Objektorientierte Datenmodellierung: Ein Klasseneditor zur Entwurfsunterstützung

Studienarbeit von
Thomas Kaß

Januar 1994

Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme
Vogt-Kölln-Straße 30
D-22527 Hamburg

Danksagungen

Die vorliegende Arbeit wäre wahrscheinlich nie ohne die Hilfe einiger Mitarbeiter entstanden. Mein besonderer Dank gilt:
Sven Müßig, der mich bei der Einarbeitung in das NeWS-Toolkit unterstützte und mir wertvolle Tips zur optischen Verbesserung des Editors gab,
Ralf Löst, der mich beriet wenn es darum ging, graphische Funktionen zu realisieren, die den Umfang der Quest-Schnittstelle überstiegen,
Petra Münnix, die das Endprodukt unermüdlich testete und auf versteckte Mängel aufmerksam machte. Ihr ist es zu verdanken, daß der Klasseneditor in einer eigens eingerichteten Demo-Umgebung in ein optimales Licht gerückt wurde,
Andreas Rudloff, der auch bei hartnäckigen Problemen Ausdauer zeigte und stets einen kühlen Kopf bewahrte und schließlich
Ingrid Wetzel, die das Projekt leitete und mir eine intensive Betreuung zukommen ließ. Sie half mir unter anderem diesen Text von Metaphern zu befreien, um ihm so einen wissenschaftlicheren Anspruch zu verleihen.

Inhaltsverzeichnis

Kapitel 1

Vorwort und Motivation

Das Entwickeln und Betreiben großer Datenbanken hat in den vergangenen Jahrzehnten zunehmend an Bedeutung gewonnen. Im Zuge eines ständig wachsenden Informationsbedarfs werden Datenbanken und Informationssysteme einerseits immer umfangreicher und komplexer; andererseits kommt es zu einer Erweiterung der Anwendungsfelder. Mit zunehmender Komplexität der Systeme jedoch ist es unverzichtbar, geeignete Werkzeuge zur Verfügung zu stellen, die den Vorgang der Datenmodellierung in bestmöglicher Weise unterstützen und dabei eine große Nähe zur textuellen Repräsentation des zugrundeliegenden Datenmodells aufweisen. Ziel dieser Studienarbeit ist es, dieses Problem durch die Implementierung eines multifunktionalen Texteditors zu lösen, der zum einen eine gewisse Grundfunktionalität bereitstellt, zum anderen jedoch auch externe Dienste integriert.
Als Datenentwurfsumgebung stehen hierbei die objektorientierte Datenmodellierungssprache OM1[1], Generatoren für die Transformation von textuellen Klassendefinitionen in die Programmiersprache TYCOON sowie Funktionen zur Modellierung und Datenbankbenutzung zur Verfügung. Für die programmiertechnische Realisierung ist die strikt typisierte, polymorphe, funktionale Programmiersprache *Quest* gewählt worden, die von Luca Cardelli bei DEC SRC in Palo Alto, USA entwickelt wurde [Car89]. Aufgrund vieler abgeschlossener und noch laufender Projekte kann auf eine Vielzahl bereits bestehender Bibliotheksfunktionen zurückgegriffen werden, die aufgrund ihrer Generizität nur geeignet instanziiert werden mußten, beziehungsweise mit geringem Aufwand angepaßt werden konnten.
Im Rahmen einer angemessenen Benutzerführung muß für die genannten Aufgaben die Unterstützung durch eine graphische Benutzeroberfläche in Betracht gezogen werden. Da bereits *Open Windows* als Anwendungsumgebung zur Verfügung steht, liegt es nahe, die Dienste eines Entwicklungswerkzeuges in Anspruch zu nehmen, das den *Open Look*-Standard unterstützt. Die Entscheidung für das *NeWS*[2] *Toolkit* wird im Laufe der Arbeit näher begründet; beeinflußt wurde diese Wahl auch dadurch, daß die NeWS-Funktionen bereits in einer Quest-Programmbibliothek vorliegen.
Der Aufbau der Arbeit gliedert sich wie folgt:
Kapitel zwei gibt eine Einführung in die grundlegende Konzepte semantischer und objek-

[1] **O**bject **M**odelling language 1
[2] **N**etwork-**e**xtensible **W**indow **S**ystem

torientierter Datenmodellierung. Anschließend wird die Datenmodellierungssprache OM1 vorgestellt und eine Einordnung dieser Arbeit in die Gesamtentwurfsumgebung STYLE[3] gegeben. Den Abschluß bildet die Herausarbeitung eines ersten groben Anforderungsprofils für den zu erstellenden Editor.
Die Konzeption der Funktionalität des zu realisierenden Klasseneditors ist Gegenstand des dritten Kapitels. Hier werden die grundlegenden und weiterführenden Konzepte, sowie die damit verfolgten Ziele herausgearbeitet.
Im vierten Kapitel folgt eine Darstellung der Werkzeuge, die zur Realisierung der Editorfunktionalität benutzt wurden. Es werden die Ausdrucksmächtigkeit der Sprache Quest, sowie die grundlegenden graphischen Bausteine vorgestellt, die das NeWS-Toolkit zur Verfügung stellt und die hier Anwendung finden. Der folgende Abschnitt beschreibt die Integration der externen Dienste in die Implementierungssprache Quest. Zuletzt wird der sukzessive Aufbau des Editors aus diesen Basiskomponenten erläutert.
Mit der Realisierung der in Kapitel drei geforderten Konzepte beschäftigt sich das fünfte Kapitel. Hier wird die konkrete Implementierung von Funktionen analysiert und in Auszügen dargestellt.
Die Erfahrungen im Umgang mit den benutzen Werkzeugen und eine kritische Abschlußbewertung meiner Arbeit mit Vorschlägen zu weitergehenden Verbesserungen sind dann die Bestandteile des abschließenden Ausblicks.

[3]STYLE steht für **S**ystematics of **T**yped **L**anguage **E**nvironments

Kapitel 2

Einleitung und Szenario

Dieses Kapitel stellt die grundlegenden Konzepte semantischer und objektorientierter Datenmodellierung vor. Ausgehend von diesen allgemeinen Konzepten wird dann die konkrete Datenmodellierungssprache OM1 vorgestellt und die Arbeit wird zur Gesamtenwurfsumgebung STYLE in Bezug gesetzt. Die Entwicklung eines groben Anforderungsprofils für den zu implementierenden textuellen Editor bildet dann den Abschluß dieses Kapitels.

2.1 Semantische Modellierung

Die Motivation für ein semantisches Datenmodell entspricht dem Ziel der Objektorientierung: die Modellierung von Daten aus der realen Welt und deren Beziehungen zueinander mit dem Anliegen, sie so wirklichkeitsnah wie möglich zu beschreiben [AbKh90]. Ein semantisches Datenmodell erlaubt zwar keine Modellierung abstrakter Datentypen, also eine Abstraktion bezüglich des Verhaltens von Objekten; es stellt aber die Möglichkeit zur strukturellen Abstraktion zur Verfügung. Es ermöglicht es unter anderem, einen Teil der Integritätsbedingungen strukturinhärent zu definieren.
Aufgrund der stark eingeschränkten Fähigkeit, Datenbankoperationen adäquat auszudrücken, werden semantische Datenmodelle in erster Linie als Werkzeug zum Entwurf und zur Modellierung von Datenstrukturen eingesetzt. Real existierende Datenbanksysteme, die ausschließlich auf semantischen Datenmodellen basieren, sind bisher kaum bekannt. Das Ergebnis eines Datenentwurfs ist daher häufig ein relationales Datenbankschema. Nach [Heu92] lassen sich statische Konzepte wie

1. Objekttypen und Typkonstruktoren zum Aufbau komplexer Datentypen (Aggregation, Gruppierung)
2. Funktionen zwischen Objekttypen
3. Is-a-Beziehungen
4. Metatypen, die die Aggregation von komplexen Objektypen erlauben

5. statische Integritätsbedingungen wie beispielweise Schlüsselbedingungen und Disjunktheitsforderungen zwischen gewissen Objekttypen

und dynamische Konzepte wie

1. abgeleitete Attribute
2. Dämonen oder *Trigger*, die beim Eintreten eines bestimmten Ereignisses gewisse Nachfolgefunktionen auslösen können
3. dynamische Integritätsbedingungen wie Einschränkungen von Operationen auf den Objekttypen

unterscheiden.
Ein Beispiel für die graphische Repräsentation eines semantischen Datenmodells zeigt die folgende Abbildung:

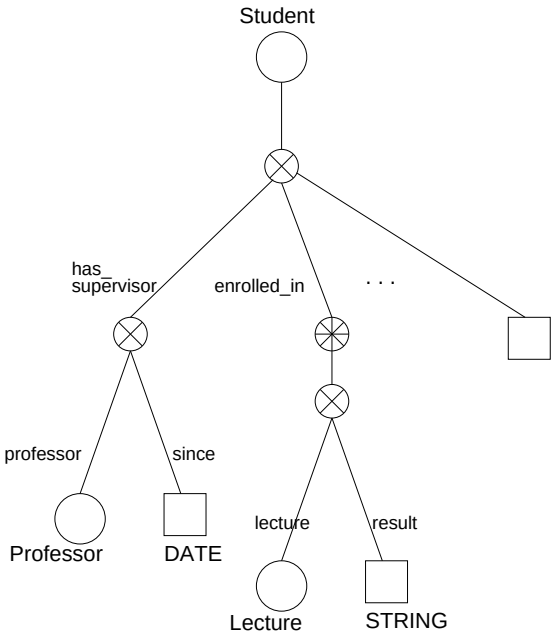

Abbildung 2.1: Beispiel für eine semantische Datenmodellierung

Zur Darstellung von Objektklassen wurden Kreise gewählt, Quadrate repräsentieren Typen. Mengenkonstruktoren sind auf "Wagenräder" abgebildet worden und ein Kreuz in einem Kreis bedeutet Tupelkonstruktor. Dem Beispiel kann somit entnommen werden, daß jeder Student seit einem bestimmten Datum einen bestimmten Professor hat und an einer Menge von Vorlesungen teilnimmt, für die er einen Leistungsnachweis erhält.

2.2 Ein objektorientiertes Datenmodell

Das objektorientierte Datenmodell kann als eine Weiterführung des semantischen Datenmodells angesehen werden, da es die selben Ziele verfolgt. Zusätzlich wird hier jedoch versucht, Konzepte objektorientierter Programmiersprachen miteinzubeziehen. In diesem Kapitel wird versucht, dem Leser diese Konzepte näherzubringen.

2.2.1 Grundbegriffe der Objektorientierung

Werte sind Ausprägungen von Typen. Sie sind Abstraktionen von Informationen, die selbsterklärend sind und daher keiner weiteren Erläuterung bedürfen.

Objekte

1. werden durch eine Menge von Werten, Beziehungen zu anderen Objekten und anwendbarer Methoden beschrieben
2. besitzen eine unveränderliche Identität, die dem Benutzer verborgen bleibt. Sie ist unabhängig von sie beschreibenden Werten oder implementationsbezogener Adressierbarkeit.
3. gehören mindestens einer Klasse (Erklärung folgt) an, über die sie definiert und kreiert werden
4. haben einen veränderlichen Zustand, der nur über objekteigene Methoden verändert werden kann

Klassen

1. dienen der Spezifikation applikationssensitiver Abstraktionen
2. legen die (unveränderliche) strukturelle Beschreibung für eine Objektmenge fest
3. ist eine Menge von Objekten zugeordnet, die ihre Beschreibung erfüllen (*Extension*). Die Extension einer Menge ist dynamisch; sie kann zu verschiedenen Zeitpunkten unterschiedlich sein, da Klassenobjekte migrieren, neu erzeugt oder gelöscht werden können. Extensionen sind nicht disjunkt, da ein Objekt mehreren Klassen angehören kann. Die Extension einer Subklasse ist Teilmenge der Extension ihrer Superklasse.
4. besitzen einen veränderlichen Zustand, der sich aus der Anzahl der in einer Klassenextension enthaltenen Objekte ergibt.
5. bieten den Rahmen für die Identifizierbarkeit von Objekten innerhalb ihrer Extensionen.

2.2.2 Die Datenmodellierungssprache OM1

Ziel der Datenmodellierung in OM1 ist der Entwurf von Schemata. Diese bestehen aus Typ- und Klassendefinitionen, auf die im folgenden eingegangen werden soll:
Für die Modellierung von **Typdefinitionen** stehen Basistypen (Nat, Int, String und Bool) zur Verfügung, aus denen mit Hilfe von *Typkonstruktoren* wie Verbunden (*Records*), Mengen (*SetOf*), Listen (*ListOf*), Feldern (*Array*) und Auswahltypen (*Option*) auch komplexere Typen erzeugt werden können. Des weiteren können in OM1 Aufzählungstypen (*enumOf*) und rekursive Typen, sowie in naher Zukunft auch prädikativ eingeschränkte Typen definiert werden.
Eine Klassendefinition gliedert sich in die folgenden (optionalen) Komponenten:

Class <Name>
<Parameter>
<Instantiierung>
<Spezialisierung>
<Struktur>
<Integritätsbedingungen>
<Methoden>
End

Die Komponenten Parameter und Instanziierung dienen der Spezifikation generischer Klassen. Auf sie soll im Rahmen dieser Arbeit nicht näher eingegangen werden.

In der **Spezialisierungskomponente** kann das Vererbungsverhalten in einer Subklassen-Superklassenbeziehung spezifiziert werden. Eine rein *extensionale* Spezialisierung (*isSubClassOf*) zeigt an, daß eine Subklasse zwar zur Extension einer Superklasse gehört, jedoch keine ihrer Eigenschaften erbt. Sie ist abzugrenzen von einer Spezialisierung, die *intensional* und extensional erfolgt. Hier erbt sie Subklasse zusätzlich die strukturelle Beschreibung der Superklasse.

In der **Strukturkomponente** werden die Attribute einer Klasse spezifiziert, deren Instanziierungen später den Zustand eines Objektes beschreiben. Als Modellierungshilfe sind *Referenzen* auf bereits existente Klassen- oder Typdefinitionen erlaubt. Die Klassifikation von Attributen erfolgt in:

- *Schlüsselattribute*, die dem Benutzer die Identifikation von Objekten ermöglichen,
- abgeleitete Attribute, die dynamisch aus dem aktuellen Zustand der Datenbank berechnet werden,
- konstante Attribute, die nach ihrer Initialisierung nicht mehr modifizierbar sind und

- *partielle Attribute*, die keinen Wert haben müssen. Mit ihnen lassen sich NULL-Werte modellieren.

Integritätsbedingungen sind Beschränkungen auf den Daten und Methoden, die die Konsistenz der Datenbank gewährleisten sollen. Sie können

- zustandsunabhängig (*static*),
- bei bestimmten transaktionsbedingten Zustandsübergängen (*transition*),
- bei der Objekterzeugung oder
- beim Löschen eines Objektes

gelten. OM1 erlaubt die Benennung solcher Integritätsbedingungen, um später auf sie Bezug nehmen zu können. Ihre Formulierung erfolgt in Termen der Prädikatenlogik 1.Ordnung.

Methoden schließlich definieren die Transaktionen, über die der Zustand von Objekten einer Klasse verändert werden kann.

2.3 Einordnung in das Gesamtszenario

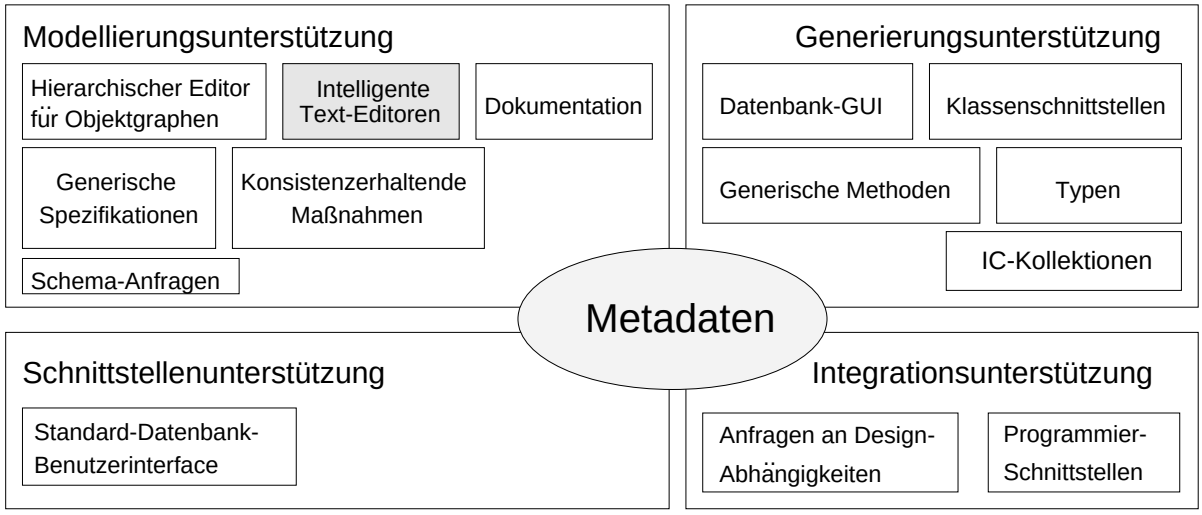

Abbildung 2.2: Das OM1-Szenario

STYLE ist eine interaktive Entwurfsumgebung zum systematischen Design datenintensiver Anwendungen. Der Editor ist hierbei nur eine Komponente des Gesamtsystems. Die gesamte Umgebung setzt sich aus folgenden Teilen zusammen:

1. einer Integrationskomponente, die das Speichern und Laden von Klassen-, Typ- und Schemadefinitionen erlaubt. Sie wurde in Gestalt einer Metadatenverwaltung realisiert und nimmt eine zentrale Stellung in diesem System ein.

2. Funktionen zur Modellierungsunterstützung

   - einem System- und einem Schemaeditor, mit deren Hilfe neue Anwendungen erzeugt, oder bestehende geöffnet werden können.
   - einem Typeditor zur Modellierung von Typen.
   - dem hier beschriebenen Klasseneditor für die Modellierung von Klassendefinitionen
   - einer Bibliothek von Standardanfragen, die dem Benutzer in der Modellierungsphase Abhängigkeiten zwischen Typen und Klassen aufzeigt und sein Informationsbedürfnis befriedigt.
   - einem Graphikeditor zur Visualisierung einer Anwendung und deren Beziehungsgeflecht. Dieser erlaubt die 1:1 Abbildung zwischen graphischer und textueller Repräsentation und ermöglicht seinerseits die Datenmodellierung.
   - eine Schnittstelle zur Ausgabe einer Klassendokumentation auf dem Drucker. Das Layout des Ausdrucks entsteht in Anlehnung an die Syntax von OM1 und den zur Definition benutzten Editor.
   - Funktionen zum Parsen bestehender Definitionen, um deren syntaktische Korrektheit zu verifizieren.

3. Funktionen zur Generierungsunterstützung

   - Generatorfunktionen, die ein Übersetzen von OM1-Spezifikationen in eine andere Implementationssprache unter Beibehaltung ihrer Funktionalität erlaubt. Diese sind in für verschiedene Granularitäten (Typen, Klassen, Schemata, Methoden, Integritätsbedingungen und Editoren) definiert.

4. Funktionen zur Integrationsunterstützung

   - eine Anfragekomponente, mit der Abhängigkeiten zwischen Klassendefinitionen und ihren generierten Moduln angezeigt werden können.
   - einer Schnittstelle zu sämtlichen Bibliotheksmoduln. Diese versetzt den Benutzer in die Lage, sich die Anzahl und Typen der von einer Funktion erwarteten Parameter anzeigen zu lassen.

5. Funktionen zur Schnittstellenunterstützung

   - Alle Funktionen, die die Programmierumgebung unterstützen, können von einer eigenen graphisch unterstützten Schnittstelle aus aufgerufen werden.

2.4 Anforderungen an einen textuellen Editor

Das Benutzen eines Editors ist eng mit einer gewissen Erwartungshaltung bezüglich seiner Funktionalität und dem Design verbunden. In einem Zeitalter, in dem die Hardwarekosten immer mehr sinken, die Rechnergeschwindigkeiten aber steigen, muß daher auch über eine anspruchsvolle graphische Repräsentation von Programmen nachgedacht werden, um die erzielten Performanzsteigerungen nutzbringend einzusetzen. Kein Unternehmen der 90er Jahre kann es sich heute mehr erlauben, ein kommerzielles System ohne graphische Benutzeroberfläche auf den Markt zu bringen. Neben einer benutzeradäquaten Programmgestaltung, sollte ein Editor über folgende Fähigkeiten verfügen:

1. Der Editor muß *ganzseitenorientiert* sein. Er muß das Einfügen, Überschreiben und Löschen einzelner Zeichen über die Tastatur erlauben. Die aktuelle Textstelle, auf die sich die Tastaturereignisse beziehen, ist durch eine Markierung (*Cursor, Caret*) zu kennzeichnen.

2. Um überhaupt leicht und sinnvoll mit seiner graphischen Benutzeroberfläche interagieren zu können, ist für den Editor die Eingabemöglichkeit über die Tastatur allein nur sehr unzureichend. Daher muß zusätzlich auch die Kontrollmöglichkeit durch eine Maus, zum Öffnen von Menüs, Drücken von Knöpfen, Markieren u.s.w. bereitgestellt werden.

3. Er muß das Markieren einer Textpassage ermöglichen. Die gekennzeichneten Daten müssen gelöscht(*cut*), überschrieben(*replace*) oder verschoben (*cut & paste*) werden können.

4. Es muß eine Prozedur zum gezielten Suchen eines kurzen Textabschnitts, gegebenenfalls in Verbindung mit einer Möglichkeit zur Substitution desselben, vorhanden sein. Hierbei sollten bei wiederholtem Auftreten desselben Suchbegriffs alle Vorkommen angezeigt werden.

5. Eine Schnittstelle zur Speicherverwaltung muß das Speichern erzeugter Dokumente aus und umgekehrt das Laden persistent gehaltener Informationen in den Editor ermöglichen. Dabei muß es eine Möglichkeit geben, den Verzeichnispfad in einfacher Weise neu zu setzen.

6. Nach erfolgreicher Abarbeitung einer Befehlsfolge sollte der Benutzer nicht im Unklaren darüber gelassen werden, ob die von ihm gewünschte Aktion auch tatsächlich durchgeführt wurde. Ebenso darf er in kritischen Situation nicht gänzlich auf sich allein gestellt sein. Kontextsensitive Systemmeldungen mit vorgeschlagenen Handlungsalternativen, die einen Hilfsfunktionscharakter haben, sollen dem Benutzer Gelegenheit geben, mit dem Editor in geeigneter Weise zu interagieren.

Kapitel 3

Konzeption des Klasseneditors

Bevor mit der Implementierungsarbeit begonnen werden kann, ist zunächst erst einmal eine genaue Analyse von Aussehen und Funktionalität der Applikation angezeigt. Diese wird dann den eigentlichen Entwurfsvorgang und dessen Umsetzung in erheblichen Maße vereinfachen.

3.1 Basisfunktionalität

Nachdem wir im letzten Abschnitt die Erwartungen des Benutzers an einen Editor herausgearbeitet haben, gilt es nun, diese bei der Konzeption des zu realisierenden Editors für die Datenmodellierung weitestgehend miteinzubeziehen.
Die sachgerechte Behandlung von Tastaturereignissen muß durch spezielle Kontrollprogramme (*Tastaturtreiber*) gewährleistet werden. Sie haben auch für eine Belegung von Sondertasten, die das Kopieren, Ausschneiden und Einfügen von Zeichenketten, sowie das Wiederholen und Rückgängigmachen von Aktionen unterstützen, Sorge zu tragen. Entsprechende Kontrollfunktionen sind auch für den sachgerechten Gebrauch einer Maus zu installieren.
Dokumente sollen geladen und gespeichert werden, wobei auch ein Wechseln des Verzeichnispfades mit geringem Aufwand möglich sein sollte. Möchte der Benutzer eine Klasse laden, so ist es ihm insbesondere bei großen Applikationen kaum zuzumuten, sich sämtliche Dateinamen, sowie die Verzeichnisse, in denen sie stehen, zu merken. Vielmehr erwartet er eine Liste von Klassen, aus denen er gezielt die von ihm gesuchte öffnen kann. Auch das Wechseln des Verzeichnispfades sollte mit einem Minimum an Aufwand durchführbar sein. Zur Realisierung dieser Funktionalität sollte der Editor ein Datei-Auswahl-Fenster zur Verfügung stellen, das Aktionen dieser Art so einfach wie möglich gestaltet. Systemmeldungen und Entscheidungsmöglichkeiten, sollten sofort beantwortet werden, so daß sie nicht ignoriert werden können. Sie dienen dazu, den Benutzer durch das Programm zu führen, ihn über den Stand der Dinge zu informieren und Warnungen und wertvolle Hinweise zu geben, die das ungewollte Auftreten von Fehlern eingrenzen.
Während das soeben erstellte Anforderungsprofil sich auf einen allgemeinen Texteditor be-

zogen hat, muß nun auch darüber nachgedacht werden, welche Funktionalität im Kontext einer Datenmodellierung erforderlich ist und daher realisiert werden muß. Eine sinnvolle Weiterverarbeitung von Klassendefinitionen kann zum Beispiel nur gewährleistet werden, wenn deren syntaktische Korrektheit nachgewiesen werden kann. Es sind folglich adäquate Analyseroutinen zur grammatikalischen Überprüfung bereitzustellen. Daten, die direkt in Dateien abgelegt werden und somit der Gefahr einer Manipulation durch das Betriebssystem ausgesetzt sind, stellen nicht immer die beste Möglichkeit der Datensicherung dar. Im Kontext von Datenbanken ist es daher üblich, sie in Datenwörterbüchern zu verwalten, die über ein zur Datenbank gehörendes Objektspeichersystem persistent gehalten werden. Um auch Vergleiche mit bereits existierenden Klassen durchführen zu können, muß auch darüber nachgedacht werden, mehrere, voneinander unabhängige Editoren gleichzeitig zu öffnen.

3.2 Ein Texteditor mit Komponenten

Bei der Konzeption eines Editors zur Entwurfsunterstützung ist es zweckdienlich, dessen Struktur in Anlehnung an die gegebene Syntax anzupassen. Wie das im Falle von oml geschehen kann, soll im folgenden beschrieben werden. Eine Klassendefinition ist kein monolithischer Block, sondern läßt sich in mehrere Komponenten untergliedern. Der Texteditor muß daher der speziellen Struktur einer Klasse Rechnung tragen, um dem Benutzer die Arbeit so weit wie möglich zu erleichtern. Die Gestaltung eines Klasseneditors muß daher in enger Anlehnung an die Syntax der verwendeten Datenbankprogrammiersprache stehen. Eine STYLE-Klasse besteht aus folgenden Teilen:

- dem Klassennamen: Er wird durch das Schlüsselwort "**Class**" gefolgt von einer Eingabezeile für den Benutzer kenntlich gemacht,
- einer Beschreibung der Klassenstruktur, welche durch das Schlüsselwort "**Structure**" eingeleitet wird,
- der Definition etwaiger benannter Integritätsbedingungen, die die Klasse tangieren. Sie erhält das Schlüsselwort "**Constraints**",
- einer Komponente, die Aufschluß über die Einordnung der Klasse in das gesamte Schema gibt. Sie gibt Auskunft über bestehende Subtyp-Supertyp-Beziehungen, sowie ererbte Attribute und ist durch den Bezeichner "**Specialization**" gekennzeichnet.
- die Klassendefinition wird durch ein "**End**" abgeschlossen.

Für eine sinnvolle Klassendefinition ist neben dem Klassennamen und der Ende-Marke eine Strukturbeschreibung erforderlich. Für generische Klassenspezifikationen stehen außerdem die Komponenten

- **Parameter**

- **Instantiation** und
- **Methods**

zur Verfügung, von denen jedoch zum derzeitigen Stand der Entwicklung noch kein Gebrauch gemacht werden kann. Das folgende Beispiel soll die Definition einer Klasse unter STYLE veranschaulichen:

```
Class PackageTour

Specialization

  isSubClassOf Tour

Structure

  Attributes
    key tourNo :Int
    constant key country :▷Country
    travelTime :TravelTime
    constant area :▷Area
    constant town :▷Town
    constant hotel ▷Hotel
    flights :[ forth :▷Flight, back :▷Flight ]

Constraints

  static
    HotelFlight: (this.hotel.area = this.flights.forth.route.destAirport.area) ∧
                 (this.hotel.area = this.flights.back.route.depAirport.area)
    TraveltimeOk: ((this.travelTime.from after this.hotel.travelTime.from) ∧
                   (this.travelTime.from after this.flights.forth.travelTime.from)) ∧
                  (((this.travelTime.from after this.flights.back.travelTime.from) ∧
                    (this.travelTime.till before this.hotel.travelTime.till)) ∧
                   ((this.travelTime.till before this.flights.forth.travelTime.till) ∧
                    (this.travelTime.till before this.flights.back.travelTime.till)))

End
```

Abbildung 3.1: Beispiel für eine Klassendefinition

3.3 Funktionen zur selektiven Größenadaption

Bei der Spezifikation einer neuen Klasse werden nicht notwendigerweise alle oben genannten Komponenten benötigt. Es ist daher wünschenswert, wenn der Benutzer selbst entscheiden kann, welche Komponenten er auf dem Bildschirm angezeigt haben möchte. Dies führt zu einer besseren Lesbarkeit der Klassendefinition. Diese Funktionalität stellen die Routinen "**Show**" und "**Hide**" zur Verfügung.
Mit Hilfe der in jedem Popup-Menü zur Verfügung stehenden "Hide"-Funktion wird die gerade aktive Klassenkomponente vom Bildschirm entfernt. Der freigegebene Platz wird

proportional unter den verbleibenden n Komponenten aufgeteilt, so daß sich jede Einheit vertikal um $1/n$-tel vergrößert. Um eine Klassenkomponente im Editor wieder anzuzeigen, steht die "Show"-Funktion des Hauptmenüs zur Verfügung. Nach ihrer Aktivierung öffnet sich ein Submenü, daß alle mit der "Hide"-Funktion entfernten Komponenten enthält. Der durch den Benutzer ausgewählte Klassenbestandteil wird auf dem Bildschirm angezeigt und eine automatische Größenanpassung der nun sichbtaren Komponenten wird analog zur oben beschriebenen Adaption durchgeführt.
Die "**Zoom**"-Funktion, die ebenfalls im Popup-Menü jeder Komponente vorhanden ist, unterstützt den Benutzer bei der Erstellung einer neuen Klasse. Mit ihrer Hilfe wird die ausgewählte Komponente auf die maximale Größe transformiert, so daß sie den Inhalt des Fensters genau ausfüllt. Die Inanspruchnahme dieser Funktion erleichtert dem Anwender nicht nur den Überblick über die gerade zu bearbeitende Komponente; sie erspart ihm oft auch ein "Scrollen" über den Inhalt, der nun in der Regel vollständig sichtbar ist. Das Auslösen der **Unzoom**-Funktion stellt den Zustand vor dem "Zoomen" wieder her.
Für die vertikale Veränderung der Fenstergröße ist eine Funktion "Change Height" bereitzustellen. Eine Veränderung der Fensterhöhe führt automatisch auch zu einer Modifikation des Platzes, der den Komponenten zur Verfügung steht. Diese kann beispielsweise sinnvoll sein, wenn alle Komponenten der Klasse nur einen sehr geringen Füllungsgrad aufweisen. Eine Anpassung der Fensterbreite erscheint dagegen nicht sinnvoll, da eine Klassendefinition aus Gründen der Lesbarkeit eine gewisse horizontale Ausdehnung haben muß.

3.4 Funktionen zur Lese- und Modellierungsunterstützung

Da dieser Editor zur Modellierung von Klassen eingesetzt werden soll, die in einem übergeorneten Kontext, dem Schema, in Beziehung zueinander stehen, ist es wichtig, Möglichkeiten zu schaffen, die deren Affinität zueinander erfassen können und dem Benutzer verfügbar machen. Diese Möglichkeiten sollen im folgenden kurz vorgestellt werden.

3.4.1 Menüs

Menüs erlauben es dem Benutzer Auswahlen aus einer vorgegebenen Menge möglicher Optionen vorzunehmen. Ist die Kardinalität dieser Menge sehr groß, so eignet sich eine rollbare Liste (siehe weiter unten) für die Bewältigung dieser Aufgabe besser.
Wird das Menü nach Aktivierung eines (Menü)-Knopfes geöffnet, so spricht man von einem *Pulldown-menu*; öffnet es sich dagegen beim Drücken der Menütaste der Maus, so handelt es sich um ein *Pop-up Menü*. Die Menüeinträge selbst bestehen aus einem kurzen, die mit ihm assoziierte Funktion erklärenden Namen, der - nach seiner Aktivierung - invertiert dargestellt wird. Je nach Art des Eintrags wird dann ein weiteres Menü (*Submenü*) geöffnet, oder die für diesen Eintrag vorgesehene Aktion ausgeführt.

3.4.2 Kommentarfenster

Nicht immer sind die Integritätsbedingungen und Methoden der modellierten Klassen für jeden Benutzer einsichtig und verständlich. Denn häufig können kompakt formulierte Definitionen und Anweisungsfolgen sehr kryptisch wirken. Der Editor muß dem Benutzer daher die Möglichkeit geben, erläuternde Kommentare in der Klassendefinition unterzubringen, die mit einem Blick eingesehen werden können und einen Beitrag zum Verständnis der Anwendung leisten. Dies sollte nicht in den Komponenten der Klasse selbst geschehen, sondern in eigenen Pop-up Fenstern, die bei Bedarf aus lokalen Pop-up Menüs heraus geöffnet werden. Als nützlich erweist es sich, wenn jede Integritätsbedingung explizit mit einem Namen versehen wird, so daß auf sie bezug genommen werden kann.

3.4.3 Übersichtsanfragen

Die Modellierung einer nicht-trivialen Datenbankanwendung kann schnell zu einem hohen Grad an Komplexität führen. Sie verlangt dem Programmierer neben Detailwissen über die Klassen selbst auch eine Akquisition von Strukturwissen, das heißt von Wissen über die Beziehungen und Interaktionsmöglichkeiten der Klassen untereinander, ab. Daraus leitet sich die zentrale Zielforderung ab, ein Instrumentarium bereitzustellen, das den Überblick über die entstehende Applikation für den Entwickler so leicht wie möglich gestaltet. Ein benutzerfreundlicher Editor muß Anfragen bereitstellen, die den aktuellen Informationsbedarf befriedigen können. Auf der Ebene der Klassen tangiert dies zum einen die Frage der Klassenvollständigkeit, zum anderen die Frage nach allen Klassen, die in einer Subklassen-Superklassen Relation involviert sind. Eine Klassendefinition gilt als vollständig, wenn sie keine ungebundenen Referenzen auf Typen oder Klassen enthält.
Der STYLE-Klasseneditor stellt diese Funktionalität zur Verfügung. Ferner sind bei der Klassenmodellierung die Strukturen (transitiv) referenzierter Klassen, sowie referenzierender Klassen von Interesse. Diese werden zum Beispiel benötigt, wenn man Integritätsbedingungen spezifizieren will. Ensprechende Anfragemöglichkeiten sind auch hier vorgesehen worden.

3.4.4 Anwendungssensitive Vernetzung von Klasseneditoren

Basierend auf den soebend beschriebenen Übersichtsanfragen läßt sicht leicht ein weiterer Gewinn an Bedienungskomfort erzielen, wenn die Klassen, die das Anfrageprädikat erfüllen nicht nur namentlich am Bildschirm angezeigt werden, sondern wenn zusätzlich noch ein Dienst bereitgestellt wird, der das Auswählen und Anzeigen einer dieser Klassen ermöglicht. Diese Fähigkeit spiegelt den Wunsch des Benutzers wieder, ausgehend von dem Ergebnis einer Anfrage nähere Informationen über die Klassen zu erlangen, die in einer Beziehung mit der aktuellen Klasse stehen. Dies geschieht häufig mit der Intention, Modifikationen vorzunehmen, die den evolutorischen Prozeß eines Schemas vorantreiben, oder die die Wahrung oder Wiederherstellung seiner Konsistenz unterstützen. Doch auch dann, wenn kein Handlungsbedarf angezeigt ist, sorgt eine solche Möglichkeit dafür, das Beziehungsgeflecht der Klassen transparenter zu gestalten.

Kapitel 4

Realisierung der Fensteroberfläche

Nachdem im vorigen Kapitel der konzeptuelle Rahmen für den Aufbau des Klasseneditors abgesteckt worden ist, beschäftigt sich dieser Abschnitt mit dessen konkreter Verwirklichung. Dazu wird im folgenden zunächst ein kurzer Überblick über die Konzepte der verwendeten Programmiersprache Quest gegeben, ehe dann nachfolgend auf die Möglichkeiten des graphischen Benutzerwerkzeugs und seine Einbindung in die Programmiersprache eingegangen wird.

4.1 Die polymorphe Programmiersprache Quest

Die Implementation des Klasseneditors erfolgte in der Programmiersprache Quest. *Quest* ist eine funktionale Programmiersprache mit strikter Typisierung, die polymorphe Konzepte zur Verfügung stellt. Im folgenden soll diese Sprache kurz vorgestellt werden. Ich beschränke mich dabei auf die wesentlichen Konzepte und abstrahiere von Basistypen und elementaren Kontrollstrukturen, deren Syntax anderen Programmiersprachen sehr ähnlich ist. Dem interessierten Leser sei [MMN92] empfohlen. Der Übersichtlichkeit halber werden im folgenden in allen Programmbeispielen die Schlüsselworte, das heißt alle Worte, die zum Sprachumfang von Quest gehören, in Fettdruck dargestellt.

4.1.1 Statische und strenge Typisierung

Um die Vorteile von streng typisierten gegenüber untypisierten Systemen herauszukristallisieren, sei hier zunächst kurz auf deren Unterschiede eingegangen: In einem untypisierten System sind die untypisierten Objekte ungeschützt und die ihnen zugrundeliegende Repräsentation ist für jedes Objekt von außen sichtbar. Dies kann zu Manipulationen führen, die nicht Intention des Programmierers waren. Wird beispielsweise ein numerischer Wert (*Integer*) mißbräuchlich als Zeiger (*Pointer*) benutzt, kann dies zu ungewünschten Modifikationen der Daten führen oder einen fehlerhafte Programmausführung zur Folge haben.. Um Typverletzungen dieser Art zu verhindern, wird das Konzept der statischen Bindungen eingeführt. Hierbei werden semantische Objekte, wie Konstanten, Operatoren, Variablen

und Funktionssymbole, an benutzerdefinierte Namen gebunden [Mat92]. Dadurch werden die Möglichkeiten, diese Objekte zu modifizieren und zu referenzieren, eingeschränkt und Typkonflikte vermieden. Ist die vom Programmierer gegebene Typinformation nur unzureichend oder gar nicht vorhanden, so sorgt das Typinferenzsystem von Quest für die entsprechende Bindung. Sind alle Ausdrücke und Variablen bereits zum Übersetzungszeitpunkt gebunden, so spricht man von *statischer Typisierung*. Statische Typisierung garantiert das typkonsistente Ausführen von Programmen, da Typinkonsistenzen bereits beim Kompilieren erkannt werden. Typkonvertierungen können somit nicht durchgeführt werden.
Häufig ist die Forderung nach statischer Typisierung zu restriktiv. Wünschenswert ist es vielmehr, die Typkonsistenz zum Zeitpunkt der Übersetzung auch dann zu garantieren, wenn die statische Bindung einiger Typen noch nicht bekannt ist. Programmiersprachen, deren Typüberprüfung erst zur Laufzeit erfolgt, nennt man *streng typisiert*. Akzeptiert das Übersetzungsprogramm (*Compiler, Interpreter*) ein streng typisiertes Programm, so wird seine typfehlerfreie Ausführung garantiert. Aus dem bisher Gesagten folgt, daß jedes statisch getypte Programm auch streng typisiert ist, nicht aber umgekehrt. Während statische Typisierung die frühzeitige Typfehlererkennung erlaubt und zu einer höheren Laufzeiteffizienz führt, liegt der Vorteil der strengen Typisierung in der größeren Flexibilität, die auf Kosten der Performanz erreicht wird [CaWe85].

4.1.2 Polymorphismus

Die meisten herkömmlichen Programmiersprachen basieren auf dem Konzept, daß alle Funktionen, Prozeduren und deren Operanden *einen* eindeutigen Typ haben. Sprachen dieser Art bezeichnet man als *monomorph*. *Polymorphe* Funktionen hingegen zeichnen sich dadurch aus, daß ihre Aktualparameter mehr als einen Typ haben können. Es werden zwei Klassen von Polymorphismen unterschieden:

1. der universelle Polymorphismus. Von ihm spricht man immer dann, wenn eine gegebene Funktion aufgrund ihrer Struktur eine unendliche Menge von Typen akzeptiert, sofern diese eine gemeinsame Struktur haben. Man kann also zusichern, daß einige Werte, zum Beispiel polymorphe Funktionen, viele Typen haben können. Hierbei wird jeweils *derselbe* Programmcode für *verschiedene* zulässige Typen ausgeführt. Betrachten wir dazu folgendes Beispiel:

   (1) **let** makeIntIter(a:**Int** b:**Int** c:**Int**):Iter_T(**Int**) = iter.enum of a b c **end**
   (2) **let** makePolyIter(A::**Type** a:A b:A c:A):Iter_T(A) = iter.enum of a b c **end**

   Während das (monomorphe) Konstrukt (1) eine Iteration aus drei ganzen Zahlen erzeugt, ist der Elementtyp der Iteration (2) zum Kompilationszeitpunkt noch ungewiß. Erst wenn die Parameter der Funktion makePolyIter zur Laufzeit geeignet gebunden werden, kann gesagt werden, welchen Elementtyp die Iteration besitzt. Für die Instanziierung des Typparameters A eignet sich dabei jeder wohlgeformte Typ. Der universelle Polymorphismus läst sich in folgende zwei Ausprägungen weiter

unterteilen:

(a) Der **Inklusionspolymorphismus** erlaubt es Subtypbeziehungen und Vererbung zu modellieren. Er eröffnet somit die Möglichkeit zu objektorientierter Programmierung. Ein Objekt wird hierbei als Element mehrerer, nicht notwendigerweise disjunkter Klassen angesehen.

(b) Die Einheitlichkeit der Typstruktur beim **parametrischen Polymorphismus**, auch Generik genannt, wird durch die implizite oder explizite Angabe von Typparametern erreicht.

2. den Ad-hoc Polymorphismus. Er arbeitet nur auf einer endlichen Menge von verschiedenen Typen, die meist keine Gemeinsamkeiten aufweisen. Eine polymorphe Funktion dieser Art kann also als eine endliche Menge monomorpher Funktionen angesehen werden. Daraus folgt, daß unter Umständen für jedes Argument ein unterschiedliches Programmstück exekutiert wird. Der Ad-hoc Polymorphismus kommt in den folgenden beiden Varianten vor:

   (a) Beim Konzept der **Typanpassung** (*coercion*) wird der übergebene Typ programmintern derart konvertiert, daß er bei der Übergabe an die für ihn bestimmte Funktion keinen Typfehler auslöst.

   (b) Bei der **Typüberladung** (*overloading*) wird Polymorphismus dadurch erreicht, daß derselbe Variablenname benutzt wird, um Funktionen mit unterschiedlicher Semantik auszuführen. Die richtige Evaluierung erfolgt dann kontextsensitiv. So wird beispielsweise in der Programmiersprache Pascal der '+'-Operator zum Addieren von ganzen Zahlen und Gleitkommazahlen, zum Konkatenieren von Zeichenketten, sowie zum Hinzufügen eines Elementes in eine Menge benutzt.

4.1.3 Funktionen höherer Ordnung

In Quest sind Funktionen *Objekte erster Klasse*. Es können daher insbesondere Funktionen als Funktionsargumente übergeben werden und auch der Rückgabewert einer Funktion kann wiederum eine Funktion sein. Dieser Abstraktionsmechanismus trägt zu einer deutlichen Erhöhung der Ausdrucksmächtigkeit einer Programmiersprache bei. Analog zu [AbSu85] betrachten wir folgendes Beispiel: Die nachstehenden Funktionen sollen jeweils die Werte im Intervall von a bis b addieren.

```
let rec summeZahlen(a,b : Int) : Int =
      if a > b then 0
      else a + summeZahlen(a+1 b)
      end

let rec summeQuadrate(a,b : Int) : Int =
      if a > b then 0
```

```
        else a * a + summeQuadrate(a+1 b)
        end
```

Diese beiden Funktionen verfügen über eine weitgehend gleiche Struktur und man kann sich weitere Funktionen (Summe der Primzahlen, Kehrwerte ...) vorstellen, die in dieses Schema passen. Hier kann daher das Konzept der Funktionen höherer Ordnung nutzbringend eingesetzt werden. Die Funktion summeAllgemein erwartet neben den beiden Intervallgrenzen auch eine Funktion, die die Berechnungsvorschrift angibt, sowie eine Funktion, die die nächste, zur Berechnung heranzuziehende Zahl bestimmt

```
let rec summeAllgemein(a,b : Int term,nächsteZahl:All(:Int) Int) : Int =
        if a > b then 0
        else
                term(a) + summeAllgemein(nächsteZahl(a) b term nächsteZahl)
        end
```

Geeignete Funktionen für term und nächste Zahl wären:

```
let term(a:Int) : Int =
        a*a         bzw. a

let nächsteZahl(a:Int) : Int =
        a+1
```

4.1.4 Optionstypen

Objekten der realen Welt können nicht immer eindeutige Attribute zugeschrieben werden. Häufig fehlen bestimmten Objekten einer Klasse Attribute oder ein Attribut kann durch ein anderes substituiert sein. Zur Modellierung solcher varianten Datenstrukturen stellt Quest *Optionstypen* zur Verfügung. Ihre Benutzung soll exemplarisch an der Definition eines Menüeintrags verdeutlicht werden:

```
Def Menüeintrag =
   Option    with name:String end
      nurEintrag with end
      mitAktion with funktion:Action_T end
      mitSubmenü with  subMenu:T end
   end
```

Menüeinträge können folglich in drei Ausprägungen auftreten:

1. Der Eintrag besitzt nur einen Namen

2. Der Eintrag ist benannt und nach seiner Aktivierung wird eine mit ihm assoziierte Funktion ausgeführt

3. Die Aktivierung des benannten Menüeintrags öffnet ein weiteres Auswahlmenü

Der Zugriff auf die Varianten eines Optionstyps erfolgt durch die Mehrfachalternativanweisung (*case-statement*):

case menüeintrag:Menü_Menüeintrag
  when nurEintrag **then** {*drucke Namen des Eintrags aus*}
  when mitAktion **with** fun **then** {*führe fun aus*}
  when mitSubmenü **with** menü **then** {*öffne menü*}
end

4.1.5 Tupel und Tupelkomponentenselektion

Tupel sind ein Sprachinstrument zum Aggregieren von unterschiedlichen Typen, die unter einem neuen Namen subsummiert werden sollen. Ein Beispiel:

Let Auto = **Tuple** typ:**String** baujahr:**Int** allrad:**Bool end**
let bus:Auto =
  tuple
    let typ = "VW Transporter"
    let baujahr = 1981
    let allrad = false
  end

Ein Zugriff auf die Komponenten des Tupels erfolgt über eine Punktnotation, mit der die richtige Referenz gesetzt wird.

drucke(bus.baujahr)

liefert die Jahreszahl 1981 als Ergebnis.

4.1.6 Abstrakte Datentypen

Ein Datentyp besteht aus einer Repräsentation seiner Datenstruktur, sowie einer Menge von Operationen oder Algorithmen, die auf ihm definiert sind. Letztere lassen sich in Konstruktionsoperationen und Basisoperationen unterteilen. Während bei "herkömmlichen" Datentypen deren Struktur nach außen hin sichbtar ist, die Methoden jedoch in der Regel als eingebaute (und damit versteckte) Funktionen zur Verfügung gestellt werden, gestaltet sich das Konzept bei den abstrakten Datentypen etwas anders: In einer Schnittstelle (*interface*) werden dem Benutzer die Signaturen, das heißt die Aktualparameter und Rückgabewerte aller Kontruktoren, Mutatoren und Selektoren zuzüglich einer kurzen Beschreibung,

die das Verhalten der Operation beschreibt, zur Verfügung gestellt, die er benutzen darf. Die konkrete Implementation dieser Operatoren, sowie die Repräsentation des eigentlich Datentyps bleiben ihm jedoch verborgen. Sie werden in einer Datenkapsel versteckt (Fachtermini: *information hiding, data encapsulation*). Der Benutzer bekommt also nur eine Zusicherung, daß die von ihm aufgerufene Methode ein bestimmtes Verhalten hervorruft, der dafür benutzte Algorithmus ist für den Außenstehenden jedoch unbekannt.
Die Realisierung eines abstrakten Datentyps in der Programmiersprache TYCOON [1] soll hier exemplarisch am Beispiel eines Bankkontos verdeutlicht werden:

```
interface Konto
export
        Konto : TYPE
        neuesKonto(inhaber:String betrag:Int) :Konto
        kontoLeer(k:Konto) :Bool
        einzahlen(k:Konto betrag:Int) :Konto
        abheben(k:Konto betrag:Int) :Konto
        KontoinhaberBestimmen(k:Konto) :String
end

module konto:Konto

export
        Let Konto = Tuple var
                kontonr:Int
                guthaben:Int
                inhaber:String
        end
        let neuesKonto(inhaber:String betrag:Int) :Konto =
                tuple let var
                        größte existierende Kontonummer+1
                        betrag
                        inhaber
                end
        let kontoLeer(k:Konto) :Bool =
                if k.guthaben is 0 then true else false
        ...
end
```

[1] Die Programmiersprache TYCOON ist der Nachfolger von Quest und orientiert sich daher stark an deren Konzepten.

4.1.7 Subtypisierung

Quest stellt auch Möglichkeiten zur Subtypisierung bereit. Ein Typ A ist Subtyp eines Typs B, wenn er mindestens die Spezifikation von B erfüllt. Die Spezifikation von A darf also auch spezieller als die von B sein. Besteht eine solche Beziehung, so können Funktionen, die auf B angewendet werden, auch mit Werten des Typs A erfolgreich evaluiert werden. Eine Subtypisierung zwischen Funktionstypen führt zur sogenannten *Kontravarianzregel*: Ein Funktionstyp mit Signatur S und Ergebnistyp A ist ein Subtyp eines Funktionstyps mit Signatur S' und Ergebnistyp B, genau dann, wenn A Subtyp von B und S' Subsignatur von S ist [Hoa69]. Ein Beispiel:

```
Let A =
   Tuple
      a :Int
   end

Let B =
   Tuple
      a :Int
      b :Bool
   end

Let SuperFun = Fun(:B) :Ok
Let SubFun = Fun(:A) :Int
```

Die Signatur des Typs B ist spezieller als die des Typs A; seine extensionale Menge ist jedoch kleiner. Daher ist der Typ B ein Subtyp des Typs A. Ebenso gilt, daß die Funktion SubFun ein Subtyp der Funktion SuperFun ist, da die oben beschriebene Kontravarianzregel gilt.
Das Konzept der Subtypisierung leistet einen wesentlichen Beitrag zur Erweiterbarkeit von Softwaresystemen, insbesondere in persistenten Umgebungen [Mat92]. Systeme mit mannigfaltigen Subtypbeziehungen können daher einen Beitrag dazu leisten, generische Algorithmen zu entwickeln, die dann unverändert auch auf spezielleren Typausprägungen arbeiten und von deren für die Anwendung irrelevanten Details abstrahieren.

4.2 Graphische Bausteine

Wie bereits eingangs erwähnt zeichnen sich benutzerfreundliche Programme auch durch ihre leichte Handhabbarkeit aus. Diese wird unter anderem durch den Einsatz einer graphischen Benutzeroberfläche erreicht. Auch dieses Softwareprodukt konnte sich daher nicht der Notwendigkeit entziehen, entsprechende Möglichkeiten bereitzustellen. Als Entwurfs-

werkzeug fungierte hierbei NeWS, ein objektorientiertes System, daß den Aufbau von komplexen Bildschirmobjekten nach dem Baukastenprinzip (*Toolkit*) unterstützt. Da es sich um ein objektorientiertes System handelt, sind sämtliche Objektklassen hierarchisch angeordnet und nutzen die Möglichkeit der Vererbung aus. Eine der grundlegendensten Klassen ist die Klasse Canvas (*dt. Leinwand*), die Superklasse fast aller anderen Klassen ist. Ein Canvas ist eine begrenzte Fläche, in der Anwendungsinformationen dargestellt werden können. Flächen dieser Art dürfen geschachtelt werden oder sich mit anderen Flächen überschneiden. Zum näheren Kennenlernen dieser Hierarchie dient die folgende Abbildung, die alle Objektklassen in einer baumartigen Struktur wiedergibt. Nachfolgend möchte ich zunächst jene graphischen Objekte des NeWS-Toolkits vorstellen, aus denen der STYLE-Klasseneditor zusammengesetzt wurde, ehe ich dann näher auf die Einbindung von dessen Funktionalität in die Programmiersprache eingehe.

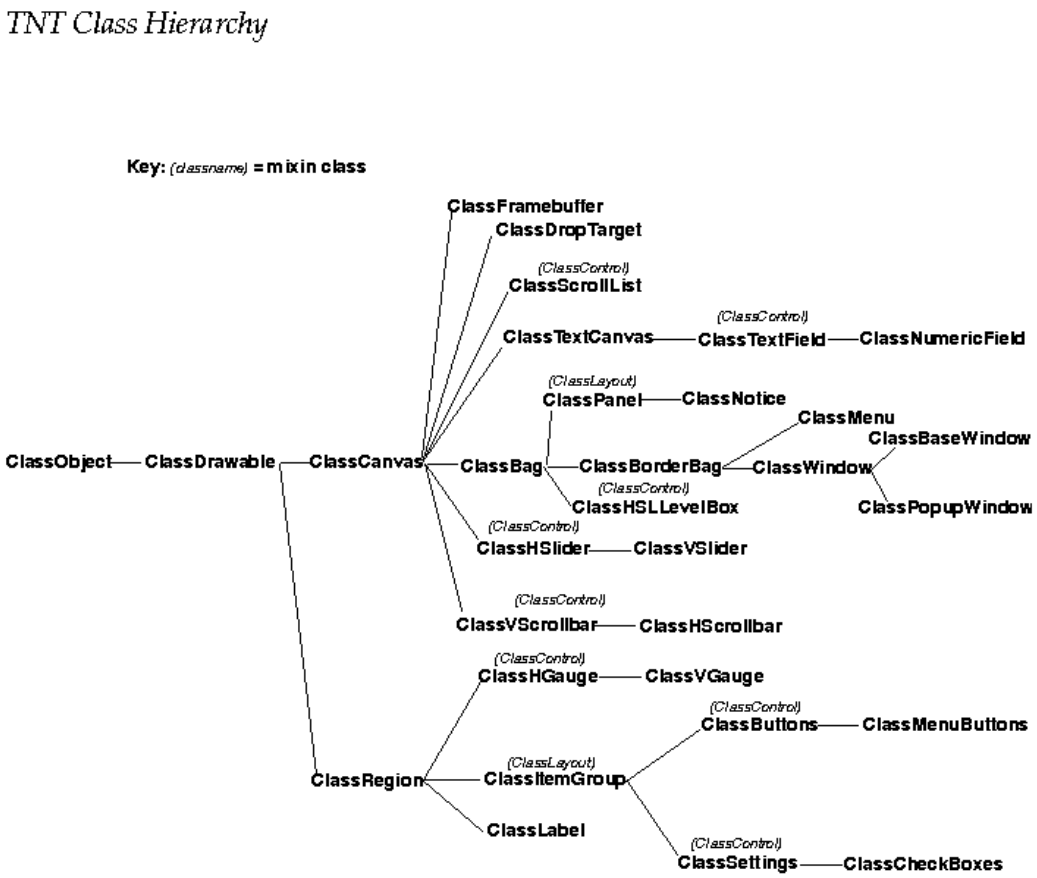

Abbildung 4.1: Die TNT-Klassenhierarchie

4.2.1 Fenster

Man unterscheidet folgende zwei Arten von Fenstern:

- Basisfenster
  Dem **Basisfenster** (*Base Window*) kommt eine zentrale Bedeutung zu, da es der Ausgangspunkt für jede gestartete Anwendung ist. Es besteht in seiner Grundausführung aus folgenden Elementen:

  1. einer **Überschrift** (*Header*), die den Namen der gestarteten Applikation enthalten kann,
  2. dem **Standardmenu**, das das Schließen (*close*), Vergrößern (*full size*), Restaurieren (*refresh*) und Verlassen (*quit*) des Fensters erlaubt,
  3. einem **Menüknopf** (*Menu Button*) zum Schließen des Fensters
  4. **beweglichen Ecken** (*resize corners*), die ein wahlfreies Skalieren des Fensters ermöglichen.
  5. **Fußzeilen** (*footer*), die zum Anzeigen von Zustands- oder Fehlermeldungen eingesetzt werden können,
  6. einem **Kontrollbereich** (*control area*), der Knöpfe oder Menüknöpfe aufnehmen kann und
  7. der eigentlichen **Fensterscheibe** (*pane*), die die Anzeige und Manipulation der Anwendungsinformation erlaubt. Ihr ist oftmals ein Rollbalken (siehe unten) zugeordnet.

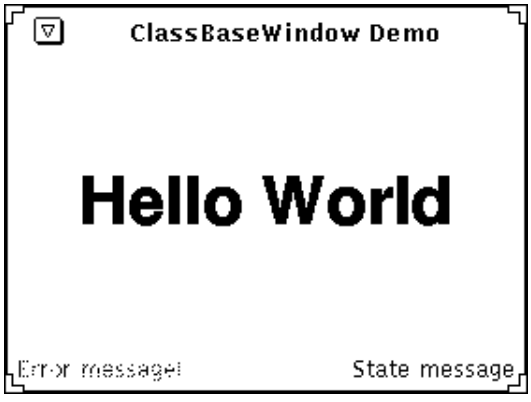

Abbildung 4.2: Ein einfaches Basisfenster mit Fußzeilen

Wird ein Basisfenster geschlossen, so wird es als quadratische Ikone (*icon*) am unteren Bildschirmrand dargestellt.

- Aufklappfenster
  Aufklappfenster (*Pop-up Windows*) werden vom Basisfenster einer Applikation generiert, um Systemparameter vom Benutzer zu erfragen (beispielsweise die Eingabe eines Dateinamens), Eigenschaften von Anwendungen, Objekten oder Fenstern zu verändern (zum Beispiel zur Umstellung der Systemuhr auf Sommerzeit) oder um in unklaren Situation textuelle Hilfestellung zu geben. Aufklappfenster besitzen immer eine Stecknadel (*pin*), mit der sie "festgesteckt" werden können. Das Schließen dieses Fensters führt automatisch zu seiner Entfernung vom Bildschirm; eine Ikone wird nicht angezeigt.

4.2.2 BorderBags

BorderBags sind spezielle Kontrolloberflächen, die dazu gedacht sind, bis zu fünf Klienten zu verwalten. Ein Klient wird in der Mitte postiert, während die anderen vier Objekte an seinen vier Kanten (Nord- Ost- Süd- und Westkante) plaziert werden können. Eine typische Anwendung eines BorderBags ist zum Beispiel ein Texteditor, der im Norden eine Knopfleiste und im Osten einen Rollbalken besitzt. Ex definitione ist auch jedes Basisfenster ein BorderBag, da Fenster eine Subklasse von BorderBags sind.

4.2.3 Rollbalken

Mit Hilfe eines **Rollbalkens** (*Scrollbar*) kann die Sicht auf den Inhalt eines Fenster verändert werden. Dies ist unter Umständen erforderlich, weil nicht immer die gesamte Information im Fenster dargestellt werden kann. Rollbalken können vertikal oder horizontal installiert werde

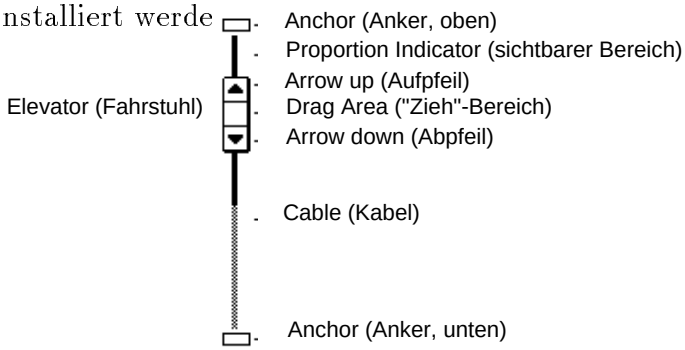

Abbildung 4.3: Ein Rollbalken

Mit ihrer Hilfe kann der Inhalt eines Fensters in verschiedenen definierten Granularitäten verschoben werden. So ist das seitenweise Blättern (durch "Anklicken" des Kabels), der Sprung zum Anfang oder Ende des Dokumentes (durch Aktivierung des Ankers) und das Verschieben um eine einzige Zeile (mit Hilfe der Pfeiltasten) ebenso möglich, wie das wahl-

freie Ansteuern eines bestimmten Bildschirmbereichs, was mit dem Schieber des Fahrstuhls erreicht werden kann.

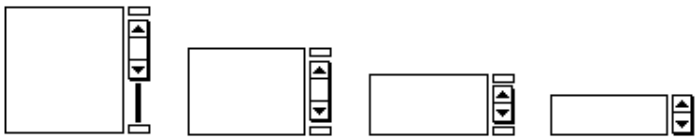

Abbildung 4.4: Anpassung von Rollbalken bei unterschiedlichen Fenstergrößen

4.2.4 Bezeichner

Ein **Bezeichner** (*Label*) ist eine Zeichenkette, die in Fettdruck am Bildschirm dargestellt wird und in erster Linie zum Benennen eines bestimmten Bildschirmabschnitts oder -objektes eingesetzt wird. Sie kann weder vom Benutzer modifiziert noch auf dem Bildschirm verschoben werden.
Beispiel:

Structure

4.2.5 Knöpfe und Menüs

Man unterscheidet zwei Arten von Knöpfen: "einfache" Knöpfe und Menüknöpfe. Einfache Knöpfe sind direkt mit einer Aktion verknüpft, die ausgelöst wird, wenn sie gedrückt werden. Menüknöpfe dagegen öffnen nach ihrer Aktivierung durch die Menütaste der Maus ein Menü, aus dem eine von zahlreichen Funktionen aufgerufen werden kann. Menüknöpfe befinden sich grundsätzlich im Kontrollbereich der Fenster und sind mit einem kleinen Dreieck versehen, welches angibt, in welche Richtung (nach unten oder nach rechts) das Menü aufklappt. Jeder Knopf ist mit einem Namen versehen, der Aufschluß über das Verhalten gibt, das die aufgerufene Aktion nach außen zeigen wird. Die Aktivierung eines Knopfes wird durch seine kurzzeitige inverse Darstellung verdeutlicht. Für häufig benutzte Aktionen kann ein Standardknopf (*default button*) oder eine Standardaktion (*default action*) definiert werden. Die Repräsentation am Bildschirm erfolgt dabei durch eine zusätzlich Umrandung. Wird ein Menüknopf mit der Auswahltaste der Maus "angeklickt", so wird sofort, sofern vorhanden, die Standardaktion aufgerufen, ohne daß das Menü erscheint.

4.2.6 Notizen

Eine **Notiz** (*notice*) wird benutzt, um Sicherheitsabfragen (z.B. beim Löschen einer Datei), Fehlermeldungen und Warnungen auf dem Bildschirm anzuzeigen. Sie ist ein spezielles Aufklappfenster und besteht aus folgenden Komponenten:

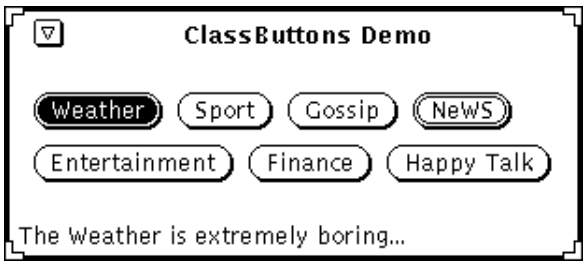

Abbildung 4.5: Knöpfe

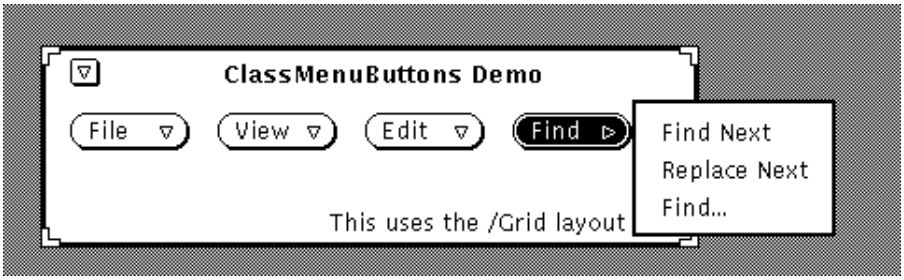

Abbildung 4.6: Eine Menüknopf mit aufgeklapptem Menü

1. einem Fenster, in dem sie angezeigt wird. Dieses Fenster wird von der Anwendung aufgerufen. Es besitzt weder eine Kopf- oder Fußzeile, noch eine Stecknadel (*pin*) und kann nicht verschoben werden. Die initiierende Anwendung wird so lange "eingefroren", bis der Benutzer eine der ihm angebotenen Wahlmöglichkeiten in Anspruch genommen hat.

2. einer kurzen Mitteilung an den Benutzer, die den augenblicklichen Zustand beschreibt. Dieser wird in Fettdruck auf dem Bildschirm dargestellt.

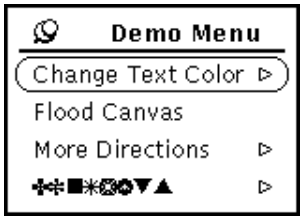

Abbildung 4.7: Ein Auswahlmenü

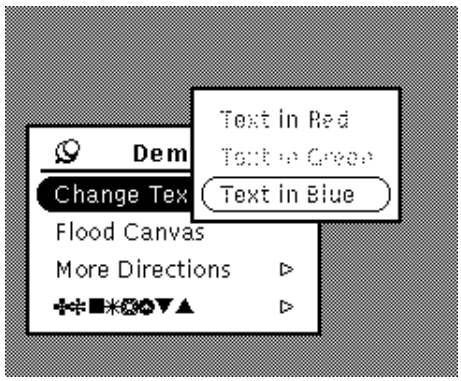

Abbildung 4.8: Ein Menü mit Submenü

3. einer begrenzten Anzahl von Knöpfen, die dem Benutzer Reaktionsmöglichkeiten aufzeigen; eine der angezeigten Aktionsmöglichkeiten wird als Standardaktion (*default button*) durch eine doppelte Umrandung gekennzeichnet. Der Mauszeiger springt beim Anzeigen der Notiz automatisch auf den Knopf, der diese Standardaktion repräsentiert.

TNT versieht Notizen außerdem mit einem dreieckigen Schatten, dessen Spitze auf die initiierende Applikation zeigt. Die Notiz selbst klappt immer zur Bildschirmmitte hin auf.

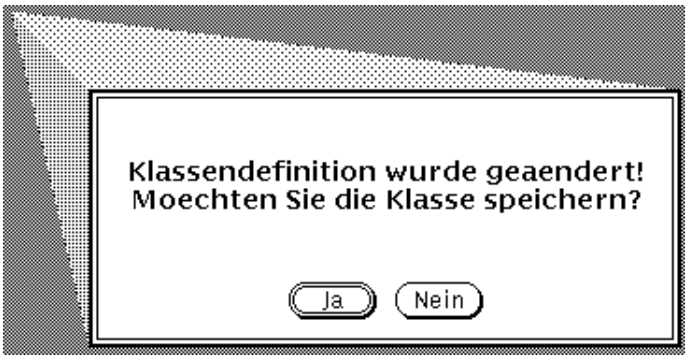

Abbildung 4.9: Eine Notiz in Form einer Systemwarnung

4.2.7 Rollbare Listen

Um eine große Liste von Auswahlmöglichkeiten effektiv zu verwalten können rollbare Listen (*Scrolling Lists*) eingesetzt werden. Sie bestehen aus folgenden Komponenten:

- einer rechteckigen Umrandung

- einer Liste von kurzen Zeichenketten (*Strings*), die es zu verwalten gilt
- einem Rollbalken mit dem durch die Liste "gerollt" werden kann
- einem eigenen Popup-Listenmenü

Je nach Konzeption der Liste steht dem Benutzer einer der folgenden Auswahlmodi zur Verfügung.

- Es muß genau ein Element der Liste ausgewählt werden. In diesem Fall spricht man von einer **exclusiven** Auswahl.
- Eine **Variation der exklusiven Auswahl** erlaubt die Selektion genau eines oder keines der Listenelemente.
- Ist es dagegen mögliche eine beliebige Anzahl (inklusive 0) Elemente der Liste auszuwählen, so handelt es sich um eine **nicht-exklusive Auswahl**.

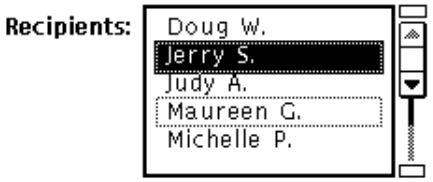

Abbildung 4.10: Eine rollbare Liste

4.2.8 Schieber

Ein Schieber(*slider*) stellt einfache Möglichkeiten zur Verfügung, Werte zu setzen oder sie sich anzeigen zu lassen. Er besteht aus einem Schiebeknopf (*drag box*), der analog zu einem Lautstärkeregler in einer Rille verschoben werden kann. Optional kann er an jedem Ende mit einem Zahlenwert versehen werden, um so die Bandbreite der einstellbaren Werte anzuzeigen. Außerdem können Schieber über eine Skala (*tick marks*) verfügen, die ein besseres Ablesen der eingestelten Werte erlaubt.

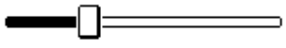

Abbildung 4.11: Ein einfacher Schieber

4.2.9 JotTexts, JotViews, Spans

Die Klassen Jot, JotText und JotView stellen Primitive zur Erstellung eines Texteditors zur Verfügung. Ihre Funktionalität korrespondiert mit dem Model-View-Controller Paradigma.
Ein JotText entspricht hierbei dem Datenobjekt (Modell), das die Information der Anwendung, in der Regel einen langen Text, enthält. Der JotText selbst ist nicht sichbtar. 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 sogenannte Spans auf dem JotText zu definieren. Ein Span beschreibt eine bestimmte (kurze) Region innerhalb eines JotTextes durch seine Ursprungsposition und seine Länge. Der Positionsparameter paßt sich automatisch an, wenn Zeichen vor dem Span eingefügt oder gelöscht werden. Ebenso verändert sich der Längenparameter des Spans automatisch, wenn innerhalb seiner Begrenzungen Zeichen hinzugefügt oder entfernt werden. Jeder JotText verwaltet eine Liste, der mit ihm verknüpften Spans.
Ein Beispiel:

Ein **Ball** (Span beginnt ab Position 5 mit Länge 4)
Einfügen des Wortes Feder ab Position 5
Ein **Federball** (Span beginnt ab Position 5 mit Länge 9)
Einfügen des Wortes " großer" ab Position 4
Ein großer **Federball** (Span beginnt ab Position 12 mit Länge 9)

4.2.10 Panel

Ein Panel ist eine Kontrolloberfläche die mehrere Klienten der Klasse Canvas oder Region enthalten kann. Das Format, mit dem diese Klienten plaziert werden, richtet sich nach der Art des Panels. Man unterscheidet folgende vier Typen:

1. Beim **Grid Panel** teilt der Benutzer den zur Verfügung stehenden Platz in Reihen und Spalten beliebiger Kardinalität ein und bestimmt, ob die Zeilen vor den Spalten gefüllt werden sollen oder vice versa. Neu hinzugefügte Klienten werden ans Ende der vom Panel verwalteten Klientenliste angehängt.

2. Das **Absolute Panel** erwartet vom Benutzer für die Plazierung jedes neuen Klienten ein x-y-Koordinatenpaar

3. Das **Spaced Panel** sorgt für eine automatische Plazierung der Klienten.

4. Das **Calculated Panel** erlaubt es dem Benutzer bei der Plazierung eines neuen Klienten Bezug auf bereits vorhandene Objekte und deren Position zu nehmen. So erlaubt es beispielsweise die Angabe

[/North {/South PREVIOUS POSITION}]

daß die nördliche Kante des neuen Objektes exakt mit der südlichen Kante des zuletzt plazierten Klienten abschließt, während

[/NorthWest {/NorthWest PARENT POSITION 10 -10 xyadd}]

bewirkt, daß die linke obere Ecke des neuen Objektes seinen Bezugspunkt in der linken oberen Ecke des Panels hat und von dort, um jeweils 10 Pixel nach unten und nach rechts verschoben, plaziert wird.

4.2.11 Interaktion zwischen NeWS und Quest

Die Bedeutung von Applikationen mit einer Kunden-Diensterbringer (*Client-Server*)-Architektur hat im Laufe der letzten Jahre stark zugenommen. Dieses Konzept drückt den Wunsch aus, die relativ teure Rechenleistung eines Zentralrechners (*mainframe*) durch ein flexibleres vernetztes System von Arbeitsplatzrechnern (*workstations*) zu ersetzen. Das netzwerkfähige Fenstersystem NeWS(*network extensible window-system*), das der vorliegenden Anwendung zugrunde liegt, bietet ideale Voraussetzungen zur Realisierung einer Client-Server-Architektur an. Das Fensterentwicklungssystem *The NeWS Toolkit*, im folgenden kurz TNT genannt, gliedert sich in zwei große Einheiten: Eine Schnittstelle zur *Programmiersprache C* und eine Schnittstelle zur dynamischen, interpretativen, kellerspeicherorientierten und zugleich geräteunabhängigen *Seitenbeschreibungssprache PostScript*. Diese scharfe Trennung zwischen der Programmiersprache auf der einen und dem Werkzeug zur Beschreibung der grafischen Repräsentation der Daten auf der anderen Seite ermöglicht gleichzeitig eine separate Distribution der zu bearbeitenden Programmsegmente an den Kundenprozeß und den Diensterbringer: Während sämtliche Berechnungen auf den Daten lokal vorgenommen werden, interpretiert der entfernte Server alle Aufgaben, die die Graphik tangieren. Um die Kommunikation zwischen Client und Server zu ermöglichen steht auf der Client-Seite ein Dienst zum Verbindungsauf- und abbau zum Server zur Verfügung. Es können drei Arten von Nachrichten verschickt werden:

1. Nachrichten, die keine Antwort benötigen
2. Nachrichten, die eine Rückmeldung brauchen und den laufenden Prozeß solange blockieren, bis diese zurückgesandt wurde (*synchroner Antwortprozeß*)
3. Nachrichten, die zwar eine Antwort erwarten, den laufenden Prozeß aber bis zu deren Eintreffen fortsetzen (*asynchroner Antwortprozeß*).

Übersteigt die vom Kundenprozeß auf die Verbindung gebrachte Informationsmenge die Abbarbeitungsgeschwindigkeit durch den Diensterbringer, so wird ein Teil der Information gepuffert und bei gesunkener Auslastungskapazität wieder freigegeben.
Die Kommunikation zwischen Klient und Diensterbringer erfolgt über einen simplen Strom

von ASCII-Zeichen. TNT stellt jedoch auch die Möglichkeit zur Interaktion über komprimierte Markierungen (*compressed tokens*) zur Verfügung. Eine solche Markierung setzt sich aus mehreren Ziffern zusammen. Es gibt drei Arten dieser Markierungen:

1. Mit Hilfe von **Typmarkierungen** (*typed tokens*) können Gleitpunktzahlen einfacher und doppelter Präzäsion (*floating point numbers*), ganzzahlige Werte (*Integer*), Fixpunktzahlen (*fixed point numbers*), kurze und lange Zeichenketten (*Strings*), Wahrheitswerte (*Booleans*), Felder (*Arrays*), gerasterte Bilder (*images*) und Marken (*Tags*) kodiert werden.

2. **Systemmarkierungen** (*system token*) erlauben die Kodierung systemdefinierter Objekte. Hierzu zählen in erster Linie PostScript-Sprachoperatoren und Operatorerweiterungen in NeWS.

3. Vom Benutzer definierte Objekte können mithilfe von **Benutzermarkierungen** (*user tokens*) versandt werden. Diese Markierungen werden in einer Liste verwaltet, die bis zu 65.536 Einträge erlaubt. Die Kodierung dieser Markierungen ist unterschiedlich: Die ersten 32 Objekte der Liste können mit einem Byte verschlüsselt werden, die nächsten 1024 Einträge benötigen zwei Byte zu ihrer Kodierung und eine Referenz auf die verbleibenden 64.480 Objekte schließlich erfordert drei Byte.

Ziel dieser Kodierung ist ein Geschwindigkeitsgewinn, da nun weniger Information über die Verbindung transportiert werden muß.
Funktionen, die den Verbindungsauf- und abbau, sowie das Betreiben dieser Verbindung ermöglichen, stellt das Questmodul "Wire" zur Verfügung, das der gleichnamigen Klasse des TNT entspricht.

4.2.12 Das Model-View-Controller Paradigma

Das Model-View-Controller Paradigma unterteilt eine Anwendung in drei Komponenten:

- das Datenobjekt (Modell), welches die Information der Anwendung enthält
- den View, der das mit ihm assoziierte Modell graphisch repräsentiert und
- den Controller, der als Schnittstelle zwischen der Eingabeeinheit und dem Modell fungiert und so dem Benutzer die Interaktion mit dem Modell über den View ermöglicht. Er beschreibt wie Eingaben das Modell und den View verändern.

Auch in NeWS findet sich dieses Paradigma wieder. Zur Veranschaulichung der Interaktion zwischen NeWS und den oben genannten Komponenten möge die folgende Abbildung dienen:

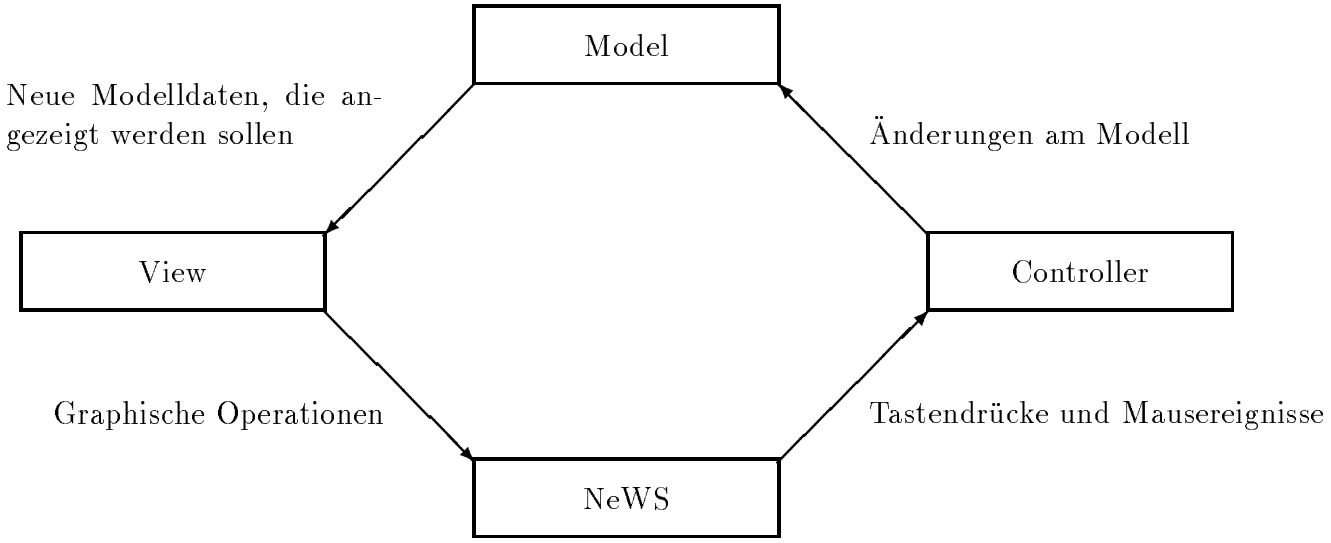

Abbildung 4.12: Der Informationsfluß zwischen Modell,View,Controller und NeWS

4.3 Die TNT-Quest-Schnittstelle

Nachdem man sich von der Brauchbarkeit des NeWS-Toolkits überzeugt hat, muß man sich unweigerlich die Frage stellen, wie man dessen Funktionalität in die Programmiersprache Quest einbinden kann. Denn NeWS erwartet seine Befehle ja ausschließlich im PostScript-Format. Die Lösung dieses Problems erfolgte durch die Schaffung einer eigenen Bibliothek, in der alle möglichen graphischen Funktionen aggregiert werden konnten. Somit wurde eine effiziente Möglichkeit gescaffen, den Dienst des externen NeWS-Servers in den Programmiersprachenkontext zu integrieren. Die Einbindung der Postscript-Aufrufe in die Programmiersprache sei hier exemplarisch an einigen Beispielfunktionen des Moduls Gui veranschaulicht. Dazu wird zunächst die Schnittstelle vorgestellt, ehe dann im Anschluß die Implementation der Funktion "installQuitAction" analysiert wird.

interface Gui

import
:Action
:Buttons
:Iter
:Canvas
:EventMgr
:Base
:Window
:Popup

```
:DisplayItem


export

(* —– Definitions ———————————— *)

Def DispItem = DisplayItem_T
Def GridItem = Buttons_GridItem

(* —– Base window ———————————— *)

newBase(client :Canvas_T label :DispItem
quit :Action_T em :EventMgr_T) :Base_T

installQuitAction(win :Base_T quit :Action_T) :Ok
(* Install a new quit action for the base window. *)

(* —– Popup window ———————————— *)

newSubwindow(client :Canvas_T label :DispItem
dismiss, unpin :Action_T
super :Window_T) :Popup_T

installDismissAction(win :Window_T dismiss :Action_T) :Ok
(* Install a new dismiss action for the popup window. *)

installPinAction(win :Window_T pin :Action_T) :Ok
(* Install a new pin action for the popup window. *)

(* —– Notice ———————————————— *)

displayNotice(win :Base_T
text :Iter_T(DispItem)
items :Iter_T(DispItem)
defaultButton :Int) :Int
(* Return the index of the selected button in the range 1..size(buttons).
Choose defaultButton=0 if there is no default. *)


end;

module gui :Gui
```

```
import
:EventMgr
:GuiRep
window :Window
base :Base
popup :Popup
object :Object
layout :Layout
buttons :Buttons
fmt :Fmt
iter :Iter
print :Print
control :Control
wire :Wire
action :Action
canvas :Canvas
notice :Notice
itemGroup :ItemGroup


export

(* —– Definitions ———————— *)

Let DispItem = Gui_DispItem
Let GridItem = Gui_GridItem

(* —– Miscellaneous ———————- *)

let installQuitAction(win :Base_T quit :Action_T) :Ok =
begin
(1) let quitTag = action.new(quit)
(2) object.setProperty(win "quitTag" fmt.int(quitTag))
(3) object.installMethod(win "QuitFromUser"
"/quitTag /property self send tagprint " <>
"/QuitFromUser super send")
end
...
end;
```

Zweck dieser Funktion ist es, die Aktion, die beim Verlassen eines Fensters aufgerufen wird (das Entfernen des Fensters von der Bildschirmoberfläche) zu erweitern. Die Inan-

spruchnahme dieser Funktion erweist sich als nützlich, da beim Verlassen des Editors noch ungespeicherte Änderungen vorhanden sein können, die ohne spezielle Maßnahmen verlorengingen.
Anweisung eins bindet die übergebene Aktion an einen Bezeichner namens quitTag. Anweisung zwei erweitert die Eigenschaften des übergebenen Fensters um diese Aktion und die dritte Anweisung schließlich beschreibt, wie sich das Fenster fortan im Falle einer "Quit"-Aktion verhalten soll. Hierbei wird die bereits existente Methode QuitFromUser überschrieben und erweitert. Die Argumente werden dem NeWS-Server in der angegebenen Reihenfolge geschickt und dort auf dem Stack abgelegt. Die Abarbeitung erfolgt dann nach jeder "send"-Anweisung.
Der Effekt ist dann folgender: Wird ein von Quest aus geöffnetes Fenster geschlossen, so wird dessen "QuitFromUser"-Aktion aktiviert."QuitTag" wird auf den Stack geschrieben, "property" prüft, welche Aktion mit dieser Marke assoziiert wurde. Diese wird dann beim nächsten "send" ausgeführt, ehe daß Fenster dann wie bisher geschlossen wird.

4.4 Aufbau des Editors

Nachdem nun ausgiebig auf die grafischen Objekte bezug genommen wurde, die für das Hauptfenster des Editors unerläßlich sind, wird nun auf dessen genauen Aufbau eingegangen werden.

4.4.1 Aufbau einzelner Klassenkomponenten

Hier sei zunächst der Aufbau einer Klassenkomponente beschrieben:
Jede Komponente wird durch ein Border Bag dargestellt. Dieses enthält als zentralen Klienten einen JotText, dem ein entsprechender JotView zugeordnet ist. Zur genauen Identifikation der Komponente wurde im Norden des Border Bags ein Bezeichner(Parameter, Instantiation, Specialization, Structure, Constraints oder Methods) eingefügt. Im Osten schließlich befindet sich ein Rollbalken, der den Inhalt des JotTextes kontrolliert. Aus Gründen der Übersichtlichkeit verfügt jede Komponente über eine Umrandung. Da nicht alle Aktionsmöglichkeiten die ganze Klasse respektive den Editor tangieren, wurde jede Komponente mit einem eigenen Menü versehen. Es enthält standardmäßig folgende Einträge:

1. Zoom: ermöglicht das Skalieren einer Komponente auf maximale Größe
2. Unzoom: macht einen Zoom-Vorgang rückgängig
3. Load: ermöglicht das gezielte Laden einer einzelnen Klassenkomponente
4. Save: speichert eine einzelne Klassenkomponente auf Datei
5. Hide: entfernt diese Komponente vom Bildschirm

Zusätzlich können folgende Einträge vorhanden sein:

- Parse: erlaubt die gezielte Syntaxüberprüfung einer einzelnen Komponente (nur bei Specialization, Structure und Constraints)
- Show Description: Gibt erläuternde Kommentare zu Integritätsbedingungen und Methoden
- Anfragen bezüglich referenzierender und referenzierter Klassen (nur Structure)

4.4.2 Einbindung der Komponenten in den Editor

Der Editor ist wie folgt aufgebaut:
Die Applikation befindet sich in einem Basisfenster. Dieses enthält als zentralen Klienten ein Border Bag, welches in der Mitte mit einem Panel versehen ist, das die zur Anzeige einer Klasse erforderlichen Komponenten beinhaltet. Im Norden schließt sich ein Bezeichner namens Klasse nebst einem JotText für den anzugebenden Klassennamen an. Im Süden befindet sich analog ein End-Bezeichner, dem erneut der Klassenname in einem JotText folgt. Dieser Klassenname kann nur gelesen werden. Er paßt sich automatisch dem weiter oben angegebenen Klassennamen an. Dieses ist ein gutes Beispiel dafür, wie auf einem JotText mehrere Sichten definiert werden können. Das Hauptfenster schließlich verfügt direkt unter seiner Titelleiste über fünf Menüknöpfe, deren Möglichkeiten hier kurz erläutert werden sollen:

1. Das Datei-Menü (*engl. File*):
   Es stellt Funktionen zum Laden und Speichern von Klassendefinitionen zur Verfügung. Diese können entweder in einer Datei oder in den Metadaten stehen.

2. Das Editor-Menü:
   Seine Funktionen ermöglichen das Öffnen eines neuen Editors, das Wiederanzeigen ausgeblendeter Komponenten, das Ändern der Fenstergröße, das komplette Löschen des Editors, sowie eine komfortable Latex-Dokumentenerzeugung, um sich Klassendefinitionen auch auf Papier ausdrucken zu lassen.

3. Das Anfragen-Menü (*engl. Queries*):
   Hier könnenAnfragen bezüglich der Klassenvollständigkeit, Subklassen und transitiv referenzierter Klassen gestellt werden.

4. Das Prozeß-Menü (*engl. processing*):
   Hier werden Funktionen zur grammatikalischen Analyse der gerade angezeigten Klassendefinition zur Verfügung gestellt, sowie eine Möglichkeit, aus diesen Definitionen Programmtext in einer anderen Programmiersprache(TYCOON) zu generieren.

5. Das Informationsmenü schließlich gibt Auskunft über Ort und Zuständigkeiten des Projektes, in dessen Rahmen dieser Klasseneditor entstand.

4.4.3 Aufruf des Klasseneditors

Die Exportschnittstelle des Editors, die dessen Funktionalität als Dienst nach außen zur Verfügung stellt, gestaltet sich wie folgt:

interface ClassEditor
export

filled(path : **String** desiredClass : **String** schemaName : **String**) :**Ok**

create(path : **String** schemaName : **String**):**Ok**

end;

Die Funktion create erzeugt einen neuen leeren Editor. Path gibt dabei den Verzeichnispfad an, in dem Klassendefinitionen, die nur als Datei existieren, gespeichert und später wieder geladen werden können. Programmintern sind selbstverständlich Methoden vorgesehen worden, diesen Pfad bei Bedarf zu ändern. Der Parameter schemaName erwartet den Namen eines Schemas. Dieser sollte im Schemawörterbuch eingetragen sein, da man sonst keine Anfragen, die das Schema betreffen, an die Klasse und deren Beziehungen zueinander stellen kann.
Beim Aufruf der Funktion filled wird zusätzlich zur Erzeugung des Editors noch eine bereits bestehende Klasse in denselben hineingeladen.

Kapitel 5

Realisierung der Funktionalität

Nachdem in den vorangegangenen Kapiteln ein Katalog für die angestrebte Funktionalität des Editors aufgestellt wurde und auf die Programmiersprache und die zur Verfügung stehenden grafischen Benutzerwerkzeuge eingegangen wurde, soll nun erläutert werden, wie die gewünschten Funktionen durch das Zusammenspiel sämtlicher Instrumente realisiert wurden.

5.1 Basisfunktionen

Hier sei zunächst auf die grundlegenden Funktionen wie Laden, Speichern, Übersetzen und Datenverwaltung eingegangen, die der Benutzerkreis als Grundfunktionalität erwartet.

5.1.1 Dateioperationen

Dateioperationen sind in zwei Granularitäten definiert: Zum einen kann eine ganze Klasse geladen oder gespeichert werden. In diesem Fall wird für jede der sechs Komponenten einer Klasse, sowie für die textuelle Erklärung zu den Integritätsbedingungen eine eigenen Datei angelegt. Basis für die Wahl des Dateinamens ist der Name der Klasse, dem dann verschiedene Extensionen hinzugefügt werden. In welches Verzeichnis diese Dateien geschrieben werden, liegt in der Hand des STYLE-Topeditors. Er übergibt dem Klasseneditor den gewünschten Verzeichnispfad. Zum anderen ist es möglich, nur jeweils eine Komponente der Klasse zu sichern oder zu laden. Möchte der Benutzer eine Klasse (oder Teile derselben) laden, so öffnet sich das oben beschriebene Datei-Auswahl-Fenster und fordert zum Selektieren einer Klasse auf. Die initiierende Applikation wird so lange "eingefroren", bis der Benutzer eine Auswahl getroffen hat, oder das Fenster durch Betätigen des "Cancel"-Buttons verläßt.
Die Abbildung 15 zeigt die Realisierung des Klassen-Auswahl-Fensters. Nach erfolgter Selektion einer Klasse und Aktivierung des "Open"-Knopfes erscheint ein neuer Klasseneditor auf dem Bildschirm, der die ausgewählte Klasse anzeigt.

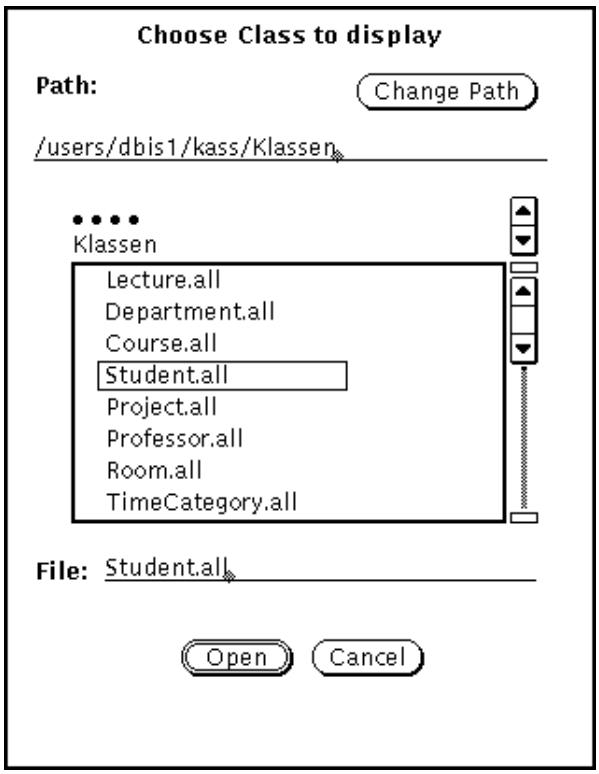

Abbildung 5.1: Eine File-Selection-Box zum Laden einer Klasse

5.1.2 Parsen, Unparsen

Um sinnvoll mit den erzeugten Klassen arbeiten zu können, ist zunächst eine grammatikalische Analyse der Klassendefinition vonnöten. Sie ist Aufgabe des **Parsers**. Er zerlegt die Klassenkomponenten in einzelne Worte, Satz- und Sonderzeichen. Hierbei wird überprüft, ob zwingend vorgeschriebene Schlüsselworte vorhanden sind, zu jeder öffnenden Klammer auch eine schließende vorhanden ist, die verwendeten Typen erlaubt und bekannt sind usw. Etwaige Fehlermeldungen mit Hinweisen auf fehlende Zeichen oder falsche Bezeichner werden dem Benutzer auf der Quest-Maschine angezeigt. Beispiel:

[endofinput:0.1: Parse error in ; expecting [Bool,Int,Nat,String,Ok,EnumOf....].,
endofinput:0.1: Parse error in ; expecting [:].]

Im Falle eines erfolglosen Kompilationsvorganges kann entschieden werden, ob der bisherige Stand der Arbeit in einer Datei gesichert werden soll, oder ob ein erneutes Editieren der Klasse mit anschließender Übersetzung gewünscht wird. Diese Möglichkeit wird über eine Notiz zur Verfügung gestellt.

Der Parser stellt seinen Dienst in zwei Granularitäten zur Verfügung: Die Analyse kann für die ganze Klasse oder für eine einzelne Komponente derselben erfolgen.

5.1.3 Schnittstelle zur Metadatenverwaltung

Konnte der Parser die syntaktische Korrektheit der Klassendefinition verifizieren, so wird der Anwender über eine aufgeklappte Notiz hierüber in Kenntnis gesetzt. Gleichzeitig wird

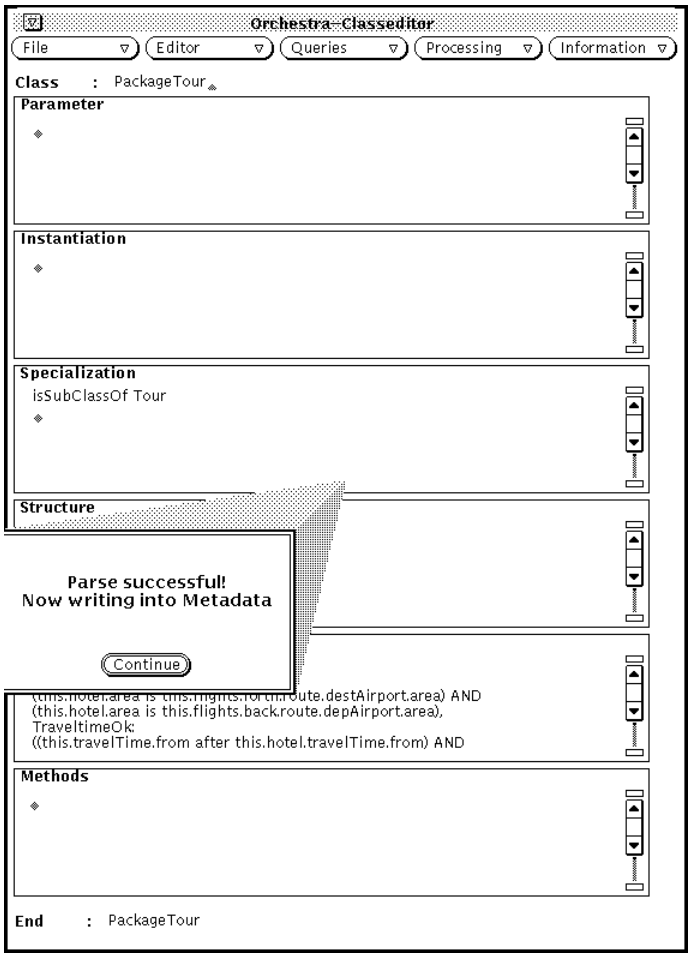

Abbildung 5.2: Meldung nach einem erfolgreichen Parse-Vorgang

ihm mitgeteilt, daß nun ein Eintrag in die Metadaten erfolgt. Hierfür stehen Wörterbücher in verschiedenen Granularitäten zur Verfügung, in die mit Hilfe des Moduls OMTransform Daten hinein- bzw. mit MOTransform Daten herausgelesen werden.

5.1.4 Öffnen eines Klasseneditors

In den seltensten Fällen wird eine einzelne Klasse für sich allein stehen. Vielmehr sind Klassen in der Regel in einen größeren übergeordneten Kontext, ein Schema oder ein System, eingebunden. Somit kann es aus Gründen der Übersichtlichkeit erforderlich werden, sich mehrere Editoren gleichzeitig auf dem Bildschirm darstellen zu lassen. Es werden folgende Möglichkeiten für das Öffnen eines weiteren Editors bereitgestellt:
Zum einen kann über das Hauptmenü ein neuer leerer Editor erzeugt werden. Dieser ermöglicht dem Anwender die Spezifikation einer neuen Klasse und trägt somit zur Vervollständigung seiner Anwendung bei. Zum anderen kann der Benutzer einen neuen Editor mit einer als Datei bereits vorhandenen Klasse füllen. Diese Methode bedient sich des bereits weiter oben beschriebenen Datei-Auswahl-Fensters.
In Hinblick auf echte Datenbankfunktionalität wurde dafür gesorgt, daß auch das Laden von Klassen aus den Metadaten direkt möglich ist. In einem Schemawörterbuch wird nach dem Namen der gewünschten Klasse gesucht, die dann - sofern vorhanden - dem Unparser übergeben wird. Dessen Ausgabe wird dann direkt in die Komponenten der Klasse umgeleitet und für den Benutzer sichbtar gemacht.
Während die bisher beschriebenen Möglichkeiten den **wahlfreien** Zugriff aus eine beliebige Klasse erlaubten, gibt es außerdem die Möglichkeit, Editoren **kontextsensitiv** zu öffnen, das heißt die Auswahl auf solche Klassen einzuschränken, die in Relation zu einer bestimmten Klasse stehen. Diese Möglichkeit wird über Anfragen (siehe weiter unten) zur Verfügung gestellt. Klassen, für die das Anfrageprädikat erfüllt ist, werden in der rollbaren Liste eines Klassenauswahlfenster, das in seinem Aufbau dem Datei-Auswahl-Fenster ähnlich ist, angezeigt. Die Auswahl einer Klasse bewirkt das Kreieren eines neuen Editors, der dann mit der gewünschten Klasse gefüllt wird. Darüber hinaus muß natürlich noch einmal zur Verdeutlichung betont werden, daß auch das Öffnen eines leeren oder mit einer bestimmten Klasse gefüllten Klasseneditors aus der nächst höheren Schicht, nämlich dem Schemaeditor, heraus möglich ist.

5.2 Funktionen zur selektiven Größenadaption

Die Veränderung der Größe einer oder mehrerer Komponenten zur Unterstützung lesender und schreibender Zugriffe auf Klassendefinitionen ist Gegenstand dieses Abschnittes.

5.2.1 Funktionen zum Entfernen und Anzeigen von Komponenten

Der zentrale Klient des Fensters, in dem die Applikation läuft ist ein Grid Panel, welches aus sechs Zeilen und einer Spalte besteht. In jeder dieser Zeilen ist eine Komponente der Klassendefinition untergebracht. Will der Benutzer eine Komponente der Klasse vom Bildschirm entfernen, so werden *alle* Klienten aus dem Panel entfernt, ehe ihm die verbleibenden Komponenten wieder hinzugefügt werden. Diese Maßnahme ist erforderlich, da

NeWS intern versucht, die durch das Entfernen einer Komponente entstehende Lücke mit dem jeweils letzten Objekt der Klientenliste aufzufüllen. Dies würde zu einer unerwünschten Veränderung der Reihenfolge der Klassenkomponenten führen.
Beim Wiederauffüllen des Panels findet gleichzeitig eine Anpassung der Größe statt. Die neue Größe wird intern kalkuliert und errechnet sich für die Höhe als Quotient aus der Höhe des Panels und der Anzahl der verbleibenden Komponenten. Eine Änderung der Breite wird nicht vorgenommen. Die Internas bleiben dem Benutzer verborgen, da nur das Modell, nicht aber der View auf die Daten verändert wird. Der Benutzer hat während der Umstrukturierungsphase die alte Sicht auf den Editor. Erst wenn das Panel wieder den von außen geforderten richtigen Inhalt hat, wird der View aktualisiert, so daß der aktuelle Inhalt des Panels jetzt wieder richtig repräsentiert wird. Die programmiertechnische Realisierung sei im folgenden auszugsweise vorgestellt:

```
01 let hide(a:Int locater:Array(Int) contents:String
02     windowClient:Panel_CalculatedPanel
03     showsub:GuiRep_Menu components:Iter_T(GuiRep_Object)):Ok =
04 begin
05         menu.appendItem(showsub tuple contents fun()
06         show(a locater windowClient showsub components) end)
07         panel.removeClient(windowClient "methods") (6x)
08         locater[a-1] := locater[6]
09         locater[6] := locater[6] + 1
10         adaptSize(locater windowClient components)
11         if locater[0] is  1
12         then panel.addGridClient(windowClient "parameter"
13         iter.nth(components 1)) end (6x)
14 end
```

Als wichtigste Parameter werden die Position der zu versteckenden Komponente a, ein Feld, dessen Inhalt Aufschluß darüber gibt, welche Komponenten aktuell sichtbar sind und die Bezeichnung der Komponente übergeben. Zeile fünf und sechs bewirken, daß die Liste, die alle ausgeblendeten Komponenten enthält, um den Namen der zu entfernenden Komponente und die Funktion zum Wiedersichtbarmachen erweitert wird. In Zeile 7 werden alle Komponenten aus dem Panel entfernt. Anschließend wird vermerkt, an welcher Position im Menü diese Komponente auftaucht. Zeile zehn nimmt die angesprochene Größenanpassung vor. Abschließend werden alle noch sichtbaren Komponenten wieder in das Panel eingefügt. Das Hinzufügen einer Komponente erfolgt analog zur oben beschriebenen Vorgehensweise.

5.2.2 Zooming

Soll eine Klassenkomponente auf maximale Größe transformiert werden, so wird ähnlich wie im vorigen Abschnitt beschrieben vorgegangen. Zunächst werden alle Klienten aus

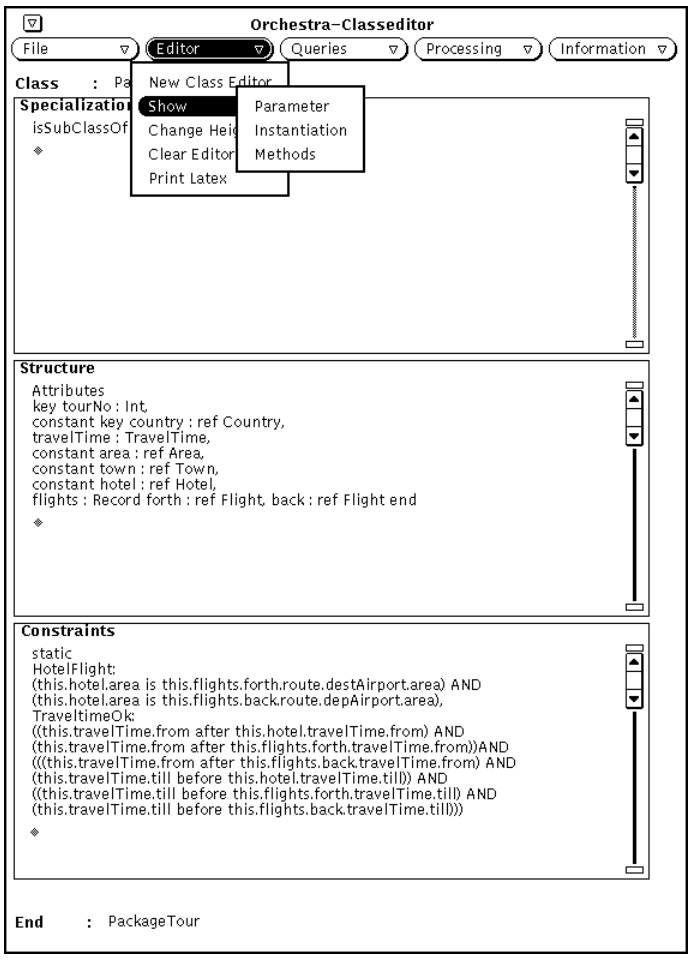

Abbildung 5.3: Editor mit drei versteckten Komponenten

dem Panel entfernt. Im nächsten Schritt erfolgt die Größenanpassung: Die Größe der zum "Zoomen" ausgewählten Komponente wird der aktuellen Größe des Panels gleichgesetzt. Anschließend wird die nun vergrößerte Komponente dem Panel wieder als Klient hinzugefügt. Im Popup-Menü der vergrößerten Komponente werden die Funktionen "Zoom" und "Hide" deaktiviert; stattdessen wird die Funktion "Unzoom" zugänglich gemacht, die die Umkehroperation zum "Zoom" darstellt. Erst nach der abschließenden Aktualisierung des Views auf den Editor wird dem Benutzer die "gezoomte" Komponente, die nun das gesamte Fenster ausfüllt, angezeigt.
Programmauszug:

```
01 let zoom(c:Int windowClient:Panel_CalculatedPanel
02       components:Iter_T(GuiRep_Object)):Ok=
03         begin
```

```
04          panel.removeClient(windowClient "methods") (6x)
05          let windowSize:Drawable_Size = drawable.size(windowClient)
06          let sizeString:String = fmt.int({windowSize.w - 5}) <> " "
07          <> fmt.int({windowSize.h - 15})
08          if c is 1 then
09          panel.addGridClient(windowClient "parameter" iter.nth(components 1)) (6x)
10          object.installMethod(iter.nth(components 1) "preferredsize" sizeString)
11          drawable.xReshape(iter.nth(components 1))
12      end
```

In Zeile vier werden alle Komponeneten aus dem Panel entfernt; die zu vergrößernde Komponente wird in den Zeilen acht und neun bestimmt und wieder eingefügt. Ihre Größe, die der Fenstergröße gleichgesetzt wird (Zeile 5), wird dem externen Server als neue Größe geschickt, der dann die entsprechende Änderung am Fenster vornimmt.

5.2.3 Fensterhöhenanpassung

Wünscht der Benutzer eine vertikale Größenadaption des Fensters, so wird ihm in einem Aufklappfenster ein Schieber zur Verfügung gestellt, mit dem er dessen Höhe im Bereich von 25% - 100% einstellen kann. Nach Aktivierung eines Ok-Knopfes wird der Aktuelle Wert des Schiebers abgelesen und die Höhe des Fenster neu berechnet. Sukzessive werden dann auch alle internen Klienten in ihrer Größe angepaßt. Ist dieser Vorgang abgeschlossen, erfolgt eine Aktualisierung der Sicht auf den Editor, wodurch dessen Änderung am Bildschirm sichbtar wird.
Eine Änderung der Fenstergröße über dessen Dimensionierungsecken ist nicht möglich. Sie wurden entfernt, da sich das interne Panel nicht automatisch der neuen Fenstergröße anpassen kann. Eine Redimensionierung des Applikationsfensters würde somit seiner Wirkung beraubt werden.

5.3 Anfragen zur Modellierungsunterstützung

Im folgenden Abschnitt werden die Anfragemöglichkeiten vorgestellt, die dem Modellierer einen Überblick über das Schema und in Beziehung stehende Klassen ermöglichen und so seine Tätigkeit unterstützen.
Der prinzipielle Weg, eine Anfrage in den bestehenden Editor zu integrieren, ist der folgende:

1. Bereitstellung der benötigten Anfragefunktion. Diese stellt das Modul EditSupport zur Verfügung. Die Funktion bekommt die benötigten Parameter übergeben und greift auf die Metadaten zu, um die Anfrage zu lösen.

2. Einbindung der ausprogrammierten Anfrage in ein Menü.

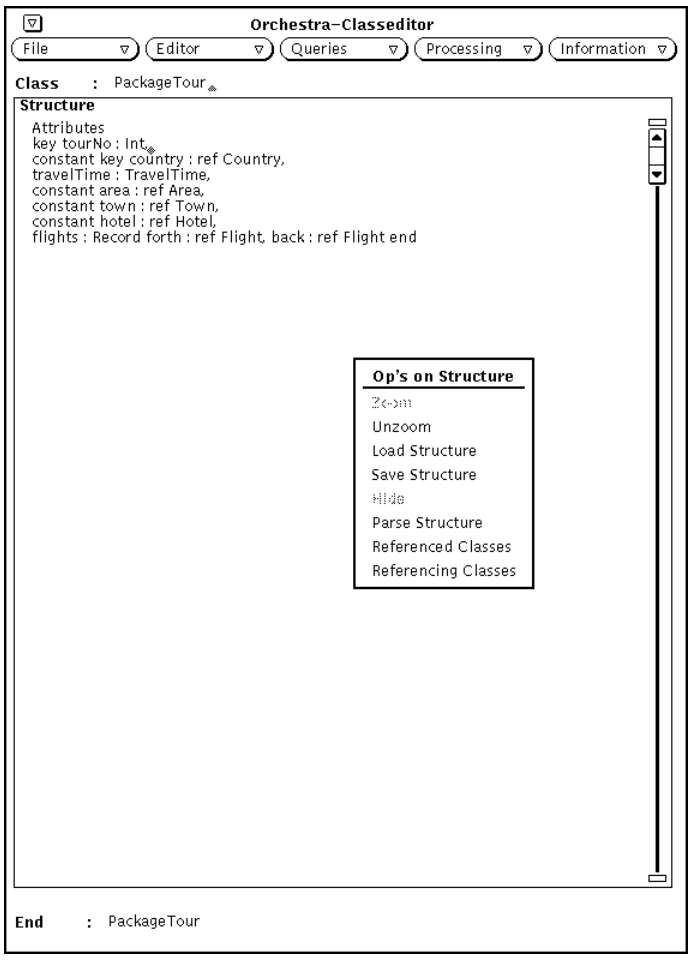

Abbildung 5.4: Vergrößerte Strukturkomponente mit lokalem Menü

3. Aufbereitung des Anfrageergebnisses in einer Notiz oder einem eigenen Fenster.

5.3.1 Anfragen nach referenzierter Klassen

Die Strukturkomponente einer Klasse kann Beziehungen zu anderen Klassen aufweisen. Der Benutzer kann daher insbesondere daran interessiert sein, sich sämtliche dieser Referenzen anzeigen zu lassen. Bei der Verwirklichung dieser Anfragen wurde nicht nur die Möglichkeit geschaffen *direkt* referenzierte Klassen anzuzeigen, sondern es können auch die *transitiv* referenzierten Klassen ermittelt werden.
Es können jedoch auch die referenzierenden Klassen von Interesse sein. Soll beispielsweise der Eintrag einer Klasse aus den Metadaten gelöscht werden, so muß auf jene Klassen geachtet werden, bei denen dieser Vorgang zu "hängenden" Referenzen führen kann. Aus

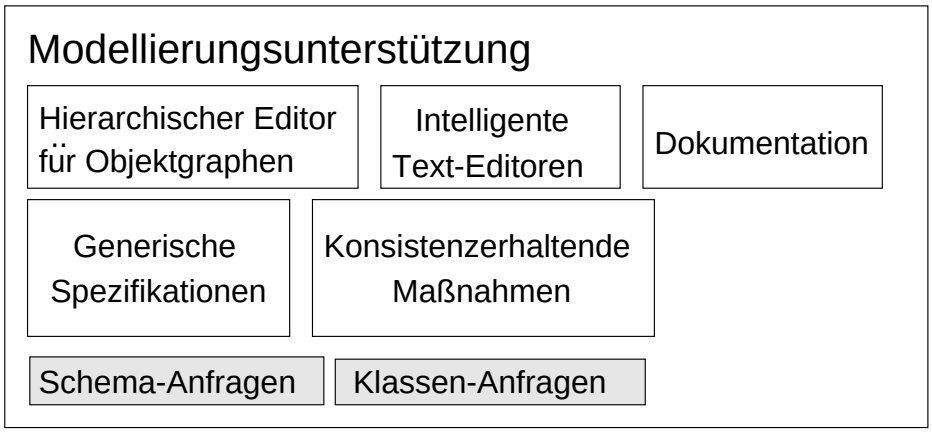

Abbildung 5.5: modellierungsunterstützende Anfragen

diesem Grund wurde auch eine solche Anfragemöglichkeit in den Hauptmenüs des Editors untergebracht.

5.3.2 Anfragen an Subklassen

In der realen Welt lassen sich Objektklassen häufig in eine Taxonomie einordnen, die den Zusammenhang zwischen allgemeinen Klassen und deren spezielleren Ausprägungen offenbahrt. Ist eine Klasse die Generalisierung beziehungsweise Spezialisierung einer anderen Klasse, so findet bei der Datenmodellierung das Konzept der Subklassen Anwendung. In einer solchen Beziehung stehen beispielsweise Boot und Kanu, sowie Haus und Bungalow. Diesem Konzept trägt die Spezialisierungskomponente der Klasse Rechnung: Sie gibt an welche Superklasse eine Klasse hat und damit welche Klasse ihre gesamten Attribute an sie weitervererbt. Um nun aber auch in Erfahrung zu bringen, welche Subklassen die gerade angezeigte Klasse hat, wurde im Anfragen-Menü eine entsprechende Funktion untergebracht. Sie bezieht ihre notwendigen Information aus dem Schemawörterbuch für die aktuelle Anwendung.

5.3.3 Anfragen nach der Klassenvollständigkeit

Eine Klasse darf keine "hängenden" Referenzen, das heißt Referenzen auf andere Klassen oder Typen, die nicht existent sind, beinhalten. Sind alle referenzierten Objekte auch tatsächlich erreichbar (bezüglich der aktuellen Schemadefinition), so bezeichnet man die Klasse als **vollständig**, anderenfalls als unvollständig. Bei dieser Anfrage wird in einem Schema-Wörterbuch geprüft, ob für alle Objekte, auf die Bezug genommen wird, ein Eintrag in den Metadaten existiert. Ist dies nicht der Fall, so wird der Benutzer mit Hilfe einer Notiz, die die fehlenden Typen und Klassen anzeigt, hierüber informiert. Die Realisierung

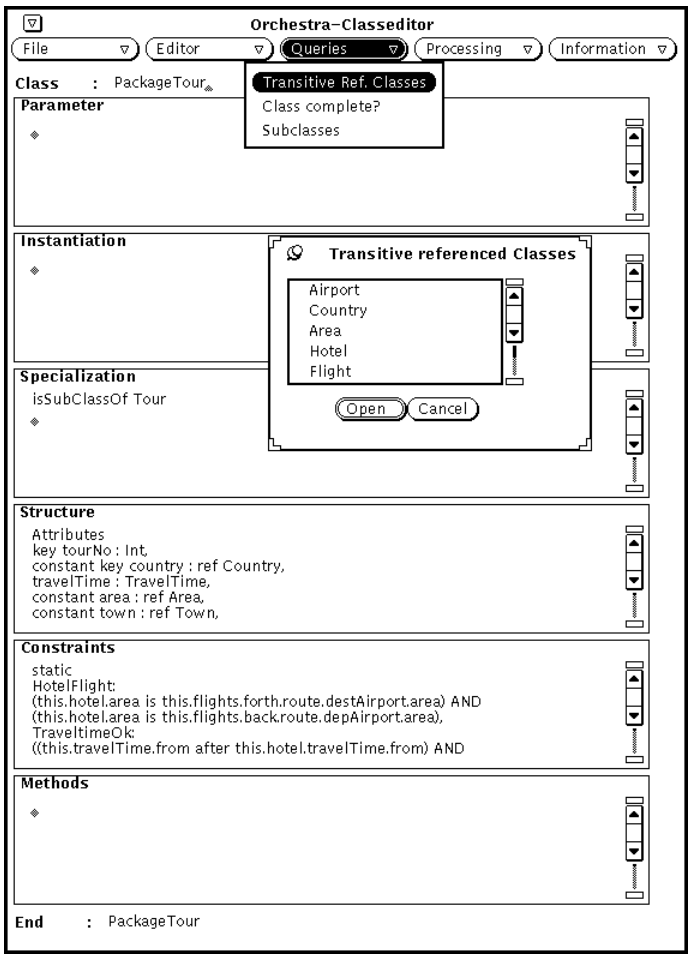

Abbildung 5.6: Rollbare Liste, die alle transitiv referenzierten Klassen anzeigt

wurde folgendermaßen vorgenommen:

```
01 option notify of Menu_Item with "Class complete?" fun() begin
02   let r = systemStart.getRoot()
03   let s = r.stdsystem
04   let title = getContents(classtext)
05   let mySchema = schemaDictionary.lookup(
06                  mSystem.getSchemaDictionary(s) schemaName)
07   if editSupport.isCompleteClass( mySchema title)
08   then gui.displayNotice(win1
09              iter.enum of "Class complete!" end
10              iter.enum of "Ok" end 1)
```

```
11      ok
12  else
13    let classes = editSupport.notDefinedClassesOfClass(mySchema title)
14    let types   = editSupport.notDefinedTypesOfClass(mySchema title)
15    let mssg1   = iter.enum of "Class not complete!" "Classes missing:"
16                "==============" end
17    let mssg2   = iter.enum of "-------------" "Types missing:"
18                "==============" end
19    let mssg3   = iter.append(mssg1 classes)
20    let mssg4   = iter.append(mssg2 types)
21    gui.displayNotice(win1 iter.append(mssg3 mssg4) iter.enum of "Ok" end 1)
22    ok
23 end
24 end end
```

Zeile eins ergänzt ein bestehendes Menü um die Abfrage nach der Klassenvollständigkeit. Nachdem der Funktion der aktuelle Stand der Metadaten bekannt gemacht worden ist (Zeilen 2 und 3), wird auf der aktuellen Klasse (Zeile 4) basierend geprüft, ob alle referenzierten Klassen und Typen in der Schemadefinition bereits existieren (Anfrage in Zeile 7). Ist dies der Fall, so wird für den Datenbankbenutzer eine Notiz eingeblendet,die ihm dies bestätigt (Zeilen 8-11). Anderenfalls wird genauer nachgeforscht, welche Referenzen ungebunden sind (Zeilen 13 und 14). Anschließend wird eine Notiz kreiert, in der nach Klassen und Typen sortiert angezeigt wird, was es noch zu spezifizieren gilt, um eine konsistente Klassendefinition zu erhalten.

Kapitel 6

Ausblick

Im diesem abschließenden Kapitel soll auf meine Erfahrungen mit den benutzten Entwurfswerkzeugen näher eingegangen werden. Hierbei werden insbesondere die Programmiersprache Quest, sowie das NeWS-Toolkit näher analysiert. Das Ende dieses Kapitels bildet dann die kritische Beurteilung meiner Arbeit, sowie eine Aufstellung möglicher und sinnvoller Erweiterungen, mit denen der Editor ergänzt und weiterentwickelt werden könnte.

6.1 Erfahrungen mit dem NeWS-Toolkit und seiner Schnittstelle zu Quest

Die Benutzung der TNT-Quest-Schnittstelle hat das Konstruieren des Editors wesentlich vereinfacht, da sich die gewünschte grafische Benutzeroberfläche mit relativ geringem Aufwand nach dem "Baukastenprinzip" erstellen ließ. Dem gegenüber stand die mühsame und fehlerträchtige Programmierung von graphischen Funktionseinheiten in *PostScript*, die immer dann erforderlich wurde, wenn eine benötigte Funktion nicht im Umfang der Schnittstelle enthalten war. Trotz dieser Vorteile muß hier auch auf einige Probleme hingewiesen werden:
Zum einen hat sich herausgestellt, daß gewisse Objektklassen nicht ohne weiteres miteinander kombinierbar sind. So kann beispielsweise folgende Situation auftreten: Bei der Größenänderung eines Fensters wird dessen Inhalt nicht größenadaptiert, das heißt er bleibt in der ursprünglichen Größe bestehen. Ein weiteres Problem tritt in Zusammenhang mit der Pufferung von Befehlen bei Überlastung des Servers auf: Es kommt vor, daß ein Befehl nur zur Hälfte im Puffer steht, der andere Teil aber schon abgearbeitet wird. Bei nicht rechtzeitiger Freigabe des Puffers wird daher vergeblich auf die fehlende Information gewartet. Allerdings hat der Programmierer Möglichkeiten, dieses Problem programmiertechnisch weitestgehend in den Griff zu bekommen, indem er regelmäßig eine Freigabe des Puffers erzwingt. Eine gelegentliche Restauration des Bildschirms (mit der "refresh"-Funktion) kann jedoch nicht gänzlich vermieden werden.

6.2 Erfahrungen mit der Programmiersprache Quest

Viele Konzepte von Quest konnten im Kontext dieser Studienarbeit sinnvoll verwandt werden: Funktionen höherer Ordnung beispielsweise kamen beim Füllen von Menüs mit Funktionalität zum Tragen, während Persistenz beim Ablegen komplexer Struktureinheiten in den Metadaten ausgenutzt wurde. Bedienungsfehler führten nicht notwendigerweise zum sofortigen Abbruch des Programms, sondern konnten komfortabel in Ausnahmebehandlungen abgefangen werden. Als unentbehrlich erwies sich darüberhinaus die Vielzahl bereits bestehender Funktionen, die ein solides Fundament für die Entwicklung eines komplexeren Softwareprodukts bilden. Durch die Kombination aus Bibliotheksfunktionen zur Massendatentyp- und Graphikunterstützung, sowie Moduln, die auf das spezielle Datenbankmodell zugeschnitten worden sind, wurde die Entwicklung dieses Programms überhaupt erst möglich. Die bestehenden Funktionen konnten für die editorspezifischen Erfordernisse angepaßt werden.

6.3 Der Editor - eine Abschlußbetrachtung

Zusammenfassend kann gesagt werden, daß die bestehende Form des Editors der Lösung der gestellten Aufgabe in großem Umfang gerecht wird. Seine integrative Stellung im Datenbanksystem macht ihn jedoch sehr anfällig gegenüber Erweiterungen durch zusätzliche Funktionen. Die Menüs des Editors können jederzeit mit weiteren neuen Funktionen gefüllt werden, doch ist zu deren Einbindung eine Modifikation des Programmtextes erforderlich, da diese nicht über die Schnittstelle mitgegeben werden können. Ebenso führt eine Änderung der Schnittstellen importierter Funktionen stets zu einer Veränderung des Programmtextes.
Die Erstellung von Software ist stets einem dynamischen Prozeß unterworfen. Auch dieser Editor kann daher nur vorläufig als fertiges Produkt angesehen werden. Nachfolgend sind eine Reihe von möglichen Erweiterungen aufgeführt, die den Umgang mit dem Datenbanksystem noch effizienter gestalten könnten und zugleich den Bedienungskomfort erhöhen:

- Es liegt in der Natur menschlicher Wahrnehmung eher in Bildern als in geschriebenen Texten zu denken, daher könnte die Darstellung einer Klassendefinition auch durch die Integration einer graphischen Repräsentation unterstützt werden, die dann zur Visualisierung der textuellen Spezifikation dient. Ebenso sollte die Möglichkeit bestehen, aus einer graphischen Repräsentation Text zu erzeugen, der dann weiter editiert werden könnte. Ein solcher Graphikeditor ist Gegenstand einer laufenden Diplomarbeit. Die grafische Darstellung eines Schemas ist in Anhang B abgebildet.

- Des weiteren könnte der Editor derart erweitert werden, daß er syntaxgesteuert ist. Syntaxfehler bei der Eingabe von OM1-Spezifikationen würden dann frühzeitig, das heißt im Editor selbst und nicht erst beim Parsen, erkannt werden.

- Die Modellierung kann durch Integration eines Editors für Datenbankoperationen sinnvoll ergänzt werden. Der Datenbankeditor ermöglicht dem Benutzer die Ausführung von Standardmethoden, wie das Erzeugen, Löschen und Ändern von Objekten, sowie das Anzeigen der Extension einer Klasse. Die Auswahl einer dieser Operationen für eine bestimmte Klasse führt zum Öffnen eines generierten Editors, der die Eingabe der Parameterwerte für die Operation erwartet und sie danach ausführt. Derzeit laufen zwei Studienarbeiten, die die Generierung dieser Datenbankeditoren und ihre Anbindung an den Schemaeditor zum Ziel haben.

- Auch die Einbindung einer eigenen Anfragesprache bezüglich der Designobjekte der Metadatenbank, mit der der Benutzer Adhoc-Anfragen an das Schema stellen kann, trägt zu einer nutzbringenden Erweiterung der Funktionalität bei. Ihre Bereitstellung setzt zum einen die programminterne Abbildung dieser Anfragen auf die Metadaten voraus; zum anderen aber auch die geeignete Aufbereitung des Anfrageergebnisses. Dieses könnte beispielsweise wiederum in Editoren dargestellt werden.

- Der Klasseneditor bietet die Möglichkeit, für OM1-Spezifikationen entsprechende Klassenschnittstellen und -module in der Programmiersprache TYCOON zu generieren. Daher wird es für den Benutzer von Interesse sein festzustellen, welche Module bereits generiert und in welchen Verzeichnissen sie unter welchem Namen abgespeichert wurden. Diesen Dienst könnte eine integrative Komponente leisten, die vom Editor aus aufgerufen wird.

- Integritätsbedingungen sind häufig klassenübergreifend, da sie mehrere Klassen tangieren können. Bei der Definition einer Klasse in OM1 muß jedoch die Spezifikation von Integritätsbedingungen innerhalb einer bestimmten Klassendefinition erfolgen. Es kann aber sinnvoll sein, nach allen Integritätsbedingungen zu fragen, die zu einer Klassendefinition gehören, aber nicht direkt in ihr spezifiziert sind. Eine solche Anfragemöglichkeit existiert bereits, wurde aber bisher noch nicht in den Klasseneditor integriert.

- Die Definition eines Schemas besteht, wie bereits gesehen, nicht nur aus einer Menge von Klassen, sondern auch aus benutzerdefinierten Typen, die innerhalb einer Klassendefinition benutzt werden. Den Benutzer wird es daher interessieren, ob alle benutzten Typen existieren und wie ihre Definition gegebenenfalls aussieht. Dies setzt die Existenz eines Typeditors voraus, der den gewünschten Typ am Bildschirm anzeigt und eine dem Klasseneditor vergleichbare Funktionalität anbietet. Seine Integration muß zum einen direkt auf der Schemaebene erfolgen, was beispielsweise bei der Spezifikation neuer Typen von Interesse ist, zum anderen muß er aber auch aus dem Klasseneditor heraus aufrufbar sein, um die Struktur eines benutzten Typs anzuzeigen. Dieser Typeditor ist bereits in die Datenentwurfsumgebung integriert worden (siehe Abbildung in Anhang C).

Anhang A

Benötigte Module

Aufgrund seiner großen Komplexität macht sich der Editor die Dienste diverser Bibliotheksmodule zunutze. Diese seien dem Leser im folgenden kurz vorgestellt:

Module zur Graphikunterstützung:

- Gui
- GuiRep
- EventMgr
- Panel
- Action
- ItemGroup
- Wire
- Window
- Control
- Drawable
- Layout
- FileSelect
- Object
- Scrollbar
- Menu
- Gnt
- Popup
- MenuButtons
- Label
- Base
- ScrollList
- BorderBag
- Buttons
- Canvas
- Slider

Module zur Bereitstellung von Editiermöglichkeiten:

- Jot
- JotText
- JotView

Standardmodule:

- StringOp
- Iter
- Print
- IntOp
- Fmt
- Source
- Writer
- Process

Module zum Parsen, Unparsen und zur Dokumentenerzeugung:

- OTexUnparse
- TexUnparser
- Unparser
- OUnparser
- OParse
- OClass
- OStructure
- OConstraints

Modul zur Unterstützung von Anfragen:

- EditSupport

Module zum Laden und Speichern:

- UxFile
- Directory

Module zur Metadatenorganisation:

- MClass
- MSchema
- MSystem
- ClassDictionary
- SchemaDictionary
- SystemStart
- MOTransform
- OMTransform

das Generatormodul:

- GenTLCode

Modul zur Unterstützung des Klasseneditors:

- ClassEditorhelp

Anhang B

Graphische Visualisierung von Schemata

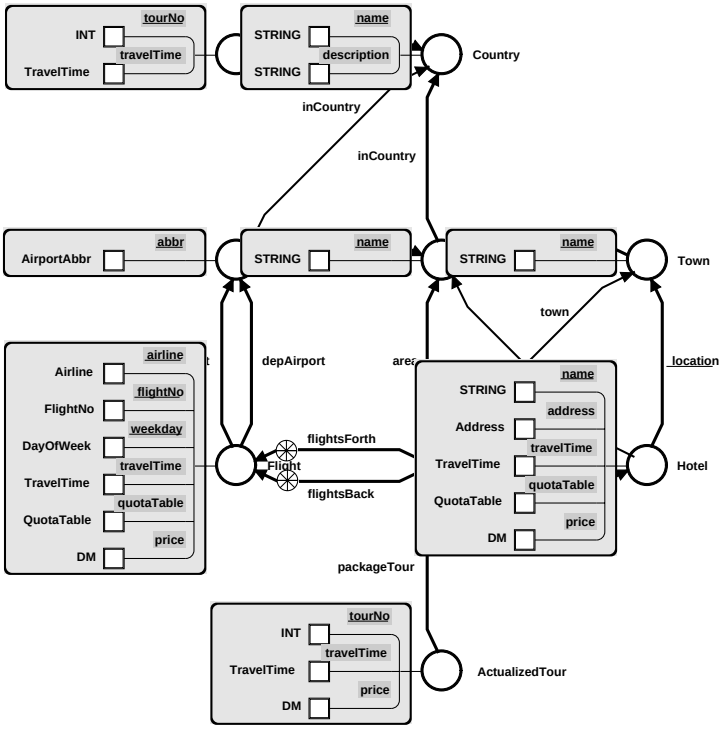

Abbildung B.1: graphische Repräsentation einer Anwendung

Anhang C

Der Typeditor

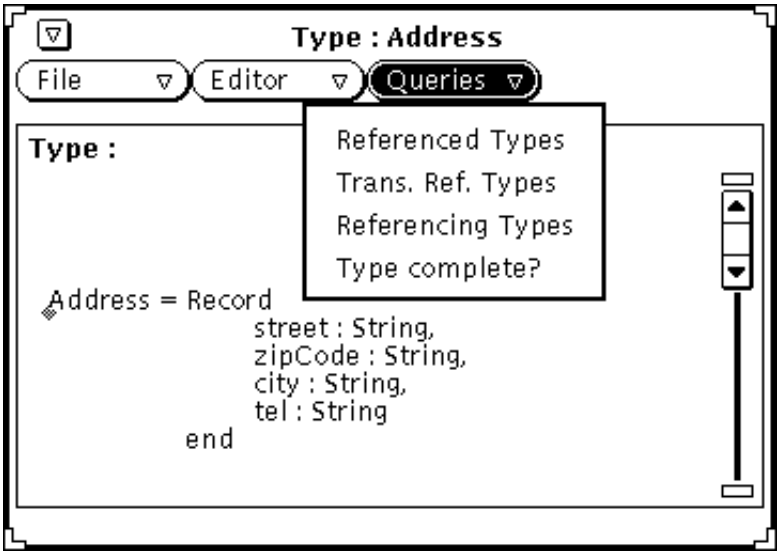

Abbildung C.1: Ein Editor für die Modellierung von Typen

Literaturverzeichnis

[AbKh90] R.Abnous, S.Khoshafian, *Object Orientation, Concepts, Laguages, Databases, User Interfaces*, Wiley Professional Computing, 1990

[AbSu85] H.Abelson, G.J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1985

[Car89] L.Cardelli, *Typeful Programming*, Report 45, Digital Systems Research Center, DEC SRC Palo Alto, Mai 1989

[CaWe85] L.Cardelli, P.Wegner, *On Understanding Types, Data Abstraction, and Polymorphism*, ACM Computing Surveys, Vol.17, 1985

[GRA89] J.Gosling, D.S.H.Rosenthal, M.J.Arden, *The NeWS Book, An Introduction To The Network/Extensible Window System* Springer-Verlag, 1989

[Heu92] A.Heuer, *Objektorientierte Datenbanken*, Addison-Wesley, 1992

[Hoa69] Hoare, C.A.R., *An Axiomatic Approach to Computer Programming*, Communications of the ACM, 1969

[Mar92] A.Marcus, *Graphic Design for Electronic Documents and User Interfaces*, ACM Press, 1992

[Mat92] F.Matthes, *Generische Datenbankprogrammierung: Sprachliche und architektonische Grundlagen*, Dissertation am Fachbereich Informatik der Universität Hamburg, Dezember 1992

[Mic90a] Sun Microsystems, Inc., *OPEN LOOK Graphical User Interface Functional Specification*, Addison-Wesley, 1990

[Mic90b] Sun Microsystems, Inc., *OPEN LOOK Graphical User Interface Application Style Guidelines*, Addison-Wesley, 1990

[MMN92] F.Matthes, S.Müßig, C.Niederée, *P-Quest Benutzerhandbuch*, DBIS Tycoon Report 102-92

[ScMa93] J.W.Schmidt, F.Matthes, *Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems*, FIDE Technical Report Series, April 1993

[Sun92] SunSoft, A Sun Microsoftsystems Company, *The NeWs Toolkit 3.1 Reference Manual*, 1992

[Sun92] SunSoft, A Sun Microsoftsystems Company, *NeWs 3.1 Programmers Guide*, 1992

[WSSB91] M.Jarke (Herausgeber), *Database Application Engineering with DAIDA*, Research Reports ESPRIT, Vol.1, Kapitel 9, I.Wetzel,K.-D.Schewe,J.W.Schmidt, A.Borgida, *Specification and Refinement of Databases and Transactions*, Springer-Verlag, 1991