von Daten.

Einige Probleme, die sich bei der Anwendung in den Projekten ergeben haben, resultierten aus dem Anspruch der allgemeinen Verwendbarkeit der Bibliothek. Die speziellen Bedürfnisse der Anwender konnten so nur zum Teil befriedigt werden. Die folgende Übersicht stellt einige der aufgetretenen Probleme dar.

- Es wurde gewünscht, Editoren für Schlüsselfelder an der Benutzerschnittstelle kenntlich zu machen.
- Ein Kritikpunkt war die unklare Funktionalität beim Verlassen der Editoren. Intern wird beim Verlassen der Editoren versucht, den Wert zu speichern. Dadurch werden auch vorhandene Integritätsbedingungen geprüft. Wurde der Wert nur angezeigt und nicht verändert, so ist eine solche Vorgehensweise überflüssig und langsam.
- Das Tycoon-System erlaubt ein Arbeiten über mehrere Benutzersitzungen. Unterstützt wurde das von der Bibliothek jedoch nicht. In einer neuen Sitzung mußten alle Editoren neu erzeugt werden.

- ▷ Editoren zum Visualisieren von Varianten, Funktionen und Tabellen fehlten. Generell kam der Wunsch nach multimedialen Editoren, wie z.B. Audio-, Bild- und Texteditoren, auf.

Aus den oben genannten Gründen und durch die Analyse anderer Systeme beeinflußt, wurde bei der Neuimplementation nicht nur der Aufbau und die Zusammensetzung der Benutzerschnittstelle geändert, sondern auch das Protokoll der Editoren um einige Komponenten ergänzt. Die Neuimplementation entspricht der in Kapitel 7 vorgestellten Bibliothek.

Die Neuimplementation hat einige der genannten Probleme beseitigt. Zum Beispiel können mit Hilfe der neuen Komponente *create* im Protokoll die Editoren jetzt in einer neuen Benutzersitzung automatisch wieder aufgebaut werden.

Durch die Trennung der beiden Knöpfe *Reset* und *Apply* vom Verbundeditor und Trennung der Menüs vom Iterationseditor ist es nun möglich, auch einfachen Editoren diese Funktionalität zur Verfügung zu stellen. Insgesamt ist es durch diese Trennungen gelungen, die Funktionalität der Editoren auf das wesentliche zu beschränken. Dadurch ist eine einfachere Integration neuer Editoren in die Bibliothek möglich.

Mit der in Kapitel 7 vorgestellten Bibliothek *editenv* ist der typsichere Aufbau von satzorientierten Bildschirmmasken durch Bibliotheksaufrufe möglich. Darüber hinaus ermöglicht die Bibliothek die generische Visualisierung von Kollektionen.

Der Datenaustausch zwischen verschiedenen Editoren und zwischen einem Editor und einer anderen Anwendung ist nicht vollständig implementiert. Einige Ideen hierfür sind in Abschnitt 7.3.2 vorgestellt worden. In der gegenwärtigen Implementation wird nur der Datenaustausch zwischen Basiseditoren unterstützt.

Abschließend kann festgestellt werden, daß die von TL angebotenen Funktionen höherer Ordnung, der parametrische Polymorphismus und das reiche Typsystem eine substantielle Vereinfachung der repetitiven Programmieraufgaben, wie das Erstellen von Funktionen zur formatierten und sicheren Dateneingabe sowie zur Ausgabe von Datenobjekten auf unterschiedlichen Ausgabemedien und für unterschiedliche Benutzerklassen, ermöglichen.

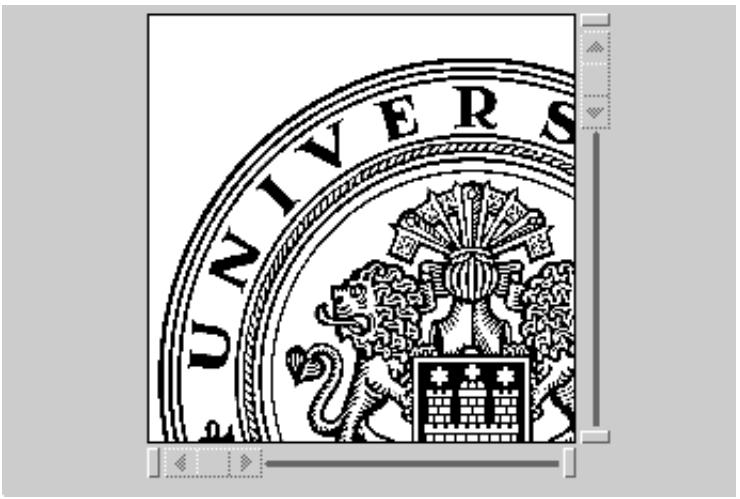
8.3 Ausblick

Nachfolgend werden kurz einige noch offene Punkte bezüglich der Funktionalität der vorhandenen und der Definition neuer Editoren vorgestellt. Im Hinblick auf die Möglichkeit, multimediale Daten im Objektspeicher von Tycoon halten zu können, werden zunächst zwei solche Editoren mit ihrer Signatur und graphischen Benutzerschnittstelle vorgestellt.

Bildeditor

In der Bibliothek *machineenv* steht ein Modul (*image*) zum Anzeigen von Bildern zur Verfügung. [Mathiske et al. 93]. Auf dieser Grundlage kann ein Editor zum Anzeigen von

Bildern implementiert werden. Die Benutzerschnittstelle ist durch einen umrahmten Bereich mit zwei Rollbalken geprägt. Die Rollbalken ermöglichen das vertikale und horizontale Verschieben des sichtbaren Bildausschnittes.



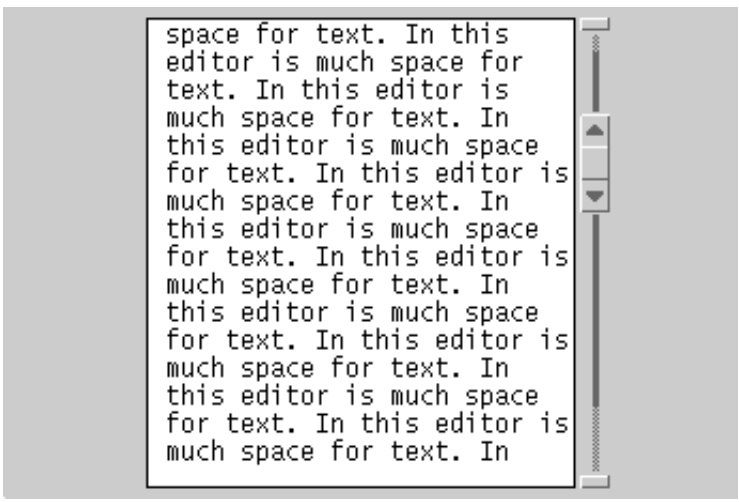
Bei Bildeditoren bietet es sich an, genau wie für Verbund- und Massendateneditoren zwei Bildschirmrepräsentationen zu verwenden. Die nachfolgende Visualisierung wird benutzt, wenn der Editor als Komponente in einem anderen Editor eingeschachtelt ist.

Nach einem Druck auf den Knopf *Display...* öffnet sich ein weiteres Fenster, in dem der eigentliche Bildeditor dargestellt wird. Die Signatur ergibt sich analog zu den anderen Editoren.

*newImage(l :DisplayItem get() :image.T set(:image.T) :**Ok**) :T*

Texteditor

Texte können durch die in Abschnitt 6.1.5 vorgestellten Jot-Texteditoren repräsentiert werden. In diesem Fall stehen die vielfältigen Manipulationsmöglichkeiten der Jot-Texteditoren für den Text zur Verfügung.



Die Signatur des Texteditors entspricht der Signatur eines Zeichenketteneditors (vgl. Abschnitt 7.3.3.1). Bezüglich der Funktionalität unterscheiden sich beide Editoren jedoch erheblich voneinander (vgl. [SunSoft 92b])

*newText(l :DisplayItem get() :String set(:String) :**Ok**) :T*

Mehrfaches Anzeigen

Probleme ergeben sich aus dem mehrfachen Anzeigen (*sharing*) von Werten. Problematisch ist dabei die geeignete Repräsentation solcher Situationen für Anwender sowie die Reihenfolge der Validierung, wenn der Anwender den *Apply*-Knopf drückt.

Dynamische Visualisierung

Besitzt eine Datenstruktur keine Integritätsbedingungen auf ihren Attributen und entstammen die Typen aller Attribute einer bekannten Menge von Basistypen, so ließe sich die Generierung von Editoren durch eine dynamische Typanalyse (**typecase**) noch weiter automatisieren, wenn die im Tycoon-System vorgesehene Funktionalität verfügbar wäre. In diesem Fall könnten beliebige Werte durch eine einzige Funktion mit der folgenden Signatur aufgerufen werden.

*newDyn(**Dyn** E <:**Ok** l :DisplayItem **var** e :E) :T*

Im Gegensatz zu den anderen Funktionen, die Editoren erzeugen, wird dieser Funktion nur die Variable übergeben, deren Werte angezeigt werden sollen. Intern wird mit Hilfe der dynamischen Typanalyse von Tycoon der dem Typ dieser Variable entsprechende Editor automatisch aufgebaut. Aus diesem Grund ist es auch nicht möglich, Integritätsbedingungen auf den Komponenten zu überprüfen.

Portabilität

Die Unterstützung verschiedener Hard- und Softwareplattformen kann durch die Verwendung des NeWS Toolkits nicht gewährleistet werden. Zu Beginn der Arbeit an dieser Diplomarbeit war es nicht absehbar, daß Sun Microsystems die Unterstützung des Werkzeuges mit Erscheinen von Solaris 2.3 einstellt. Vom Konzept her betrachtet ist das NeWS dem X11 deutlich überlegen. Insofern war es die richtige Entscheidung, das NeWS Toolkit zu verwenden.

Zur Zeit wird geplant, das Tycoon-System auch auf andere Plattformen zu portieren. Daher ist es langfristig von Vorteil, eine systemübergreifende Klassenbibliothek, wie sie z.B. das StarView-Toolkit [Busch et al. 93] bietet, zur Erzeugung von graphischen Benutzerschnittstellen zu verwenden.

A. Programmbeispiel

In diesem Kapitel sind die Implementationen eines Programmbeispiels in Tycoon, DBPL, INGRES/Windows 4GL und O_2 zu finden. Sie werden verwendet um die Visualisierungskomponente der im Rahmen dieser Arbeit untersuchten Systeme zu analysieren (vgl. Kapitel 4). In den Programmbeispielen werden verschiedene Typen und Werte sowie Visualisierungsfunktionen aus dem Bereich der Adressverwaltung definiert.

A.1 Tycoon

```
module world
import editor :Editor  editorRep  iter :Iter  linkedList :LinkedList  action :Action  poly
export
  Let FamilyStatus = Tuple case single, married, divorced end

  Let Address = Tuple
    street :String
    zip :String
    city :String
    case private with
      rooms :Int
    case office with
      country :String
  end

  Let Person = Tuple
    name :String
    fstName :String
    var age :Int
    address :Address
    familyStatus :FamilyStatus
    selected :Bool
  end
```

```
let persons = linkedList.new(:Person)

let newAddress() :Address = tuple case private of Address with "" "" "" 0 end
let newSingle() = tuple case single of FamilyStatus end
let newPerson() :Person = tuple "" "" var 0 newAddress() newSingle() false end

let familyStatusOrd(fs :FamilyStatus) :Int = poly.ordinal(fs)
let addressOrd(a :Address) :Int = poly.ordinal(a)

let private = tuple case private of Address "" "" "" 0 end
let office = tuple case office of Address "" "" "" "" end

let familyStatusPairs() :Iter.T(Tuple :FamilyStatus :String end) =
  iter.enum of
    tuple tuple case single of FamilyStatus end "Single" end
    tuple tuple case married of FamilyStatus end "Married" end
    tuple tuple case divorced of FamilyStatus end "Divorced" end
  end

let privateAddressEditor(a :Address) :Editor.T =
  editor.newRecord("Private Address"
    iter.enum of
      editor.newString("Street" fun() a.street fun(new :String) ok)
      editor.newString("Zip" fun() a.zip fun(new :String) ok)
      editor.newString("City" fun() a.city fun(new :String) ok)
      editor.newInt("Rooms" fun() a!private.rooms fun(new :Int) ok)
    end)

let officeAddressEditor(a :Address) :Editor.T =
  editor.newRecord("Office Address"
    iter.enum of
      editor.newString("Street" fun() a.street fun(new :String) ok)
      editor.newString("Zip" fun() a.zip fun(new :String) ok)
      editor.newString("City" fun() a.city fun(new :String) ok)
      editor.newString("Country" fun() a!office.country fun(new :String) ok)
    end)
```

```
  let addressEditor(a :Address) :Editor.T =
    editor.newVariant("Address"
      fun() a  fun(new :Address) ok  addressOrd
      iter.enum of
        tuple private  privateAddressEditor(a) end
        tuple office  officeAddressEditor(a)  end
      end)

  let personEditor(p :Editor.Cursor(Person)) :Editor.T =
    editor.newRecord("Person"
      iter.enum of
        editor.newString("Name" fun() p.x.name fun(new :String) ok)
        editor.newString("Fst. Name" fun() p.x.fstname fun(new :String) ok)
        editor.newInt("Age" fun() p.x.age fun(new :Int)
          if new < 0 then
            raise editor.illegal with "Age to low!" end
          else
            p.x.age := new
          end)
        editor.newEnum("Family Status" fun() p.x.familyStatus
          fun(new :FamilyStatus) ok familyStatusOrd familyStatusPairs)
        addressEditor(p.x.address)
        editor.newBool("Selected" fun() p.x.selected fun(new :Bool) ok)
      end)

  let personsEditor(persons :LinkedList.T(Person)) :Editor.T =
    editor.newIter("Title" fun() persons newPerson personEditor)

  let go() :Ok =
    begin
      let var continue = true
      let quit :Action.T = fun() continue := false

      let pEdit = personsEditor(persons)

      editor.display(pEdit quit editor.somewhere)
      editor.waitAndDispatch(fun() continue)
    end

end;
```

A.2 DBPL

```
PROCEDURE DisplayForm(persons :Persons; VAR person :Person);
VAR
  form    : Form.T;
  success : BOOLEAN;
BEGIN
  form := Form.Create(1, 1, "Person Demo");
  Form.ConstField(0, 1, "Name");
  Form.TextField(0, 15, person.name);
  Form.ConstField(1, 1, "FstName");
  Form.TextField(1, 15, person.fstName);
  Form.ConstField(2, 1, "Age");
  Form.CardField(2, 15, 0, 100, person.age);
  Form.ConstField(3, 1, "Fam. Status");
  Form.EnumField(3, 15, "Single|Married|Divorced", person.familyStatus);
  Form.ConstField(4, 1, "Sel.");
  Form.BoolField(4, 15, 'F', 'T', person.selected);
  success := Form.Edit(form);
END DisplayForm;

PROCEDURE DisplaySetForm(persons : Persons;  VAR person :Person);
VAR
  headerform, rowform : Form.T;
  setform : SetForm.T;
  success : BOOLEAN;
BEGIN
  headerform := Form.Create (1, 1, "Person Demo");
  Form.ConstField(0, 1, "Name");
  Form.ConstField(0, 11, "FstName");
  Form.ConstField(0, 31, "Age");
  Form.ConstField(0, 36, "Fam. Status");
  Form.ConstField(0, 50, "Sel.");
  rowform := Form.Create (0, 0, "Persons");
  Form.TextField(0, 1, person.name);
  Form.TextField(0, 11, person.fstName);
  Form.CardField(0, 31, 0, 100, person.age);
  Form.EnumField(0, 36, "Single|Married|Divorced", person.familyStatus);
  Form.BoolField(0, 50, 'F', 'T', person.selected);
  setform := SetForm.Create(headerform, rowform, person, persons, 10);
  success := SetForm.Choose(setform);
END DisplaySetForm;
```

A.3 INGRES

Schemadefinition

```
Table address
  number : integer2
  street : c20
  zip : c20
  city : c20
  country : c20

Table familystatus
  number : integer2
  status : c20

Table person
  name : c20
  fstname : c20
  age : integer2
  address : integer2
  familystatus : integer2
  selected : integer2
```

Füllen der Tabelle

```
on click =
begin
  i = 1;
  select
    :personTable[i].name = p.name,
    :personTable[i].fstname = p.fstname,
    :personTable[i].age = p.age,
    :personTable[i].address = p.address
  from person p
  begin
    i = i + 1;
  end
end
```

A.4 O_2

```
class Person inherit Object public
type
  tuple(
    name : string,
    fstName : string,
    age : integer,
    address : tuple(
                street : string,
                zip : string,
                city : string,
                country : string),
    familyStatus : string,
    selected : boolean)
end;
```

B. Tabellen

In diesem Anhang sind die aus der Analyse der verschiedenen Systeme (vgl. Tabelle B.1) gewonnenen Erkenntnisse tabellarisch zusammengefaßt. Zum besseren Vergleich wird das Tycoon-System zusätzlich mit in die Tabellen aufgenommen.

Die einzelnen Tabellen beziehen sich jeweils auf einen Schwerpunkt und untergliedern ihn nach unterschiedlichen Kriterien (vgl. Abschnitt 4.1.2). Die Kriterien sind auf der vertikalen Achse aufgetragen. Auf der horizontalen Achse sind der Reihe nach die Systeme wiederzufinden. Die Tabellen erheben jedoch keinen Anspruch auf Vollständigkeit. Aus Platzgründen werden für die einzelnen Systeme die in Tabelle B.1 angegebenen Abkürzungen verwendet. Zusätzlich ist jeweils der Abschnitt angegeben, in dem das System vorgestellt und analysiert wird.

Abkürzung	**System**	**Abschnitt**	**Literatur**
DP	DBPL	4.2.1	[Schmidt, Matthes 92] [Matthes et al. 92a]
IN	INGRES	4.2.2	[Ingres 89] [Ingres 90f]
O2	O_2	4.2.3	[Deux 91] [Bancilhon et al. 92]
GS	GemStone	4.2.4	[Bretl et al. 89] [Butterworth et al. 91]
VB	VisualBASIC	4.2.5	[Ehrmann 93b] [Maslo, Dittrich 93]
TY	Tycoon	3.3	[Matthes 93] [Matthes, Müßig 93]

Tabelle B.1: Abkürzungen für die analysierten Systeme

In den Schnittpunkten der Zeilen und Spalten sind unterschiedliche Zeichen vorzufinden. Die Bedeutung dieser Zeichen wird in Tabelle B.2 erläutert.

Zeichen	Bedeutung
•	vorhanden
(•)	eingeschränkt vorhanden
–	nicht vorhanden
k.A.	keine Angabe

Tabelle B.2: Übersicht der verwendeten Zeichen

B.1 Hard- und Softwareplattformen

Die Tabelle B.3 gibt eine Übersicht über die Hard- und Softwareplattformen auf denen die Systeme jeweils verfügbar sind.

Hardware	Fenstersystem	DP	IN	O2	GS	VB	TY
Sparc	OSF/Motif	•[1]	•	•	•	–	–
	OPEN LOOK	•[1]	–	–	•	–	•
Vax	VMS	•	•	–	•[2]	–	–
PC	Windows	–	•	–	•[2]	•	–[3]
	DOS	•[4]	•	–	–	•	–
Macintosh	Finder	–	•	–	•[2]	–	–[3]

[1] Textuelle Oberfläche.
[2] Nur Klientenbetrieb möglich.
[3] In Vorbereitung.
[4] Ältere Version.

Tabelle B.3: Hard- und Softwareplattformen

B.2 Systemkomponenten

In diesem Abschnitt werden die in den Systemen für Anwendungsentwickler zur Verfügung stehenden Komponenten aufgeführt. Darüberhinaus werden die Sprachschnittstellen aufgeführt mit denen die Systeme angesprochen bzw. erweitert werden können.

Komponente	**DP**	**IN**	**O2**	**GS**	**VB**	**TY**
interaktive Entwicklungsumgebung	–	•	•	•	•	–
Quelltext-Editor	–	•[1]	•[1]	•[2]	•	–
Interpreter	–	–	–	–	•	•
Übersetzer	•	•	•	•	•	•
Debugger	•[3]	•	–[4]	•	•	•
Projektverwaltung	–	•	•[5]	•	•	•
interaktive und kontextsensitive Hilfe	–	–	(•)[6]	–	•	–

[1] 4GL-Editor.
[2] Editor zur visuellen Programmierung.
[3] Debugging nicht auf relationalen Erweiterungen.
[4] Vorgesehen, jedoch noch nicht implementiert.
[5] Über Versionsverwaltung realisiert.
[6] Nur für Syntax der Sprache O_2C.

Tabelle B.4: Komponenten zur Programmbehandlung

Komponente	**DP**	**IN**	**O2**	**GS**	**VB**	**TY**
Ressource-Editor	–	•	–	•	•	•[1]
Maskengenerator	–	•	•	•	–	•
Bibliotheken	•	–	•	–	•	•
Browser	•[2]	–	•	•	(•)[3]	•[4]

[1] Mit Devguide erzeugte Benutzerschnittstellen können angebunden werden.
[2] Generische Anzeige auf Relationenwerte beschränkt.
[3] Nur über zusätzliche Werkzeuge (*custom controls*) möglich.
[4] In Vorbereitung.

Tabelle B.5: Komponenten zur Benutzerschnittstellenbehandlung

Komponente	DP	IN	O2	GS	VB	TY
Administrationswerkzeuge	–[1]	•	•	•	–[2]	–[1]
Schema Designer	–	–	•	•	–[2]	–
Abfragesprache	•[3]	•[4]	•[5]	•[6]	–[2]	•[3]
Transaktionsunterstützung	•	•	•	•	–[2]	•
Persistenz	•	•	•	•	–[2]	•
Fehlererholung	•	•	•	•	–[2]	•
Freispeicherverwaltung	•	•	•	•	–[2]	•

[1] Über Programmiersprache möglich.
[2] Keine Datenbankkomponente integriert.
[3] Abfragesprache ist in Programmiersprache integriert.
[4] SQL.
[5] O_2SQL.
[6] OPAL.

Tabelle B.6: Komponenten zur Datenbankbehandlung

Sprache	DP	IN	O2	GS	VB	TY
integrierte Sprache	•[1]	•[2]	•[3]	•[4]	•[5]	•[6]
C	•	•	•	•	•	•
C++	–	–	•	•	•	•
Modula-2	•	–	–	–	–	–
Smalltalk	–	–	–	•	–	–

[1] DBPL.
[2] INGRES/Windows 4GL.
[3] O_2C.
[4] OPAL.
[5] VisualBASIC.
[6] TL

Tabelle B.7: Sprachschnittstellen

B.3 Visualisierungsmöglichkeiten

In diesem Abschnitt werden die Visualisierungsmöglichkeiten der Systeme vorgestellt. Die Tabellen geben Auskunft über die Datentypen, deren Werte jeweils visualisierbar sind.

Datentyp	DP	IN	O2	GS	VB	TY
Ganzzahlen	•	(•)[1]	•	•	(•)[1]	•
Fließkommzahlen	•	(•)[1]	•	•	(•)[1]	•
Zeichen	•	(•)[1]	•	•	(•)[1]	•
Zeichenketten	•	(•)[1]	•	•	(•)[1]	•
Boole'sche Werte	•	–	•	•	(•)[1]	•
Nullwerte	–[2]	(•)[1]	•	k.A.	–	•
Unterbereiche	•	–	•	k.A.	k.A.	–
Konstanten	•	(•)[1]	•	•	(•)[1]	•
Aufzählungen	•	–	–	k.A.	k.A.	•

[1] Keine Zuordnung zwischen Typ und Schnittstellenkomponente.
[2] Nullwerte werden vom Typsystem nicht unterstützt.

Tabelle B.8: Visualisierung von Werten der Basisdatentypen

Datentyp	DP	IN	O2	GS	VB	TY
Felder	–	–	–	k.A.	k.A.	–
Verbunde	(•)[1]	(•)[1]	•	•	–	•
Varianten	–	–	–	k.A.	–	•

[1] Keine Zuordnung zwischen Typ und Schnittstellenkomponente.

Tabelle B.9: Visualisierung von Werten strukturierter Datentypen

Datentyp	DP	IN	O2	GS	VB	TY
Listen	–	–	•	k.A.	–	•
Mengen	–	–	•	•	–	(•)[1]
Mengen mit Duplikaten	–	–	•	k.A.	–	(•)[1]
heterogene Mengen	–	–	•	k.A.	–	(•)[1]
Relationen	•	•	–[2]	–[2]	–	(•)[1]
geschachtelte Relationen	•	–	–[2]	–[2]	–	(•)[1]
Bäume	–	–	–	k.A.	–	(•)[1]

[1] Auf Listen abbildbar.
[2] Objektorientiertes Datenmodell.

Tabelle B.10: Visualisierung von Werten der Massendatentypen

Datentyp	DP	IN	O2	GS	VB	TY
Bilder	–	(•)[1]	•	k.A.	(•)[1]	–
Audio	–	–	•	k.A.	–	•
Text	•	(•)[1]	•	k.A.	(•)[1]	–
Hypertext	–	–	•	k.A.	–	–

[1]Keine Zuordnung zwischen Typ und Schnittstellenkomponente.

Tabelle B.11: Visualisierung von Werten multimedialer Datentypen

Datentyp	DP	IN	O2	GS	VB	TY
Datum	–	(•)[1]	•	k.A.	–	–
Währung	–	(•)[1]	–	k.A.	–	–
Dateien	–	–	–	k.A.	–	–
Verzeichnisse	–	–	–	k.A.	•	•
Geschäftsgrafiken	–	–	–	k.A.	•	–
hierarchische Listen	–	–	–	k.A.	•	–

[1]Keine Zuordnung zwischen Typ und Schnittstellenkomponente.

Tabelle B.12: Visualisierung von Werten spezieller Datentypen

Eigenschaft	DP	IN	O2	GS	VB	TY
statische Masken	•	•	•	•	•	•
generische Masken	•[1]	(•)[2]	•	•	–	•
typisiert	•	–	•	•	–	•
orthogonale Schachtelung	–	–	•	k.A.	–	•
erweiterbar	•	–	•[3]	•	•	•
visuelles Hierarchie Feedback	–	–	–	k.A.	–	•
mehrfaches Anzeigen	–[4]	–	–	k.A.	–	–
Position in Kollektion	–[5]	(•)[6]	•[7]	k.A.	–	•

[1]Generische Anzeige auf Relationenwerte beschränkt.
[2]Tabellendefinitionen verwertbar.
[3]Über O_2Kit.
[4]Im Fenstersystem kann gleichzeitig nur ein Fenster aktiv sein.
[5]Position aber intern vorhanden.
[6]Programmierbar.
[7]Auf positionalen Datentypen.

Tabelle B.13: Eigenschaften der Visualisierungskomponente

B.4 Interoperabilität

In diesem Abschnitt werden die Interoperabilitätsmöglichkeiten der Systeme vorgestellt. Dabei wird jeweils nach Art der Vorgehensweise beim Datenaustausch und der Komplexität der Daten, die ausgetauscht werden können, unterschieden.

Komplexität	**DP**	**IN**	**O2**	**GS**	**VB**	**TY**
Basiswerte	–[1]	•	•	k.A.	•	–
strukturierte Werte	–[1]	•	•	k.A.	•	–
Massendaten	–[1]	–	•	k.A.	•	–
Multimediadaten	–[1]	–	•	k.A.	•	–
spezielle Daten	–[1]	•	•	k.A.	•	–
anzeigbar	–[1]	–	–	k.A.	•	–
stapelbar	–[1]	–	–	k.A.	–	–
anwendungsübergreifend	–[1]	•	•	k.A.	•	–

[1] Textuelle Oberfläche.

Tabelle B.14: Datenaustausch über die Zwischenablage

Komplexität	**DP**	**IN**	**O2**	**GS**	**VB**	**TY**
Basiswerte	–[1]	•	•	k.A.	•	•
strukturierte Werte	–[1]	•	•	k.A.	•	–
Massendaten	–[1]	–	•	k.A.	•	–
Multimediadaten	–[1]	–	•	k.A.	•	–
spezielle Daten	–[1]	•	•	k.A.	•	–
kopieren	–[1]	•	•	k.A.	•	–
verschieben	–[1]	•	•	k.A.	–	•
anwendungsübergreifend	–[1]	•	•	k.A.	•	•

[1] Textuelle Oberfläche.

Tabelle B.15: Datenaustausch über *drag & drop*

Komplexität	DP	IN	O2	GS	VB	TY
Basiswerte	–[1]	•	–	k.A.	k.A.	•
strukturierte Werte	–[1]	•	–	k.A.	k.A.	•
Massendaten	–[1]	–	–	k.A.	k.A.	•
Multimediadaten	–[1]	–	–	k.A.	k.A.	•
spezielle Daten	–[1]	•	–	k.A.	k.A.	–

[1] Textuelle Oberfläche.

Tabelle B.16: Möglichkeiten zur Fehlerbehebung

Bedienung	DP	IN	O2	GS	VB	TY
Maus	–[1]	•	•	•	•	•
Tastatur	•	•	•	•	•	–

[1] Textuelle Oberfläche.

Tabelle B.17: Bedienung des Systems

Literaturverzeichnis

Adobe Systems 88: Adobe Systems, Inc. *PostScript Language Program Design.* Addison-Wesley Publishing Company, Reading u.a., 1988.

Adobe Systems 90: Adobe Systems, Inc. *PostScript Language Reference Manual.* Addison-Wesley Publishing Company, Reading u.a., 1990.

Adobe Systems 91: Adobe Systems, Inc. *PostScript Language Tutorial and Cookbook.* Addison-Wesley Publishing Company, Reading u.a., 1991.

Ammon 94: Ammon, R.v. *Schaurig schöne GUIs, Style Guides für Benutzerschnittstellen und die Ignoranz.* c't Magazin für Computertechnik, 1994, Nr. 6, S. 46–52.

ANSI 74: ANSI. *American National Standard Programming Language COBOL.* ANS X3.23-1974. ANSI, New York, 1974.

ANSI 78: ANSI. *American National Standard Programming Language FORTRAN.* ANS X3.9-1978. ANSI, New York, 1978.

Appelrath et al. 92: Appelrath, H.-J., Lockemann, P.C., Neuhold, E., Reuter, A., Schek, H.-J., und Schweppe, H. *Interoperable Informationssysteme: Dienste für verteilte, datenintensive Anwendungen.* (unveröffentlicht), 1992.

Apple Computer 92: Apple Computer, Inc. *Macintosh Human Interface Guidelines.* Apple Computer, Inc., Cupertino, 1992.

Atkinson et al. 82: Atkinson, M.P., Chisholm, K.J., und Cockshott, W.P. *PS-algol: An Algol with a Persistent Heap.* ACM SIGPLAN Notices, Jg. 17, 1982, Nr. 7, S. 24–31.

Atkinson et al. 92: Atkinson, M., Bancilhon, F., De Witt, D., Dittrich, K., Maier, D., und Zdonik, S. *The Object-Oriented Database System Manifesto.* In: Bancilhon, F. et al. (Hrsg.). *Building an Object-Oriented Database System: The Story of O_2.* Morgan Kaufmann Publishers, San Mateo u.a., 1992, S. 3–20.

Atkinson, Bunemann 87: Atkinson, M.P. und Bunemann, P. *Types and Persistence in Database Programming Languages.* ACM Computing Surveys, Jg. 19, 1987, Nr. 2, S. 105–190.

Baloui 92: Baloui, S. *Visual Basic, Basic-Programmierung unter Windows*, Beck EDV-Berater, Basiswissen, Bd. 50115. dtv, München, 1992.

Balzer, Mylopoulos 91: Balzer, R. und Mylopoulos, J. *International Workshop on the Development of Intelligent Information Systems.* University of Southern California and University of Toronto, 1991.

Bancilhon et al. 92: Bancilhon, F., Delobel, C., und Kanellakis, P. *Building an Object-Oriented Database System: The Story of O_2.* Morgan Kaufmann Publishers, San Mateo u.a., 1992.

Blaser 90: Blaser, A. (Hrsg.). *Database Systems of the 90s*, Lecture Notes in Computer Science, Bd. 466. Springer-Verlag, Berlin u.a., 1990.

Bobrow et al. 88: Bobrow, D.G., De Michiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., und Moon, D.A. *Common Lisp Object System Specification.* ACM SIGPLAN Notices, Jg. 23, 1988, Special Issue.

Borchert 93: Borchert, D. *Daten-Esperanto, SQL im Überblick: Standards, Front-Ends, Datenbanken.* c't Magazin für Computertechnik, 1993, Nr. 4, S. 100–106.

Borras et al. 92: Borras, P., Mamou, J.C., Plateau, D., Poyet, B., und Tallot, D. *Building user interfaces for database applications: The O_2 experience.* ACM SIGMOD Record, Jg. 21, 1992, Nr. 1, S. 32–38.

Bretl et al. 89: Bretl, R., Maier, D., Otis, A., Penney, J., Schuchardt, B., Stein, J., Williams, E.H., und Williams, M. *The GemStone Data Management System.* In: Kim, W. und Lochovsky, F.H. (Hrsg.). *Object-Oriented Concepts, Databases, and Applications*, Frontier Series. Addison-Wesley Publishing Company, Reading u.a., 1989, S. 283–308.

Bronzite 89: Bronzite, M. *Introduction to ORACLE.* McGraw Hill, Berkeley, 1989.

Budde et al. 92: Budde, R., Kautz, K., Kuhlenkamp, K., und Züllighoven, H. *Prototyping, An Approach to Evolutionary System Development.* Springer-Verlag, Berlin u.a., 1992.

Busch et al. 93: Busch, A., Kuehnel, Th., und Jahnke, A. *StarView C++ Klassenbibliothek.* Star Division, Hamburg, 1993.

Butterworth et al. 91: Butterworth, P., Otis, A., und Stein, J. *The GemStone Object Database Management System.* Communications of the ACM, Jg. 34, 1991, Nr. 10, S. 64–77.

Cardelli et al. 91: Cardelli, L., Martini, S., Mitchell, J.C., und Scedrov, A. *An Extension of System F with Subtyping.* In: Ito, T. et al. (Hrsg.). *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Bd. 526. Springer-Verlag, Berlin u.a., 1991, S. 750–770.

Cardelli, Wegner 85: Cardelli, L. und Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, Jg. 17, 1985, Nr. 4, S. 471–522.

Cardelli 86: Cardelli, L. *Amber.* In: Goos, G. et al. (Hrsg.). *Combinators and Functional Programming Languages*, Lecture Notes in Computer Science, Bd. 242. Springer-Verlag, Berlin u.a., 1986, S. 21–47.

Cardelli 89: Cardelli, L. *Typeful Programming.* Technical Report 45, Digital Equipment Corp., Systems Research Center, Palo-Alto, 1989.

Cardelli 90: Cardelli, L. *The Quest Language and System.* Tracking draft, Digital Equipment Corp., Systems Research Center, Palo-Alto, 1990.

Cattell 91: Cattell, R.G.G. *Next-Generation Database Systems.* Communications of the ACM, Jg. 34, 1991, Nr. 10, S. 31–33.

Cattell 92: Cattell, R.G.G. *Object Data Management, Object-Oriented and Extended Relational Database Systems.* Addison-Wesley Publishing Company, Reading u.a., 1992.

Date 87: Date, C.J. *A Guide to INGRES.* Addison-Wesley Publishing Company, Reading u.a., 1987.

Davison et al. 92: Davison, A., Drake, K., Roberts, W., und Slater, M. *Distributed Window Systems, A Practical Guide to X11 and OpenWindows.* Addison-Wesley Publishing Company, Reading u.a., 1992.

Dearle et al. 89: Dearle, A., Connor, R., Brown, F., und Morrison, R. *Napier88 – A Database Programming Language?* In: Hull, R. et al. (Hrsg.). *Proceedings of the Second International Workshop on Database Programming Languages.* Morgan Kaufmann Publishers, San Mateo u.a., 1989, S. 179–195.

Deux 91: Deux, O. et al. *The O_2 System.* Communications of the ACM, Jg. 34, 1991, Nr. 10, S. 34–48.

Ehrmann 93a: Ehrmann, St. *Die Zwei, VisualBASIC für Windows.* c't Magazin für Computertechnik, 1993, Nr. 4, S. 178–180.

Ehrmann 93b: Ehrmann, St. *Dreisprung, Microsoft VisualBASIC 3.0 für Windows mit Datenbank-Engine.* c't Magazin für Computertechnik, 1993, Nr. 9, S. 114–115.

Fähnrich et al. 92: Fähnrich, K.-P., Janssen, C, und Groh, G. *Entwicklungswerkzeuge für graphische Benutzerschnittstellen.* Computer Magazin, 1992, Nr. 2, S. 6–13.

FIDE 90: Commission of the European Communities. *The FIDE Project, ESPRIT II Basic Research Action 3070*, 1990.

Field, Harrison 88: Field, A.J. und Harrison, P.G. *Functional Programming.* Addison-Wesley Publishing Company, Reading u.a., 1988.

Gawecki, Matthes 94: Gawecki, A. und Matthes, F. *The Tycoon Machine Language TML, An Optimizable Persistent Program Representation.* Draft, Fachbereich Informatik, Universität Hamburg, 1994.

Goldberg, Robson 83: Goldberg, A. und Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Publishing Company, Reading u.a., 1983.

Goossens et al. 94: Goossens, M., Mittelbach, F., und Samarin, A. *The LaTeX Companion.* Addison-Wesley Publishing Company, Reading u.a., 1994.

Gosling et al. 89: Gosling, J., Rosenthal, D.S.H., und Arden, M.J. *The NeWS Book, An Introduction to the Network/extensible Window System.* Springer-Verlag, Berlin u.a., 1989.

Heller et al. 91: Heller, M., Wayner, P., und Smith, B. *Tools for Window Workers.* Byte Magazine, 1991, Nr. 6, S. 139–148.

Heuer 92: Heuer, A. *Objektorientierte Datenbanken, Konzepte, Modelle, Systeme.* Addison-Wesley Publishing Company, Reading u.a., 1992.

Holzgang 90: Holzgang, D.A. *Display PostScript Programming.* Addison-Wesley Publishing Company, Reading u.a., 1990.

Hudak 89: Hudak, P. *Conception, Evolution, and Application of Functional Programming Languages.* ACM Computing Surveys, Jg. 21, 1989, Nr. 3, S. 359–411.

Hüskes, Shahrbabaki 93: Hüskes, R. und Shahrbabaki, K. *Normierter Luxus, GUIs in Theorie und Praxis.* c't Magazin für Computertechnik, 1993, Nr. 9, S. 68–72.

Hüskes 93: Hüskes, R. *Drei Musketiere, Nicht nur für Smalltalk: Modell-View-Controller-Architektur.* c't Magazin für Computertechnik, 1993, Nr. 9, S. 174–180.

Hüskes 94: Hüskes, R. *Baukasten-Software.* c't Magazin für Computertechnik, 1994, Nr. 6, S. 214–220.

Ichbiah 83: Ichbiah, J.D. et al. *The Programming Language Ada: Reference Manual.* MIL-STD-1815A-1983. ANSI, New York, 1983.

Ilg, Ziegler 88: Ilg, R. und Ziegler, J. *Direkte Manipulation.* In: Balzert, H. et al. (Hrsg.). *Einführung in die Software-Ergonomie*, Mensch-Computer-Kommunikation: Grundwissen, Bd. 1. de Gruyter, Berlin u.a., 1988, S. 175–194.

Informix Software 86: Informix Software Corp. *Informix-4GL Reference Manual.* Informix Software Corp., Menlo Park, 1986.

Ingres 89: Ingres Corp. *Introducing INGRES for the UNIX and VMS Operating Systems.* Ingres Corp., Alameda, 1989.

Ingres 90a: Ingres Corp. *Application Editor User's Guide for INGRES/Windows 4GL for the UNIX and VMS Operating Systems.* Ingres Corp., Alameda, 1990.

Ingres 90b: Ingres Corp. *INGRES ABF/4GL Reference Manual for the UNIX and VMS Operating Systems.* Ingres Corp., Alameda, 1990.

Ingres 90c: Ingres Corp. *INGRES ABF/4GL User's Guide for the UNIX and VMS Operating Systems.* Ingres Corp., Alameda, 1990.

Ingres 90d: Ingres Corp. *INGRES Database Administrator's Guide for the VMS Operating System.* Ingres Corp., Alameda, 1990.

Ingres 90e: Ingres Corp. *INGRES Embedded SQL Companion Guide for C.* Ingres Corp., Alameda, 1990.

Ingres 90f: Ingres Corp. *Language Reference Manual for INGRES/Windows 4GL for the UNIX and VMS Operating Systems.* Ingres Corp., Alameda, 1990.

Isau 93: Isau, R. *Was ihr wollt, Lösungsansätze zur plattformübergreifenden Programmierung.* c't Magazin für Computertechnik, 1993, Nr. 10, S. 68–74.

Jones 89: Jones, O. *Introduction to the X Window System.* Prentice Hall, Englewood Cliffs u.a., 1989.

Kaß 94: Kaß, Th. *Objektorientierte Datenmodellierung: Ein Klasseneditor zur Entwurfsunterstützung.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

Kernighan, Ritchie 77: Kernighan, B.W. und Ritchie, D.M. *The C Programming Language.* Prentice Hall, Englewood Cliffs u.a., 1977.

Kilberth et al. 93: Kilberth, K., Gryczan, G., und Züllighoven, H. *Objektorientierte Anwendungsentwicklung, Konzepte, Strategien, Erfahrungen.* Vieweg, Braunschweig, 1993.

Kiradjiev 94: Kiradjiev, P. *Dynamische Optimierung von Lambda-Kalkül-basierten Programmrepräsentationen.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

Kirch, Müßig 92: Kirch, F. und Müßig, S. *Entwicklung eines generischen Datenbankbrowsers in einer polymorphen Programmiersprache.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1992.

Krogull 92: Krogull, D. *Benutzergestaltbare Interaktion mit Unix-Programmen unter der OPEN LOOK GUI.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1992.

Lauer 92: Lauer, Th. *Briefkasten, Clipboard-Programmierung.* c't Magazin für Computertechnik, 1992, Nr. 12, S. 242–251.

Lauer 93a: Lauer, Th. *Höllenprotokoll, Client-Server-Programmierung mit DDE.* c't Magazin für Computertechnik, 1993, Nr. 1, S. 176–187.

Lauer 93b: Lauer, Th. *Paketdienst, Grundlagen zu Object Linking and Embedding (OLE).* c't Magazin für Computertechnik, 1993, Nr. 4, S. 264–272.

Lee 93: Lee, G. *Object-Oriented GUI Application Development.* Prentice Hall, Englewood Cliffs u.a., 1993.

Lindner 93: Lindner, St. *TEX auf dem Atari ST/TT, Eine kleine Einführung*, 1993.

Lingnau 92: Lingnau, A. *An Improved Environment for Floats*, 1992.

Löst, Möller 92: Löst, R. und Möller, K. *Entwurf einer Graphischen Schnittstelle für objektorientierte Datenbank-Spezifikation.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1992.

Löst 94: Löst, R. *Ein objektorientierter Grafikeditor.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

Mandelkern 93: Mandelkern, D. *Graphical User Interfaces: The Next Generation.* Communications of the ACM, Jg. 36, 1993, Nr. 4, S. 36–39.

Manthey 91: Manthey, R. *Declarative Languages – Paradigm of the Past or Challenge of the Future?* In: Schmidt, J.W. et al. (Hrsg.). *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, Lecture Notes in Computer Science, Bd. 504. Springer-Verlag, Berlin u.a., 1991, S. 1–16.

Markus 92: Markus, Chr. *Untersuchung verschiedener Methoden zur Entwicklung von Programmen mit grafischer Benutzerschnittstelle.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 1992.

Maslo, Dittrich 93: Maslo, P. und Dittrich, St. *Das große Buch zu VisualBASIC 3.0 für Windows.* Data Becker, Düsseldorf, 1993.

Mathiske et al. 93: Mathiske, B., Matthes, F., und Müßig, S. *The Tycoon System and Library Manual.* DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, 1993.

Matthes et al. 92a: Matthes, F., Rudloff, A., Schmidt, J.W., und Subieta, K. *The Database Programming Language DBPL: User and System Manual.* FIDE Technical Report Series FIDE/92/47, FIDE Project Coordinator, Department of Computing Science, University of Glasgow, 1992.

Matthes et al. 92b: Matthes, F., Rudloff, A., Schmidt, J.W., und Subieta, K. *A Gateway from DBPL to Ingres: Modula-2, DBPL, SQL+C, Ingres.* FIDE Technical Report Series FIDE/92/54, FIDE Project Coordinator, Department of Computing Science, University of Glasgow, 1992.

Matthes, Müßig 93: Matthes, F. und Müßig, S. *The Tycoon Language TL: An Introduction.* DBIS Tycoon Report 112-93, Fachbereich Informatik, Universität Hamburg, 1993.

Matthes, Schmidt 91a: Matthes, F. und Schmidt, J.W. *Bulk Types: Built-In or Add-On?* In: Schmidt, J.W. et al. (Hrsg.). *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, San Mateo u.a., 1991, S. 33–54.

Matthes, Schmidt 91b: Matthes, F. und Schmidt, J.W. *Towards Database Application Systems: Types, Kinds and Other Open Invitations.* In: Schmidt, J.W. et al. (Hrsg.). *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, Lecture Notes in Computer Science, Bd. 504. Springer-Verlag, Berlin u.a., 1991, S. 185–211.

Matthes, Schmidt 93: Matthes, F. und Schmidt, J.W. *System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways.* In: Spies, P.P. (Hrsg.). *Proceedings of Euro-ARCH'93 Congress.* Springer-Verlag, Berlin u.a., 1993, S. 301–317.

Matthes 91: Matthes, F. *P-Quest: Installation and User Manual.* DBIS Tycoon Report 101-91, Fachbereich Informatik, Universität Hamburg, 1991.

Matthes 93: Matthes, F. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, Berlin u.a., 1993.

Mauny 91: Mauny, M. *Functional Programming using CAML.* RT 129, INRIA, Le Chesnay, 1991.

Meyer 94: Meyer, H.-M.. *Objektorientierte Technologien für die User Interface Entwicklung.* OBJEKTspektrum, 1994, Nr. 1, S. 24–29.

Microsoft 92: Microsoft Corp. *The Windows Interface: An Application Design Guide.* Microsoft Corp., Redmond, 1992.

Minker 88: Minker, J. *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann Publishers, San Mateo u.a., 1988.

Müller 91: Müller, R. *Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und -implementierung.* Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, 1991.

Myers, Rosson 92: Myers, B.A. und Rosson, M.B. *Survey on User Interface Programming.* Technical Report CMU-CS-92-113, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1992.

Myers 92: Myers, B.A. *State of the Art in User Interface Software Tools.* Technical Report CMU-CS-92-114, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1992.

Nelson 91: Nelson, G. (Hrsg.). *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, Englewood Cliffs u.a., 1991.

Neukam 93: Neukam, F. *Die Document-Style-Famile Script*, 1993.

NeXT 91: NeXT, Inc. *NeXTStep and the NeXT Interface Builder.* NeXT, Inc., Redwood City, 1991.

Niederée et al. 92: Niederée, C., Müßig, S., und Matthes, F. *P-Quest User Manual.* DBIS Tycoon Report 102-92, Fachbereich Informatik, Universität Hamburg, 1992.

O_2 Technology 93: O_2Technology. *The O_2 User Manual.* O_2Technology, Versailles, 1993.

Oberquelle 92: Oberquelle, H. *Software-Ergonomie I + II.* Vorlesungsskript, Fachbereich Informatik, Universität Hamburg, 1992.

Open Software Foundation 90: Open Software Foundation. *OSF/Motif: Style Guide.* Prentice Hall, Englewood Cliffs u.a., 1990.

Oppermann et al. 92: Oppermann, R., Murchner, B., Reiterer, M., und Koch, M. *Software-Ergonomische Evaluation, Der Leitfaden EVADIS II.* de Gruyter, Berlin u.a., 1992.

Oracle 91: Oracle Corp. *PL/SQL User's Guide and Reference.* Oracle Corp., Redwood Shores, 1991.

Partl et al. 90: Partl, H., Schlegl, E., und Hyna, I. *LaTeX-Kurzbeschreibung.* EDV-Zentrum der Technischen Universität Wien, 1990.

Ricken 94: Ricken, W. *DirectTeX, A complete TeX program package for the Apple Macintosh*, 1994.

Rovner et al. 85: Rovner, P., Levin, R., und Wick, J. *On Extending Modula-2 for Building Large, Integrated Systems.* Technical Report 3, Digital Equipment Corp., Systems Research Center, Palo-Alto, 1985.

Rudloff 90: Rudloff, A. *Datenkonstruktion in typvollständigen Datenbankprogrammiersprachen.* Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, 1990.

Salcher 93: Salcher, E. *Das X-Window-System und MS-Windows im Vergleich.* BI-Wissenschaftsverlag, Mannheim u.a., 1993.

Schaufler 89: Schaufler, R. *X11/NeWS Design Overview.* Sun Microsystems, Inc., Mountain View, 1989.

Schmidt, Matthes 90: Schmidt, J.W. und Matthes, F. *Language Technology for Post-Relational Data Systems.* In: Blaser, A. (Hrsg.). *Database Systems of the 90s*, Lecture Notes in Computer Science, Bd. 466. Springer-Verlag, Berlin u.a., 1990, S. 81–114.

Schmidt, Matthes 92: Schmidt, J.W. und Matthes, F. *The Database Programming Language DBPL: Rationale and Report.* FIDE Technical Report Series FIDE/92/46, FIDE Project Coordinator, Department of Computing Science, University of Glasgow, 1992.

Schröder 91: Schröder, G. *Die Standardisierung von Modula-2.* Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1991.

Sheldon et al. 91: Sheldon, K.M., Barron, J.J., und Smith, B. *Window Wars.* Byte Magazine, 1991, Nr. 6, S. 124–134.

Siering 92a: Siering, P. *Viele Impulse, Windows 3.1 aus Programmierersicht.* c't Magazin für Computertechnik, 1992, Nr. 8, S. 150–152.

Siering 92b: Siering, P. *Vom Fenster zum Programm, Windows-Programmiermethoden im Vergleich.* c't Magazin für Computertechnik, 1992, Nr. 12, S. 206–218.

Soloviev 92: Soloviev, V. *An Overview of Three Commercial Object-Oriented Database Management Systems: ONTOS, ObjectStore and O_2.* ACM SIGMOD Record, Jg. 21, 1992, Nr. 1, S. 93–104.

Stonebraker et al. 90: Stonebraker, M., Rowe, L.A., Lindsay, B., Gray, J., Carey, M., Brodie, M., und Bernstein, P. *Third-Generation Data Base System Manifesto.* ACM SIGMOD Record, Jg. 19, 1990, Nr. 3, S. 31–44.

Stroustrup 92: Stroustrup, B. *The C++ Programming Language.* Addison-Wesley Publishing Company, Reading u.a., 1992.

Sun Microsystems 90a: Sun Microsystems, Inc. *OPEN LOOK Graphical User Interface Application Style Guidelines.* Addison-Wesley Publishing Company, Reading u.a., 1990.

Sun Microsystems 90b: Sun Microsystems, Inc. *OPEN LOOK Graphical User Interface Functional Specification.* Addison-Wesley Publishing Company, Reading u.a., 1990.

Sun Microsystems 90c: Sun Microsystems, Inc. *Network Interfaces Programmers's Guide.* Sun Microsystems, Inc., Mountain View, 1990.

Sun Microsystems 90d: Sun Microsystems, Inc. *X11/NeWS Version 2 Server Guide.* Sun Microsystems, Inc., Mountain View, 1990.

Sun Microsystems 92: Sun Microsystems, Inc. *OpenWindows Developer's Guide 2.0 User's Guide.* Sun Microsystems, Inc., Mountain View, 1992.

SunSoft 92a: SunSoft, Inc. *NeWS 3.1 Programmer's Guide.* SunSoft, Inc., Mountain View, 1992.

SunSoft 92b: SunSoft, Inc. *The NeWS Toolkit 3.1 Reference Manual.* SunSoft, Inc., Mountain View, 1992.

SunSoft 93a: SunSoft, Inc. *OpenWindows Version 3.2 Reference Manual.* SunSoft, Inc., Mountain View, 1993.

SunSoft 93b: SunSoft, Inc. *Solaris 2.2 User's Guide.* SunSoft, Inc., Mountain View, 1993.

Tanenbaum 87: Tanenbaum, A.S. *Operating Systems, Design and Implementation.* Prentice Hall, Englewood Cliffs u.a., 1987.

Watt 90: Watt, D.A. *Programming Language Concepts an Paradigms.* Prentice Hall, Englewood Cliffs u.a., 1990.

Wetzel 94: Wetzel, I. *Programmieren mit STYLE – Über die systematische Entwicklung von Programmierumgebungen.* Dissertation, Fachbereich Informatik, Universität Hamburg, 1994.

Wirth 71: Wirth, N. *The programming language PASCAL.* Acta Informatica, Jg. 1, 1971, Nr. 1, S. 35–63.

Wirth 85: Wirth, N. *Programming in Modula-2.* Springer-Verlag, Berlin u.a., 1985.

Wirth 87: Wirth, N. *The Programming Language Oberon.* Technical Report 82, Institut für Informatik, Eidg. Technische Hochschule Zürich, 1987.

Zierke 93: Zierke, R. *LaTeX auf den UNIX-Rechnern des FB Informatik (Local Guide).* Fachbereich Informatik, Universität Hamburg, 1993.

Zoschke 91: Zoschke, H. *Sichtbare Fortschritte, Microsofts VisualBASIC für Windows.* c't Magazin für Computertechnik, 1991, Nr. 8, S. 112–118.

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Diplomarbeit selbständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 5. Juli 1994

(Sven Müßig)