Diplomarbeit

Ein Graphikeditor für objektorientiertes Modellieren

August 1994

vorgelegt von:
Ralf Löst
Tangstedter Landstr. 455
22415 Hamburg

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

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Entwicklung datenintensiver Anwendungen mit Hilfe rechnergestützter Entwicklungsumgebungen stellt unterschiedliche Anforderungen an die zur Verfügung zu stellenden Werkzeuge und Dienste. Neben den aus traditionellen Datenbankanwendungen bekannten Aufgaben, wie z.B. Verwaltung und Auswertung von strukturierten Massendaten, sollten auch der Entwurfs- und Entwicklungsprozeß von Anwendungen unterstützt sowie verschiedene Schritte dieser Prozesse integriert werden [BEM92]. Ferner erfordern neue und komplexere Anwendungen – beispielsweise aus dem *Multimedia*-Bereich – speziell für sie geeignete Modellierungsmittel [Hug91].

In diesem Kontext ergeben sich folgende Anforderungen und Ziele. Für die Spezifikation von Anwendungen sollten Datenmodelle benutzt werden, deren Konzepte für die Erfassung anwendungsspezifischer Semantik geeignet sind. Außerdem sollten die Konzepte formal definiert sein. Es sollten verschiedene Datenmodelle flexibel unterstützt werden, um verschiedenen Anwendungen angepaßte Modellierungsarten zu ermöglichen. Ferner sind Softwarewerkzeuge bereitzustellen, die im Rahmen eines gegebenen Datenmodells eine benutzungs- und anwendungsgerechte Funktionalität zur Spezifikation und Präsentation von Anwendungen anbieten.

Neben der textuellen spielt besonders die graphische Spezifikation von Anwendungen eine wichtige Rolle. Sie verfolgt das Ziel, einen visuell schnell erfaßbaren Überblick über die wesentlichen semantischen Zusammenhänge und Beziehungen der modellierten Anwendungswelt zu verschaffen [HK87]. Die in dem Graphikmodell verwendeten Diagrammarten sollten in ihrer visuellen Gestaltung übersichtlich sein und semantische Ausdruckskraft besitzen.

Beim Entwurf und der Implementierung eines Editors zur graphischen Modellierung sind deshalb die folgenden Anforderungen besonders zu berücksichtigen.

- Benutzbarkeit:
  Verwendung einer standardisierten graphischen *Benutzeroberfläche* und Einhaltung ihrer Design- und Interaktionsrichtlinien (*style guidelines*) sowie Bereitstellung von Bearbeitungsmodi, die eine übersichtliche Gestaltung der Graphik unterstützen [Shn92] [Obe92].

- Integration der graphischen Modellierung in die Entwicklungsumgebung:
  Unterstützung des Werkzeugswechsels und der Transformation zwischen verschiedenen werkzeugspezifischen Repräsentationen einer Spezifikation [Wet94].

- Flexible Modellierung in *verschiedenen* Datenmodellen:
  Wiederverwendung generischer Bausteine, die die Realisierung von graphischen Repräsentationen anderer Datenmodelle unterstützen [Wet94].

Die genannten Anforderungen erfordern als *Realisierungsplattform* eine Programmierumgebung, die es durch ihre Benennungs-, Bindungs- und Typisierungskonzepte erlaubt, externe Dienste, wie Bildschirmserver und Werkzeuge zur Gestaltung von Benutzeroberflächen, typsicher in einen uniformen Sprachraum einzubinden [Mat93]. Ferner sollte die Programmierumgebung die typsichere Transformation zwischen verschiedenen Repräsentationen von *Entwurfsobjekten*[1] unterstützen.

Eine weitere Anforderung an Entwicklungsumgebungen für datenintensive Anwendungen ist die Unterstützung der *Implementation* der Anwendungen mit Hilfe von Datenbanken [Heu92] [Bee92a]. Dies leisten *Generatoren*, die aus formalen Spezifikationen von Anwendungen *Module* erzeugen, die Standarddatenbankoperationen, wie Erzeugen, Löschen und Ändern von Objekten, bereitstellen [BDRZ83]. Die generierten Module erlauben dem Benutzer die Erzeugung eines Prototypen zum frühzeitigen Testen der modellierten Anwendung.

Zur Realisierung dieser Anforderungen ist eine Programmierumgebung zu wählen, die eine geeignete datenbankspezifische Funktionalität anbietet. Das am Arbeitsbereich DBIS[2] entwickelte *Tycoon*[3] System [Mat93] ist eine Programmierumgebung, die die genannten Anforderungen für die Realisierung von Entwicklungsumgebungen weitgehend erfüllt.

Dieses einleitende Kapitel ist in drei Abschnitte unterteilt. Im ersten Abschnitt werden die Konzepte von Tycoon erläutert und ihre Eignung zur Implementation von Entwicklungsumgebungen für datenintensive Anwendungen begründet. Im zweiten Abschnitt wird das *STYLE*[4] Projekt [Wet94] vorgestellt, das Beiträge zum systematischen Aufbau von Entwicklungsumgebungen für verschiedene Anwendungskategorien leistet und für ein exemplarisch gewähltes objektorientiertes Datenmodell die Umgebungsunterstützung realisiert. Es bildet den Rahmen, in den sich die vorliegende Arbeit einfügt. Der letzte Abschnitt beinhaltet die Ziele der eigenen Arbeit und deren inhaltliche Gliederung.

1.1 Tycoon als Realisierungsplattform für Graphikeditoren

In diesem Abschnitt werden zunächst die Zielsetzung und die Konzepte des Tycoon Sprachkerns TL[5] [MM93] [MS92] vorgestellt. Dann folgt die Zusammenfassung der als *add-on* Ansatz [MS91] in Bibliotheken verfügbaren Funktionalität zur Datenbankanwendungsprogrammierung. Schließlich wird die Eignung der Tycoon Umgebung als Realisierungsplattform für datenintensive Entwicklungsumgebungen erörtert.

Das Tycoon System stellt einen hochabstrakten, strikt typisierten *Sprachkern TL* mit orthogonalem Persistenzkonzept für die Unterstützung datenintensiver Anwendungsrealisierung zur Verfügung. Die Zielsetzung der Tycoon Systemrealisierung ist [Mat93]:

- Bereitstellung eines einheitlichen Sprach- und Systemkerns zur Benennung, Bindung und Typisierung der für datenintensive Anwendungen relevanten Objekte und Dienste;

[1] Entwurfsobjekte sind Objekte, die während des Entwurfsprozesses erzeugt werden, z.B. Repräsentationen von graphischen und textuellen Spezifikationen sowie von generierten Datenbankschnittstellen.

[2] *Datenbanken und Informationssysteme*. Dieser Arbeitsbereich gehört zum Fachbereich Informatik der Universität Hamburg.

[3] *Typed communicating objects in open environments*

[4] Systematics of TYped Language Environments

[5] *Tycoon Language*

- Erweiterung des Sprachkerns zu einer algorithmisch vollständigen Programmiersprache, die den Prozeß der spezifischen Verarbeitung von Objekten unterstützt;
- Bereitstellung einer geeigneten Systemarchitektur mit Programmrepräsentationen, die Optimierung, Portierung und Interoperabilität von Datenbankanwendungen ermöglichen.

TL beinhaltet ein mächtiges Typsystem mit Konzepten, wie Subtypisierung, Polymorphismus und Typoperatoren. Es werden sowohl funktionale als auch imperative (veränderliche Bindungen, Zuweisungen), modulare (Modularisierung, Bibliotheken) und objektorientierte (abstrakte Datentypen, dynamische Bindung, Subtyppolymorphismus) Programmierstile unterstützt. Orthogonal dazu besteht ein Persistenzkonzept, welches die Speicherung beliebig strukturierter Objekte erlaubt.

Über den Sprachkern hinausgehende Funktionalität wird im Rahmen eines *add-on* Ansatzes in generischen *Bibliotheken* zur Verfügung gestellt. Dazu zählt u.a. *Datenbankfunktionalität*, wie Massendatentypen, Iterationsabstraktion und Transaktionskonzept. In der Bibliothek *newsenv* [Mü94b] werden externe Dienste zur Erzeugung und Manipulation interaktiver Benutzerschnittstellen als Teil von im OPEN LOOK Standard [SM90b] definierten graphischen Benutzeroberflächen angeboten und typsicher in den uniformen Sprachraum der Tycoon Umgebung eingebunden.

Ein Vorteil des *add-on* Ansatzes liegt in der flexiblen Erweiterbarkeit von Funktionalität durch Bibliotheken [MS91]. Dies erlaubt, verschiedene Datenmodelle durch modellspezifische Bibliotheken zu unterstützen sowie eine Offenheit bezüglich Anbindung und Erweiterung der Funktionalität verschiedener externer Dienste zu verwirklichen.

Ein weiterer Vorteil ist die einheitliche Benennung, Bindung und Typisierung in einem uniformen Sprachrahmen. Dadurch wird die Integration von internen und externen Diensten sowie ein effizienter und typsicherer Datenaustausch zwischen ihnen unterstützt (siehe Abb. 1.1 aus [Mat93]).

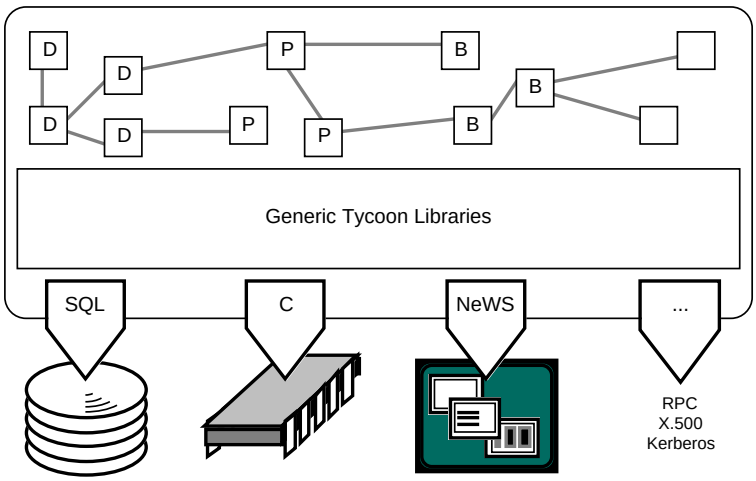


Abbildung 1.1: Integration, Erweiterung und Nutzung generischer Dienste in Tycoon

Dies erleichtert die Realisierung von komplexen und *heterogenen* Systemen, wie Entwicklungs-

systemen für datenintensive Anwendungen, in denen unterschiedliche Werkzeuge und Dienste integriert werden müssen.

1.2 Umgebungsentwicklung mit STYLE

Unterschiedliche Anwendungsbereiche erfordern eine Flexibilität der Entwicklungsumgebung hinsichtlich der Unterstützung verschiedener Datenmodelle [Bee92b]. Der STYLE Ansatz hat eine flexible Datenmodellunterstützung zum Ziel. STYLE ist als Metaprogrammierumgebung zu verstehen, die die Realisierung einzelner datenmodellspezifischer Umgebungen durch Anpassung generischer Werkzeuge, einer generischen Metadatenverwaltung und von Generatorgeneratoren ermöglicht [Wet94]. Anhand des objektorientierten Datenmodells OM1 wird in [Wet94] exemplarisch gezeigt, wie ein semantisch anspruchsvolles Datenmodell mit Hilfe von STYLE effektiv realisiert werden kann.

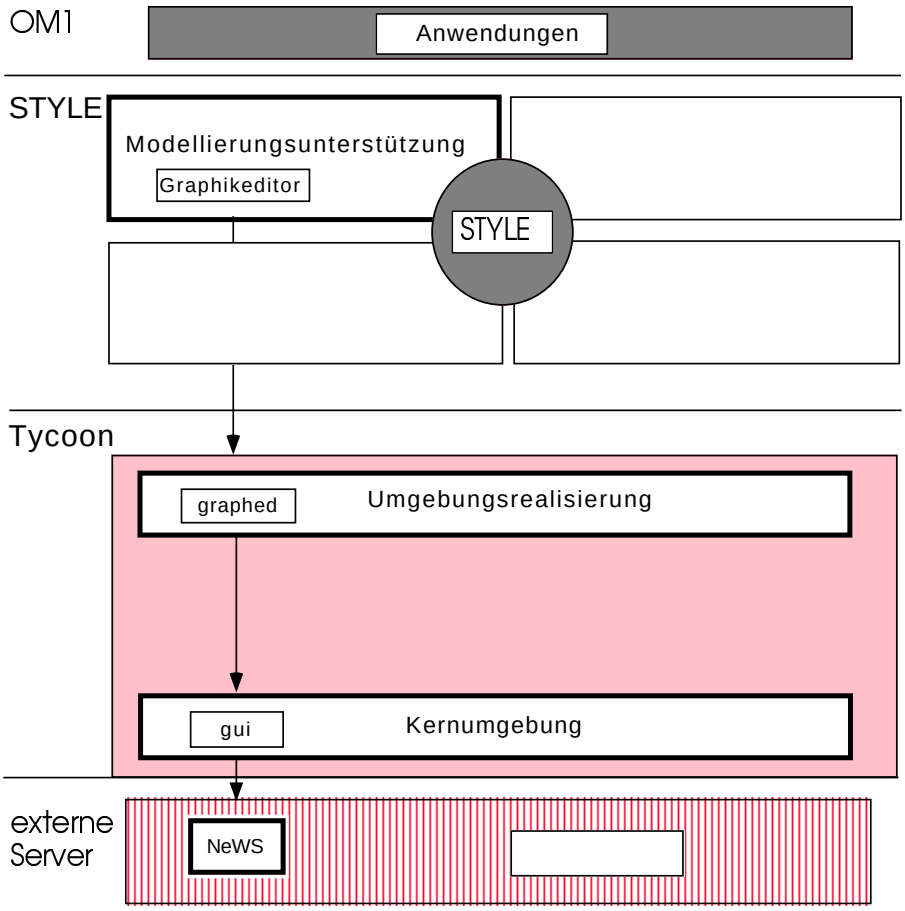


Abbildung 1.2: Architektur der Datenbankentwicklungsumgebung

Ausgehend von OM1 als konkret zu realisierendem Datenmodell, werden in STYLE Dienste und Werkzeuge zur Verfügung gestellt, die die Modellierung, Kodegenerierung, Integration und Programmierung datenintensiver Anwendungen unterstützen. Als Realisierungsplattform dient Tycoon, dessen Kernumgebung um neue Dienste im Rahmen eines *add-on* Ansatzes erweitert

wird. Die Dienste der Tycoon Umgebung lassen sich in die drei Kategorien Umgebungsrealisierung, Datenmodelldienste und Datenbankdienste aufteilen, die auf der Kernumgebung aufsetzen. *Externe Dienste*, wie Datenbankserver oder Bildschirmserver, sind in das Tycoon System eingebunden.

Die Architektur der STYLE Entwicklungsumgebung ist in Abb. 1.2[6] dargestellt. Die Realisierungsplattform Tycoon ermöglicht die Realisierung folgender Eigenschaften der Entwurfsumgebung [Wet94]:

- *typsichere Integration* von Werkzeugen, Metadatenverwaltung, Generatoren und Anwendungsprogrammen;
- *Wiederverwendung von Umgebungserweiterungen*, bezogen auf datenmodellspezifische Dienste und auf Dienste der Kernumgebung.

Im folgenden werden die auf die Entwicklungsumgebung OM1 bezogenen Dienste und Werkzeuge, gruppiert nach den vier Kategorien der Abbildung 1.2, vorgestellt.

1.2.1 Modellierungsunterstützung

Die in den folgenden Abschnitten vorgestellten Werkzeuge sind alle in einer graphischen Benutzerschnittstelle, die dem OPEN LOOK Standard [SM90b] folgt, integriert.

1.2.1.1 Der OM1 Graphikeditor

Dieser Editor dient der graphischen Modellierung von OM1 Schemata. Da die Spezifikation der Diagramme als Visualisierung von OM1 Konzepten und der Aufbau des Graphikeditors in den Abschnitten 2.3.2 und 3.3 ausführlich erörtert werden, wird hier nur kurz auf die Art der Modellierung eingegangen.

- Die unterschiedlichen Beziehungsarten zwischen Klassen (referentielle und Vererbungsbeziehungen) werden in *getrennten Diagrammen* dargestellt.
- Werteattribute von Klassen sind in verschiedenen *Sichten* ein- und ausgeblendet.
- Klassenrepräsentationen sind auf *gerasterten Positionen* angeordnet.
- Diagramme können geladen, modifiziert, gespeichert und gedruckt werden.

Es ist möglich, von der Graphikmodellierung in die textuelle Modellierung zu wechseln. Aus einer graphischen Repräsentation eines OM1 Schemas wird eine *textuelle Repräsentation* generiert. Diese steht zur textuellen Weiterbearbeitung in den Klasseneditoren zur Verfügung. Die Generierung ist als ein Teil der in Abschnitt 1.2.2 vorgestellten *Integrationsunterstützung* einzuordnen.

1.2.1.2 Der OM1 Klasseneditor

Für die textuelle Modellierung von OM1 Klassen und Typen existieren der OM1 Klasseneditor und der OM1 Typeditor [Kas94]. Der Klasseneditor ist in die syntaktischen Komponenten (vgl. Abschnitt 2.3.1), in die eine OM1 Klassendefinition gegliedert ist, unterteilt [Kas94]. Zur Förderung der Übersichtlichkeit können je nach Bedarf einzelne Komponenten ein- und ausgeblendet

[6] Diese Abbildung ist an Abbildungen aus [Wet94] angelehnt.

werden. Desweiteren können zu einem Klasseneditor über angezeigte Beziehungen der betreffenden Klasse zu weiteren Klassen eines Schemas *navigierend* weitere Klasseneditoren geöffnet werden.
Ferner ist im Klasseneditor Funktionalität für Schemaanfragen[7], zur Dokumentenerzeugung, zur persistenten Speicherung einer definierten OM1 Klasse, zu ihrer Syntaxüberprüfung sowie zur TL Kodegenerierung für lauffähige Datenbankprototypen integriert.

1.2.2 Integrationsunterstützung

Dieser Abschnitt beschreibt die Integration von Werkzeugen und Diensten unter einer einheitlichen Benutzerschnittstelle. Über Menüs bestimmter *Browser- und Auswahlwerkzeuge* sind aufrufbar:

- verschiedene Entwicklungsumgebungen und Schemata mit jeweiligen Klassen und Typen;
- verschiedene Modellierungswerkzeuge und Instanzeditoren zur Spezifikation und Instanziierung eines Schemas;
- Funktionen für Schemaanfragen bezüglich Vollständigkeit und inhaltlichen Teilsichten, zur Generierung von Datenbankprototypen sowie zur Dokumentenerzeugung.

Für die Aufrufe stehen zwei Fenster mit vier *Browsern* zur Verfügung, die im folgenden vorgestellt werden. Eine detaillierte Beschreibung ist in [Wet94] zu finden.

Im *StyleTop* Fenster wird der *Systembrowser* benutzt, um zwischen verschiedenen Entwicklungsumgebungen wählen zu können. Eine Entwicklungsumgebung beinhaltet ein vorgegebenes Datenmodell sowie eine Menge von Schemata. Nach Wahl der Entwicklungsumgebung[8], z.B. *OM1*, wird über den *Schemabrowser* die Menge der vorhandenen OM1 Schemata angezeigt.
Nach der Selektion eines Schemas stehen drei weitere Editorarten zur Verfügung:

- der Graphikeditor (siehe Abschnitt 1.2.1.1) zur graphischen Modellierung;
- der Schemabrowser (s.u.), über den Klassen- und Typeditoren zur textuellen Modellierung aufgerufen werden;
- Instanzeditoren (siehe Abschnitt 1.2.3), die zur Anzeige und Manipulation von Objekten der generierten Datenbank verwendet werden.

Im *Schemabrowserfenster* werden die Klassen und die Typen eines Schemas angezeigt, zu denen selektiv Klasseneditoren und Typeditoren geöffnet werden können. Weiterhin können Schemaunvollständigkeiten und zyklische Abhängigkeiten erkannt, Datenbankprotoypen generiert sowie *LaTeX Dokumente* erzeugt werden.

Die Integration von graphischer und textueller Modellierung eines Schemas wird durch die Erzeugung von textueller OM1 Spezifikation aus OM1 Graphik unterstützt. Damit ist es möglich, vom Graphikeditor aus Klasseneditoren aufzurufen, in denen dieselben Entwurfsobjekte des Schemas – aber aus textueller Sicht – editiert werden können. Die Realisierung dieses Teils der Integration wird in Abschnitt 5.1 dieser Arbeit ausführlich beschrieben.

[7] Beispielsweise können zu einer Klasse alle (transitiv) referenzierten Klassen in einem *Browser* angezeigt werden.

[8] Zur Zeit ist nur OM1 als Datenmodell mit zugehöriger Entwicklungsumgebung verfügbar. Es wird die OM1 Entwicklungsumgebung als *Standard* angezeigt.

1.2.3 Generierungsunterstützung

Durch Generatoren [Mü94a] werden für die in einer OM1 Spezifikation definierten Klassen in TL (Standard-)Klassenschnittstellen erzeugt, die Basisfunktionen zur Objektverwaltung[9] enthalten [Wet94]. Ferner werden weitere Module für die Überwachung von benutzerdefinierten Integritätsbedingungen, in Form von Vorbedingungen für Methoden, sowie für Initialisierungsvorgänge generiert [Mü94a]. Die Generierung von lauffähigen *Datenbankeditoren* [BW94] ermöglicht dem Benutzer, über den Aufruf von Funktionen dieser Editoren, das prototypische Testen der aus der Spezifikation generierten Datenbank durch Anlegen, Manipulieren und Löschen von Objekten sowie Iterieren über die Objekte.

1.2.4 Programmierungsunterstützung

Zur Programmierungsunterstützung dienen z.B. syntaxgesteuerte Editoren, Bibliotheksbrowser, Debugger und Tracer. Diese sind in der derzeitigen STYLE Entwicklungsumgebung noch nicht implementiert. Ihre Realisierung ist Gegenstand eines laufenden Tycoon Projekts.

1.3 Ziel und Gliederung der Arbeit

Diese Arbeit befaßt sich im wesentlichen mit der Realisierung eines Editors für die graphische Modellierung von OM1 Schemata und der Integration in den oben angeführten Rahmen. Ein grundlegender Themenkomplex ist die Vorstellung der Konzepte des OM1 Datenmodells und ihrer Visualisierung in der Graphik. Ein weiterer wichtiger Komplex ist das Design des Graphikeditors, welches den Aufbau und die Funktionsweise des Graphikeditors spezifiziert.

Als Plattform für die Realisierung dient das Tycoon System. Unter Berücksichtigung der eingangs aufgeführten Anforderungen hinsichtlich Unterstützung von Benutzbarkeit, Integration und Flexibilität wird ein systematisches Vorgehen beschrieben, das folgende Aufgaben der Editorrealisierung umfaßt.

- Benutzung eines externen Werkzeugs zur Erzeugung einer graphischen Benutzerschnittstelle für den Graphikeditor;
- Erweiterung des externen Werkzeugs um erweiterbare Bausteine, die OM1 spezifische Diagramme und deren Verwaltung in die Benutzerschnittstelle integrieren;
- typsichere Einbindung der erweiterten Werkzeugfunktionalität in den uniformen Tycoon Sprachrahmen;
- Integration des Graphikeditors in die OM1 Entwicklungsoberfläche sowie Bereitstellung von Transformationsfunktionalität, die die Generierung von textueller Repräsentation aus graphischer Modellierung unterstützt.

Es folgt eine inhaltliche Übersicht der folgenden Kapitel dieser Arbeit. Die Beschreibungen sind nach Kapitelnummern und -überschriften gegliedert.

Kapitel 2. Modellierung datenintensiver Anwendungen: Es werden zunächst typische Vertreter von *semantischen Datenmodellen* beschrieben und die Vor- und Nachteile ihrer Visualisierungen bezüglich Übersichtlichkeit und Semantik [HK87] erörtert. Danach

[9] Basisfunktionen zur Objektverwaltung sind Erzeugen, Löschen und Selektieren von Objekten einer Klasse sowie Selektieren und Manipulieren von Objektattributen.

folgt die Beschreibung wesentlicher Konzepte von *objektorientierten Datenmodellen* und die Einführung des *OM1 Datenmodells* als ein konkreter Ansatz zur objektorientierten (formalen) Spezifikation datenintensiver Anwendungen. Anhand von semantischen und Gestaltungskriterien wird eine *Visualisierung* von OM1 Konzepten vorgeschlagen und durch kritische Vergleiche mit Visualisierungen anderer semantischer Datenmodelle motiviert. Dabei werden *Visualisierungstechniken* zur Förderung von Übersichtlichkeit und semantischer Gewichtung entwickelt, die auch für die graphische Modellierung in anderen Datenmodellen Verwendung finden können.

Kapitel 3. Design des Graphikeditors: In diesem Kapitel wird die Benutzerschnittstelle des Graphikeditors bezüglich optischer Gestaltung und der Dialogführung zwischen Benutzer und Editor spezifiziert. Dabei werden die durch das Datenmodell gegebenen Diagrammeigenschaften sowie verschiedene Dialogarten in Hinsicht auf die Benutzbarkeit des Editors diskutiert und berücksichtigt.

Kapitel 4. Realisierung des Graphikeditors: Zunächst werden die für die Realisierung der Benutzerschnittstelle des Graphikeditors verwendeten und erweiterten externen Dienste und Werkzeuge beschrieben. *PostScript* und *NeWS* bilden die Basis für den darauf aufsetzenden Werkzeugkasten TNT. Danach folgt die Einführung der Bibliothek *newsenv* als erweiterbare Schnittstelle von Tycoon zu den vorher genannten externen Diensten. Anhand der investiven Erweiterung der Bausteine von TNT wird gezeigt, wie durch systematisches Vorgehen unter Verwendung von objektorientierten Methoden, wie Vererbung, und Leitbildern, wie der *Werkzeug-Material* Metapher, Techniken zur effizienten Verwaltung komplexer OM1 Diagramme entwickelt werden können. Dabei wird das Ziel der Wiederverwendbarkeit von generischen Bausteinen für die flexible Unterstützung der Visualisierung weiterer Datenmodelle berücksichtigt. Es folgt die Beschreibung der Erweiterungen der Bibliothek *newsenv* zur typsicheren Einbindung der vorher entwickelten TNT Erweiterungen in Tycoon. Am Ende des Kapitels wird die Realisierung des Editors unter Verwendung von Modulen der erweiterten Bibliothek *newsenv* erläutert. Dabei liegt der Schwerpunkt auf der Darstellung der Verknüpfung von OPEN LOOK Interaktionskomponenten der Benutzerschnittstelle, wie Menüs und Kommandofenstern, mit Operationen zur Graphikverwaltung.

Kapitel 5. Integration in STYLE: Dieses Kapitel befaßt sich zum einen mit der Aufgabe der Integration von graphischer und textueller Modellierung. Es werden TL Repräsentationstypen für die Verwaltung von OM1 Graphikdaten sowie TL Funktionen zur typsicheren Datenextraktion aus Bildschirmgraphik und zur Generierung von OM1 Textrepräsentationen aus OM1 Graphik vorgestellt. Zum anderen wird die Realisierung der Integration des Graphikeditors in die Graphikoberfläche der OM1 Entwicklungsumgebung durch Funktionsaufrufe in TL Modulen gezeigt. Dabei soll insbesondere der Implementationsaufwand der Interaktion mit anderen Werkzeugen (Klasseneditor und StyleTop Editor) verdeutlicht werden.

Kapitel 6. Zusammenfassung und Ausblick: Es folgt abschließend eine Zusammenfassung und Bewertung der Arbeit bezüglich Visualisierungstechniken, Techniken zur Graphikerzeugung und Manipulation sowie bezüglich der Integrationsarbeit. Im Ausblick wird die Erweiterbarkeit der derzeitigen Implementation unter verschiedenen Kategorien, wie *Realisierung modellinhärenter Konzepte*, *flexible Unterstützung weiterer Datenmodelle* sowie *Möglichkeiten und Grenzen der Portierbarkeit*, angesprochen.

Kapitel 2

Modellierung datenintensiver Anwendungen

Ein Datenmodell stellt Konzepte zur Modellierung von Daten zur Verfügung, die die Strukturfestlegung, die Spezifikation von Abhängigkeiten und teilweise die Modellierung von Operationen unterstützen (vgl. [Wet94, S. 30 f.]). Datenbankmanagementsysteme stellen *logische Datenmodelle* in Form von Schnittstellen zur Verfügung, die die Modellierungskonzepte von Datenmodellen unterstützen und in ablauffähige Systeme einbetten. Dabei wird von der tatsächlichen Implementation abstrahiert [LS87]. Semantische Datenmodelle (*SDM*) sind entwickelt worden, um mehr Wissen über die Semantik von Datenbankanwendungen ausdrücken zu können, als dies in konventionellen Datenmodellen möglich ist [HM81].

Ein Ziel dieses Kapitels ist, neuere Konzepte sowohl auf dem Gebiet semantischer Datenmodelle als auch auf dem Gebiet objektorientierter Datenmodelle zu benennen und in dem Zusammenhang aufzuzeigen, daß semantische Datenmodelle, die auf visuellen Konzepten des *Entity Relationship* Modells (*ER Modell*) [Che76] basieren, Schwächen bezüglich der Darstellung reichhaltiger Semantik aufweisen. Ein weiteres Ziel der in dieser Arbeit vorgestellten Visualisierung objektorientierter Konzepte besteht darin, Schwächen anderer semantischer Datenmodelle bezüglich semantischer Ausdruckskraft und Übersichtlichkeit in der Wahrnehmung zu vermeiden [HK87].

Im Abschnitt 2.1 werden das ER Modell und zwei neuere semantische Datenmodelle beschrieben. Im Abschnitt 2.2 werden Konzepte der objektorientierten Datenmodellierung eingeführt. Im Abschnitt 2.3 werden die objektorientierte Modellierungssprache OM1 [Wet94] und die Visualisierung ihrer Konzepte vorgestellt. Die OM1 Visualisierung stellt eine Spezifikation des Graphikeditors hinsichtlich der von diesem zu erzeugenden und zu bearbeitenden Diagrammarten dar.

2.1 Semantische Datenmodelle

Semantische Datenmodelle bieten semantisch reichhaltige und anwendungsnahe Konzepte zur Modellierung an. Die Konzepte sollen zum einen eine präzisere Spezifikation von Anwendungen und zum anderen eine der menschlichen Konzeptualisierung nähere Modellierung unterstützen [HK87] [BMS93]. Dies gilt für das ER Modell, welches auf dem konventionellen *Relationalen Datenmodell* (RDM) als semantisches Datenmodell aufsetzt [Che76]. Die in den letzten Jahren entwickelten Konzepte der objektorientierten Datenmodellierung verlangen neue semantische

Datenmodelle zu deren Visualisierung [HK87].

Ziel dieses Abschnitts ist es, Konzepte semantischer Datenmodelle und deren Visualisierung vorzustellen. Zunächst wird einleitend das ER Modell eingeführt, dessen Konzepte durch formale Schritte in ein logisches Schema des Relationalen Datenmodells überführt werden können [LS87] [Che80]. Zwei weitere semantische Datenmodelle, die das ER Modell um neuere semantische Konzepte erweitern, werden dann beispielhaft vorgestellt. Die in den semantischen Datenmodellen vorgestellten Visualisierungen werden jeweils kurz bewertet.

2.1.1 Das Entity Relationship Modell

Das ER Modell stellt als Konzepte Gegenstandstypen (*Entities*), Beziehungstypen (*Relationships*) und Attribute zur Verfügung. Ein Gegenstandstyp klassifiziert vergleichbare Objekte der realen Welt unter einem gemeinsamen Namen. Die Vergleichbarkeit der Objekte bezieht sich auf gemeinsame, für die Anwendung signifikante Attribute, die dem Gegenstandstyp zugeordnet werden. Ein Attribut besteht aus einem Namen und einem Wertebereich. Der Wertebereich ist durch einen Wertetyp repräsentiert, dessen Domäne aus atomaren Werten besteht. Ein Beziehungstyp klassifiziert vergleichbare Beziehungen zwischen Objekten zweier Gegenstandstypen, hat einen Namen und optional weitere Attribute. Es gibt verschiedene Arten von Beziehungen, die sich durch die Kardinalität der an ihnen jeweils beteiligten Objekte aus den Gegenstandstypen unterscheiden.

- $1:1$-Beziehung: Ein Objekt des ersten Gegenstandstyps und eins des zweiten sind an der Beziehung beteiligt.
- $1:n$-Beziehung: Ein Objekt des ersten und beliebig viele des zweiten Gegenstandstyps sind beteiligt.
- $n:1$-Beziehung: Beliebig viele Objekte des ersten und eins des zweiten Gegenstandstyps sind beteiligt.
- $n:m$-Beziehung: Beliebig viele Objekte aus beiden Gegenstandstypen sind an der Beziehung beteiligt.

Dem ER Modell werden in neueren Ansätzen weitere Konzepte hinzugefügt, die zur visuellen objektorientierten Datenmodellierung verwendet werden können [TYF86] [Tha91b] [Gog94].

2.1.1.1 Visualisierung

In Abb. 2.1 ist ein ER Diagramm dargestellt, welches aus [AH87] entnommen ist und einen Ausschnitt aus einer Reiseanwendung modelliert. Es enthält verschiedene Symbole zur Visualisierung der oben genannten Konzepte. Beschriftete Rechtecke stellen Gegenstandstypen dar. Beziehungstypen werden durch beschriftete Rauten visualisiert, die über Kanten mit den an ihnen beteiligten Gegenstandstypen verbunden sind. An den Kanten stehen die Kardinalitätsangaben. Attribute von Entitätsklassen und Beziehungen sind durch Ovale dargestellt, die jeweils den Attributnamen enthalten.

2.1.1.2 Bewertung

Ein mit dem ER Modell entwickeltes Schema läßt sich über formalisierbare Übersetzungsregeln auf Relationen des Relationalen Datenmodells abbilden [TYF86]. Ein Vorteil des ER Modells

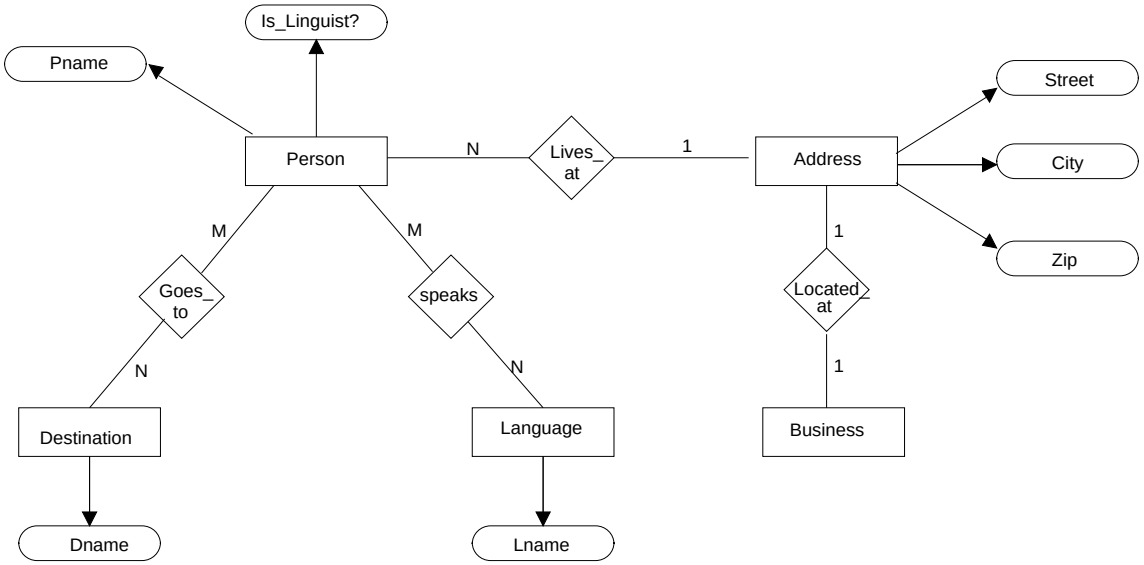


Abbildung 2.1: Ein Ausschnitt des World-Traveller Schemas als ER Diagramm

im Gegensatz zum Relationalen Datenmodell ist, daß die semantische Überladung des Konstrukts *Relation* vermieden wird [HK87] [Gog94]. Als positiv ist auch die Übersichtlichkeit von ER Diagrammen zu bewerten.

Nachteilig sind die folgenden Eigenschaften. Das ER Modell selbst ist nicht formal definiert. Somit lassen sich beispielsweise Anfragen nur im – semantisch weniger aussagekräftigen – Relationalen Datenmodell definieren. Die Modellierung von Beziehungen zwischen Gegenstandstypen durch Relationen wirkt sich nachteilig auf die Komplexität von Anfragen aus.[1] Der relationale Zugriff auf Beziehungsinformationen geht nur durch Formulierungen über oft stark geschachtelte Joinbildungen [HK87]. Ein weiterer Nachteil besteht darin, daß in den visualisierten Beziehungen keine Richtungen der Beziehungen dargestellt werden können.

2.1.2 HERM

In [Tha91b] wird das *Higher order Entity Relationship Model* (HERM) formal definiert. Es umfaßt folgende, über das ER Modell hinausgehende Konzepte.

- Beziehungen höherer Ordnung:
  Dieses Konzept schließt die Modellierung von Beziehungen zwischen mehr als zwei Gegenstandstypen und von hierarchischen Beziehungen[2] zwischen Objekten der zu modellierenden Anwendung ein.

- Beziehungen von Beziehungen:
  In einem Beziehungstyp werden nicht nur beteiligte Objekte aus anderen Gegenstandstypen klassifiziert, sondern auch Beziehungen.

[1] Nur Beziehungen, die in n:1-Richtung modelliert werden und keine Attribute enthalten, werden nicht als Relationen modelliert, sondern als Erweiterung der die „n-wertige“ Entitätsklasse modellierenden Relation um ein Fremdschlüsselattribut.

[2] Die dahinterstehenden semantischen Konzepte *Spezialisierung* und *Generalisierung* werden in Abschnitt 2.2 im Zusammenhang mit objektorientierten Konzepten eingeführt.

- Typkonstruktoren zur Spezifikation von mehrwertigen und geschachtelten Attributen: Durch Mengenbildung und Aggregation sind für Attribute auch nicht atomare, geschachtelte Typen als Wertebereiche definierbar.

2.1.2.1 Visualisierung

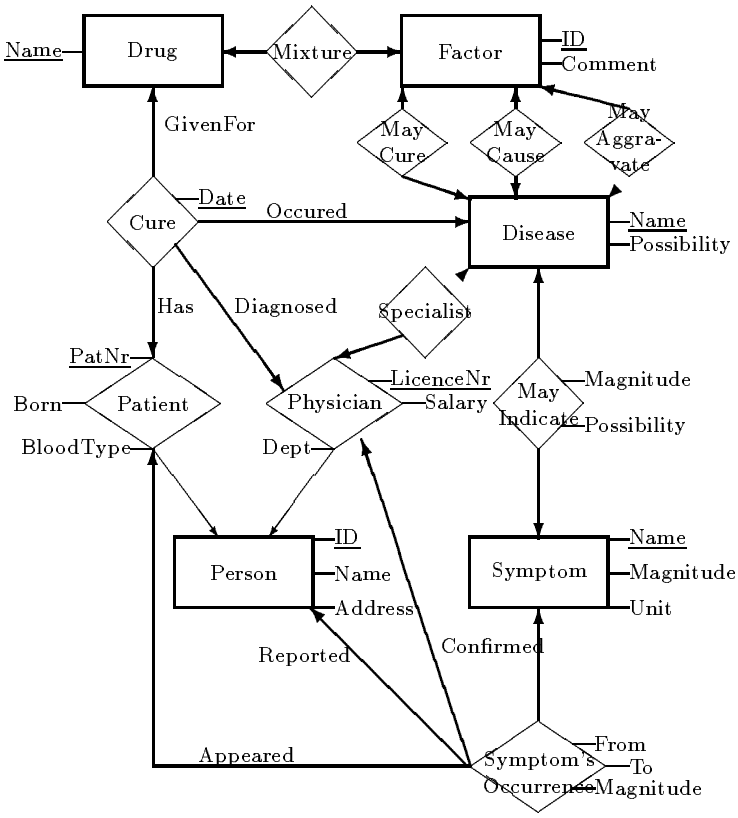


Abbildung 2.2: Die Modellierung einer Krankenhausverwaltung als HERM Diagramm

In dem Beispieldiagramm in Abb. 2.2, welches aus [Tha91b] stammt, wird die Verwaltung eines Krankenhauses modelliert. Es wird deutlich, daß im Vergleich zum ER Modell keine weiteren Symbole zur Visualisierung von neuen Beziehungsarten benutzt werden. Spezialisierungen von Gegenstandstypen sowie Beziehungen von Beziehungen werden jeweils durch Rauten und Kantenverbindungen visualisiert. Typkonstruktoren zur Modellierung von Attributwertebereichen werden textuell durch (geschachtelte) Klammern dargestellt, die Namen der Basistypen enthalten.
Abb. ...

2.1.2.2 Bewertung

Die Diagramme sind aus folgenden Gründen unübersichtlich.

- Semantische Überladung von Beziehungssymbolen für verschiedene Konzepte;

- Kombination von textuell formulierter mit symbolhaft ausgedrückter Semantik;[3]
- verwirrende Anordnung von Symbolen und Kantenüberschneidungen, die die Lesbarkeit erheblich beeinträchtigen [LM92].

2.1.3 Das IFO Modell

IFO wird in [HK87] als mathematisch formal definiertes semantisches Datenmodell vorgestellt, welches verschiedene semantische Konzepte beinhaltet. Die dazu korrespondierenden Visualisierungen sind optisch gut voneinander unterscheidbar. In IFO wird nicht zwischen Basistypen, komplexen Typen und Gegenstandstypen unterschieden. Sie werden alle als *Objekttypen* (*object types*) bezeichnet. Attribute, analog zu den Attributkonzepten im ER Modell und HERM, werden durch Funktionen definiert, die auf andere Objekttypen abbilden. Beziehungsarten und Typkonstruktoren werden näher erläutert.

- Funktionale Beziehungen gehen von nicht atomaren Objekttypen aus und können alle Arten von Objekttypen als Wertebereiche haben.
- Vererbungsbeziehungen werden in *Generalisierung* und *Spezialisierung* unterschieden. Es werden verschiedene Vererbungsmechanismen und zusätzliche Integritätsbedingungen[4] graphisch repräsentiert.
- Komplexe Objekte werden durch Typkonstruktoren für Mengen und Aggregationen gebildet.
- Es wird zwischen druckbaren, atomaren und abstrakten Objekttypen unterschieden.

Aggregation wird sowohl für Wertaggregation als auch für Gegenstandsaggregation verwendet.[5] In [LS87] werden diese Aggregationsarten semantisch unterschieden.

2.1.3.1 Visualisierung

Für die Beziehungsarten, die Typkonstruktoren und verschiedenen Objekttypen gibt es je ein Symbol.

- Funktion: schmaler Pfeil;
- Spezialisierung: breiter, schwarz gefüllter Pfeil;
- Generalisierung: breiter, weiß gefüllter Pfeil;
- Mengenkonstruktor: *Wagenrad*-Symbol;
- Aggregationskonstruktor: Kreis mit Kreuz;
- druckbare Typen: Quadrat;
- nicht druckbare Tupen: leerer Kreis;
- abstrakte Typen: Raute.

---

[3] Typkonstruktoren, die sich aus Platzgründen nicht mehr durch Symbole ausdrücken lassen, werden durch Zeichenketten dargestellt. Dies erfordert zusätzlichen Interpretationsaufwand beim „Lesen“ der Graphik.

[4] Dies sind *Überdeckung* einer Generalisierung (alle Elemente der Objektmengen aus zu generalisierenden Objekttypen sind in der Objektmenge des generellen Objekttyps vertreten) und *Disjunktheit* von Objektmengen verschiedener Spezialisierungen einer Superklasse.

[5] Aggregation ist hier ein Konzept von Relationstypen, also von kartesischen Produkten von Relationsattributen, die hier als Aggregatkomponenten definiert werden (vgl. [HK87, S. 530]).

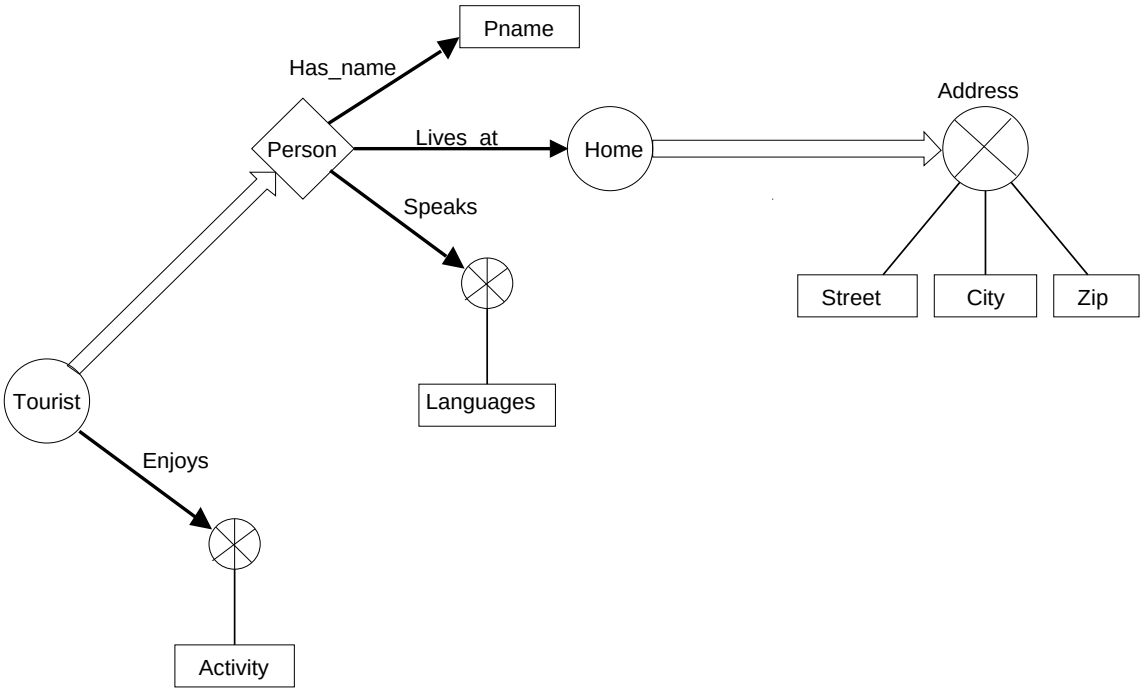


Abbildung 2.3: Teil des World-Traveller Schemas als IFO Diagramm

2.1.3.2 Bewertung

Vorteilhaft an der Visualisierung von IFO ist, daß die Nachteile, die beispielsweise in HERM durch semantisches Überladen von Symbolen entstehen, vermieden werden. Verschiedene Konzepte werden durch unterschiedliche Symbole dargestellt. Dies erhöht die Verständlichkeit der graphischen Darstellung.

Nachteilig ist jedoch, daß Gegenstände der modellierten Anwendung visuell keinen größeren Stellenwert als Werte besitzen, obwohl sie semantisch höher zu bewerten sind. Dasselbe betrifft die Gleichbehandlung von Gegenstands- und Wertaggregation. Außerdem führen zu viele Symbolarten in einem Diagramm zu dessen *visueller Überladung*. Die Lesbarkeit verringert sich aufgrund der Komplexität des Diagramms (vgl. [LM92, S. 21 ff.]).

2.2 Objektorientierte Datenmodelle

Ziel dieses Abschnitts ist die Einführung der Konzepte der objektorientierten Datenmodellierung (OODM). Zur Zeit existiert noch keine einheitliche Definition für objektorientierte Datenmodelle. In [Heu92] werden jedoch folgende Kategorien für die Spezifikation von objektorientierten Datenbankanwendungen aufgelistet.[6]

- Komplexe Objekte:
  Es sind nicht nur atomare Datentypen als Domänen für Attribute von Objekten erlaubt, sondern auch geschachtelte Typen durch wiederholte Anwendung von Typkonstruktoren (Mengen, Tupel, Listen, Arrays).

[6] Die auch in traditionellen Datenbanksystemen vorhandenen Kategorien, wie Persistenz, Transaktionen, Sekundärspeicherverwaltung und Anfragesprachen, werden hier nicht näher betrachtet.

- Objektidentität:
  Objekte existieren unabhängig von den Werten ihrer Eigenschaften. Werte (von Attributen) eines Objekts können sich ändern, während seine Identität während der Lebenszeit unveränderlich bleibt. Ein Objekt kann als Element in mehreren Klassenkollektionen auftreten [Bee92a].

- Kapselung:
  Attribute von Objekten sowie Methoden[7] werden gekapselt.

- Klassenhierarchien:
  Klassen, die Kollektionen von Objekten definieren, können jeweils in Vererbungsbeziehungen stehen. Eine Subklasse erbt alle Attribute und Methoden ihrer Superklasse und kann neue hinzufügen.

In einigen Datenmodellen, z.B. IFO, wird nicht zwischen Objekten (*Entitäten*) und Werten bzw. den sie definierenden Klassen (*Gegenstandstypen*) und Wertetypen unterschieden. Ferner ist zwischen dem Klassenbegriff und der *Klassenextension* zu trennen. In [Bee92a] werden diese Unterschiede verdeutlicht. Sie werden in den folgenden beiden Abschnitten zusammengefaßt.

2.2.1 Objekt-Wert Unterscheidung

In [Bee92a] werden Werte als Elemente von Typen und Operationen auf Werten durch die algebraische Spezifikation ihrer Typen vorgegeben. Objekte als Kollektionen von Klassen müssen explizit erzeugt werden. In den ihnen zugeordneten Klassen sind Operationen anwendungsspezifisch definiert. Nur Standarddatenbankoperationen, wie Einfügen, Löschen und Lesen, sind vorgegeben.

Typen und Klassen haben verschiedene Charakteristika. Typen sind algebraisch spezifiziert und lassen sich wie folgt unterteilen.

- Atomare Basistypen, wie Int, Real, Bool;

- atomare abstrakte Objekttypen, die als Domäne für die Identifizierung von Objekten dienen;

- nicht atomare Typen, die rekursiv über Typkonstruktoren gebildet und deren Komponenten über typkonstruktorabhängige Selektionsfunktionen zugegrifen werden.

Eine Klasse ist mit einem abstrakten Typ assoziiert. Objekte einer Klasse besitzen alle diesen Typ. Ferner werden in [Bee92a] *Attribute* als unäre Funktionen charakterisiert, die in Klassen definiert sind. Jedem Objekt, welches zur Kollektion einer Klasse gehört, wird über seine Objektidentität eine damit assoziierte Liste von Attributen als Wert eines Tupels zugewiesen. Attributwerte können typwertig sein oder referenzierend (über *Objektidentifikatoren*) andere Objekte identifizieren.

2.2.2 Klassenbegriff und Extension

In objektorientierten Datenmodellen schließt der Klassenbegriff in Anlehnung an Relationen im Relationen Datenmodell die Extension von Datenbankinstanzen ein. Modellinhärente Integritätsbedingungen müssen für Operationen auf diesen Datenbankinstanzen zugesichert werden

[7] Dies sind im Datenbankkontext Standardmethoden zur Erzeugung, Manipulation und zum Zugriff von Objekten sowie anwendungsspezifische (benutzerdefinierte) Methoden.

[Bro84]. Dies betrifft die referentielle Integrität sowie Bedingungen hinsichtlich Vererbungshierarchien von Klassen. Wird beispielsweise ein Objekt in einer Klasse neu erzeugt, so muß es auch in alle Extensionen der Superklassen dieser Klasse eingefügt werden. Auch der Wechsel von Objekten zwischen verschiedenen Klassen muß durch Operationen ermöglicht werden [ABGO93].

2.3 Das OM1 Datenmodell

Das objektorientierte Datenmodell OM1[8] umfaßt eine formale Spezifikationssprache, die Konzepte semantischer und objektorientierter Datenmodellierung beinhaltet [Wet94]. Ziel dieses Kapitels ist, die Konzepte von OM1 und graphische Konzepte zu deren Visualisierung vorzustellen und damit eine Grundlage für die Kombinierbarkeit von textueller und graphischer Modellierung zu schaffen. Die im Zusammenhang mit der Visualisierung eingeführten Symbole und Graphiken stellen eine Spezifikation für vom Graphikeditor zu bearbeitende Diagramme dar (vgl. Abschnitt 2.3.2).

Im Abschnitt 2.3.1 werden die sprachlichen Konzepte[9] von OM1 erläutert, um einen Einblick in die Syntax und Semantik zu geben. Eine ausführliche Einführung in OM1 ist in [Wet94] zu finden. Der Abschnitt 2.3.2 erklärt die zu den sprachlichen Konzepten korrespondierenden Visualisierungen. Diese beinhalten Symbole zur graphischen Repräsentation von OM1 Konzepten sowie Diagrammbeispiele, die OM1 Schemata graphisch darstellen. Anhand von Diagrammen werden die Anordnungsmöglichkeiten der Symbole erörtert. Der Abschnitt endet mit einer Bewertung der Visualisierung.

2.3.1 Sprachliche Konzepte

In diesem Abschnitt werden die OM1 Konzepte *Typen*, *Klassen* und die in den Klassen enthaltenen *Komponenten* eingeführt. Die Beispiele sind dem in Anhang A.3 vollständig angegebenen OM1 Schema *TRACY* (*TRavel AgenCY*) entnommen. Es modelliert ein Reisebuchungssystem. Diese Anwendung wurde im Rahmen des STYLE Projekts in Anlehnung an das in [Den91] eingeführte Beispiel *TuBSy* in OM1 modelliert [Koe94].

2.3.1.1 Typ

Neben den in der Sprache OM1 enthaltenen *Basistypen* (*Int*, *Bool* und *String*) gibt es *benutzerdefinierte Typen*, die benannt und durch folgende Typkonstruktoren definiert werden können.

- Recordkonstruktor **Record** ...**end** zur Aggregation von Komponenten,
- Optionskonstruktor **Option** ...**end** zum Aufbau varianter Typen,
- Mengenkonstruktor **SetOf**(...) für Mengentypen,
- Listenkonstruktor **ListOf**(...) für Listentypen,
- Feldkonstruktor **Array** ...**of** ...**end** zum Aufbau von Feldern.

Es folgt ein Kodebeispiel mit Typdefinitionen.

[8] Object Model 1

[9] Konzepte, die in später vorgestellten Beispielen und in den OM1 Diagrammen keine oder nur eine untergeordnete Rolle spielen bzw. in der *STYLE* Umgebung nicht realisiert sind, werden nur kurz erwähnt.

Type *FlightNo = Int*
Type *Date =* ***Record*** *day : Int, month : Int, year : Int* ***end***
Type *ReservationRec =* ***Record*** *booked : Int, vacant : Int* ***end***
Type *Quotatable =* ***Array*** *Date* ***of*** *ReservationRec* ***end***

Das obige Beispiel zeigt Definitionen von benutzerdefinierten Typen. *FlightNo* ist ein atomarer Typ. *Date* und *ReservationRec* sind zwei *Recordtypen*, die in dem geschachtelten Typ *Quotatable* für die Definition von Indextyp und Elementtyp des Feldes verwendet werden.

2.3.1.2 Klasse

Class Klassenname
Parameter
Parameterdefinitionen ermöglichen die Spezifikation generischer Klassen.
Instantiation
Die Instanziierungskomponente steht ebenfalls im Zusammenhang mit generischen Klassenspezifikationen und legt Instanziierungen der Klassenparameter sowie Umbenennungen von Attributnamen fest.
Relationships
In der Beziehungskomponente werden Subklassenbeziehungen sowie Existenzabhängigkeiten zwischen Klassen festgelegt.
Structure
In der Strukturkomponente wird die strukturelle Beschreibung von Objekten vorgenommen, diese kann Regeln für abgeleitete Attribute enthalten.
Constraints
In der Bedingungskomponente werden statische und Transitionsbedingungen definiert, die i.a. klassenübergreifend sind.
Methods
In der Methodenkomponente werden Objektmethoden spezifiziert, diese benutzen implizite generische Basismethoden oder überschreiben diese.
End

Abbildung 2.4: Komponentenstruktur einer OM1 Klassendefinition

Es wird zunächst auf die Bedeutung des Klassenbegriffs eingegangen. Dann folgt die Beschreibung der *Syntax*. Die in einer Klasse enthaltenen Subkomponenten werden in weiteren Sektionen

behandelt.

Bedeutung: Eine Klasse klassifiziert die Objekte, die in ihren Eigenschaften und in ihrem Verhalten vergleichbar sind [Wet94]. Die in der Klasse definierten Attribute und Methoden charakterisieren jedes Objekt, welches der Klasse angehört. Objekte werden im folgenden näher beschrieben.

Jedes Objekt gehört (mindestens) einer Klasse an, d.h. Objekte können nicht außerhalb von Klassen existieren. Ein Objekt kann seine Struktur ändern, z.B. durch Hinzufügen neuer Attribute und Methoden, was zu einem Wechsel in der Klassenzugehörigkeit führt.[10] Jedes Objekt hat, unabhängig von seiner Klassenzugehörigkeit und seinen Werten, eine *Identität*, die sich nicht ändert, solange das Objekt existiert. Diese wird vom System generiert und bleibt dem Benutzer verborgen. Ein Objekt wird vom Benutzer über Attributwerte identifiziert. Dazu können gewisse Attribute in der Klasse als *Schlüsselattribute* gekennzeichnet werden. Die Kombination der Werte der Schlüsselattribute identifiziert eindeutig ein Objekt der Klasse.

Syntax: Eine Klassendefinition in OM1 setzt sich aus Komponenten zusammen, die in Abbildung 2.4 (aus [Wet94]) aufgeführt sind. Jede Komponente wird durch Angabe eines Schlüsselwortes eingeleitet. Die Komponenten sind optional, d.h. nur die benötigten müssen durch Schlüsselwörter benannt werden. Auf die Komponenten ***Parameter*** und ***Instantiation*** wird hier nicht eingegangen.[11] Die übrigen Komponten für Strukturen (***Structure***), Beziehungen (***Relationships***), Integritätsbedingungen (***Constraints***) und Methoden (***Methods***) werden im folgenden beschrieben.

In der Strukturkomponente (***Structure***) einer Klasse werden nach dem Schlüsselwort ***Attributes*** die Attribute definiert. Eine Attributdefinition setzt sich aus dem *Attributnamen* und dem *Attributwertebereich* zusammen. Als Attributbereich können beliebig strukturierte Typausdrücke verwendet werden, die zusätzlich auch Klassennamen (Klassenreferenzen) enthalten können [Bee92a]. Typen sind entweder Basistypen oder benutzerdefiniert.

```
Class Tour
  Structure
    Attributes
    key tourNo : Int,
    constant key country : ref Country,
    travelTime : Record start : Date, till : Date end
End
```

Das obige Beispiel zeigt die Syntax von Attributdefinitionen in einer Klasse. Die darin verwendeten Schlüsselwörter werden erklärt.

- ***ref*** kennzeichnet ein objektwertiges Attribut (eine Referenz), das auf eine Klasse als Attributbereich weist.
- ***key*** kennzeichnet alle Attribute einer Klasse, die den Schlüssel für die Objektidentifizierung bilden.

[10] Die Änderung von *Attributwerten* führt zu keinem Wechsel der Klassenzugehörigkeit.

[11] Diese werden nicht in späteren Modellierungsbeispielen verwendet bzw. sind derzeit nicht in der STYLE Umgebung implementiert.

- ***constant*** bezeichnet ein Attribut, dessen Wert nicht änderbar ist. Weitere Schlüsselwörter für inverse, abgeleitete, partielle und redefinierte Attributarten sind in [Wet94] näher erklärt.
- ***Record***, ***SetOf***, ***ListOf***, ***Array*** und ***Option*** bezeichnen Typkonstruktoren, die für die Definition von Attributbereichen verwendet werden. Sie werden in derselben Weise wie in benutzerdefinierten Typen benutzt.

In der Beziehungskomponente (***Relationships***) sind neben *Existenzabhängigkeiten*[12] drei Arten von *Subklassenbeziehungen*[13] in der Subkomponente **Specialization** definiert. Das Schlüsselwort ***isA*** definiert eine *intensionale* und *extensionale Spezialisierung*, d.h. jedes Objekt der spezialisierten Klasse erbt alle Attribute seiner Superklassen und ist in der Extension aller Superklassen enthalten.

```
Class PackageTour
  Relationships
    isA Tour
  Structure
    Attributes
    area : ref Area,
    hotel : ref Hotel,
    town : ref Town,
    flightForth : ref Flight,
    flightBack : ref Flight
End
```

Das obige Beispiel zeigt eine Klassendefinition der Klasse *PackageTour*, die eine Spezialisierung der Klasse *Tour* (siehe voriges Kodebeispiel) ist.

In der Integritätskomponente (***Constraints***) sind vier Arten von Integritätsbedingungen spezifizierbar. Statische Integritätsbedingungen gelten in jedem Datenbankzustand. Dynamische Integritätsbedingungen gelten in jedem Zustandsübergang. Anfangsbedingungen gelten unmittelbar nach dem Erzeugen eines Objekts. Endbedingungen gelten unmittelbar vor dem Entfernen eines Objekts. Es folgt ein Beispiel, in dem eine statische Integritätsbedingungen definiert wird.

```
Class PackageTour
  Structure
    Attributes
      ...
    hotel : ref Hotel,
    flightForth : ref Flight,
    flightBack : ref Flight
  Constraints
    static
    HotelFlight:
    (this.hotel.area is this.flightForth.destAirport.area) AND
```

[12] *Dependencies* spezifizieren Existenzabhängigkeiten auf Referenzbeziehungen zwischen Klassen. Dadurch werden *Weak Entities* charakterisiert (vgl. [LS87]). Ein Beispiel dafür ist in [Wet94, S. 198] angeführt.

[13] Arten von Subklassenbeziehungen sind *isA*, ***isSubClassOf*** und *inheritsStructureFrom* (vgl. [Wet94, S. 55]).

*(**this**.hotel.area **is this**.flightBack.depAirport.area)*
End

Die Bedingung *HotelFlight* besagt, daß das referenzierte Hotel eines Objekts der Klasse *PackageTour* in derselben Region liegen muß wie die Flughäfen der referenzierten Hin- und Rückflüge.

In der Methodenkomponente (***Methods***) werden benutzerdefinierte Methoden oder Transaktionen spezifiziert. Syntaxbeispiele für Methodendefinitionen finden sich in [Wet94]. Standarddatenbankmethoden[14] werden in OM1 nicht spezifiziert, sondern automatisch unter Sicherstellung der modellinhärenten und ausgewählter Klassen benutzerdefinierter Integritätsbedingungen in TL generiert [Wet94] [Mü94a]. Die generierten Standarddatenbankmethoden können bei der Implementation der benutzerdefinierten Methoden verwendet werden.

2.3.2 Visuelle Konzepte

Ziel dieses Abschnitts ist, zu den im vorigen Abschnitt eingeführten sprachlichen Konzepten *Visualisierungen* und den Aufbau der durch sie erzeugbaren OM1 Diagramme vorzustellen. Es soll verdeutlicht werden, nach welchen semantischen und gestaltpsychologischen Gesichtspunkten die optische Form der Symbole sowie deren Gewichtung und Anordung in den Diagrammen gewählt wird.

Im ersten Abschnitt werden Richtlinien für das Design der Visualisierung angegeben. Der zweite Abschnitt führt Basissymbole für die Visualisierung der OM1 Konzepte ein. Die sich aus den Basissymbolen zusammensetzenden OM1 Diagramme werden im dritten Abschnitt erklärt. Im letzten Abschnit wird die Visualisierung bewertet.

2.3.2.1 Richtlinien für die Diagrammgestaltung

An die Visualisierung einer Anwendung in einer Graphik sind folgende Anforderungen zu stellen. Sie soll übersichtlich und leicht verständlich sein [Obe92] [Mar92]. In ihr sind wesentliche Zusammenhänge der Anwendung zu repräsentieren [HK87]. Zur Erfüllung dieser Anforderungen lassen sich aus Erfahrungen mit Diagrammen anderer Datenmodelle und unter Berücksichtigung gestaltpsychologischer Ansätze folgende Richtlinien aufstellen [Bal88] [Gal89].

- Vermeidung von *semantischer Überladung*:
  Jedes graphische Symbol soll nur *ein* semantisches Konzept der Anwendung visualisieren. Ähnliche, leicht verwechselbare Symbole für unterschiedliche Konzepte sind zu vermeiden. Symbole mit derselben Semantik sollen im Diagramm visuell schnell als *Gruppe* erfaßbar sein.

- Vermeidung von *optischer Überladung*:
  Es dürfen nicht zu viele verschiedene Symbolarten in einem Diagramm verwendet werden.

- *Optische Gewichtung*:
  Formen von Symbolarten sind nach Relevanz der durch sie repräsentierten Konzepte zu wählen.

[14] Das sind Methoden (create, remove, lookup, elements) zum Erzeugen, Löschen und Suchen von Objekten und Iterieren über die Elemente der Klassenextension sowie get- bzw. set-Methoden zum lesenden bzw. schreibenden Zugriff auf die einzelnen Attribute.

- Vermeidung von störenden optischen Effekten bei der Anordnung von Symbolen:
  Durch sich kreuzende Kanten können ungewollte geometrische Formen entstehen, z.B. Rechtecke. Zu lange Kanten können eine Graphik optisch zerschneiden. Zu eng nebeneinander liegende oder sich überschneidende Symbole im Diagramm stören die Lesbarkeit und das navigierende Erfassen seitens des Betrachters.

Um die genannten Richtlinien zu erfüllen, werden für die Visualisierung Prinzipien ausgewählt, die nach folgenden zwei Gesichtspunkten geordnet sind.

Förderung selektiver Wahrnehmung: Dies kann durch die Vorgabe eines rasterähnlichen Layouts für die Anordnung der Symbole erzielt werden. Die Übersichtlichkeit wird erhöht, und einige störende optische Effekte werden vermieden. Außerdem kann die Unterscheidung verschiedener Symbolarten durch Größe, Form und Farbe die menschliche Wahrnehmung von Diagramminhalten erleichtern. In [NS86] [Sta87] [Ber88] [LM92] werden Gestaltgesetze vorgestellt, die Gesetzesmäßigkeiten menschlicher Wahrnehmung[15] und optischer Hervorhebung[16] festlegen.

Begrenzung der Knoten- und Kantenarten in einem Diagramm: Dies kann durch folgende Maßnahmen erreicht werden.

- Beschränkung auf *notwendige* optische Unterscheidungen von semantisch verschiedenen Konzepten. Orthogonal kombinierbare OM1 Konzepte sollen durch Kombinationen der korrespondierenden Visualisierungen optisch repräsentiert werden.
- Einführung verschiedener *Sichten*, die es erlauben, mehrere Darstellungsebenen eines Diagramms – mit jeweils unterschiedlicher Granularität – anzuzeigen. Damit wird dem Benutzer ermöglicht, zwischen Übersichtlichkeit (in der *Standardsicht*) und Vollständigkeit (in der *erweiterten Sicht*) zu fokussieren.
- Separierung unterschiedlicher Information durch Aufteilung der Visualisierung einer Anwendung in *verschiedene Diagrammarten*.

Die folgenden beiden Abschnitte nehmen bei der Einführung der Symbolarten und Diagramme Bezug auf die hier aufgestellten Kriterien und Lösungen.

2.3.2.2 Basissymbole

In der Tabelle 2.1 werden OM1 Konzepte und deren Visualisierungen gegenübergestellt. Die ersten fünf in der Tabelle unterschiedenen Symbolarten werden der Reihenfolge nach als *Klassenknoten*, *Referenzkante*, *Vererbungspfeil*, *Typsymbol* und *Werteattributkante* bezeichnet. Die Formen und Größen der Symbole sind entsprechend der Unterschiedlichkeit und Relevanz der OM1 Konzepte gewählt.

- Klassen sind von Typen verschieden und haben ihnen gegenüber Priorität. Entsprechend sind Klassenknoten anders geformt und größer als Typsymbole.
- Werte und Objekte sind verschieden. Entsprechend werden Werteattribute von Klassen anders visualisiert als Referenzen. Werteattributkanten unterscheiden sich in äußerer Form und Zusammensetzung von Referenzkanten.

[15] Dies sind Wahrnehmungsgesetze bzgl. der Dimensionen Aufmerksamkeitsverteilung, Geschlossenheit, Symmetrie und visuelle Gruppierung.

[16] Optische Hervorhebung wird erzielt durch die Faktoren Selektivität, Assoziativität, Anordnung und Quantitativät bei Wahl verschiedener graphischer Dimensionen.

- Referenzen und ***isA***-Beziehungen sind verschiedene Beziehungsarten zwischen Klassen. Die Visualisierungen verwenden deshalb verschiedene Pfeilarten. Vererbungspfeile weisen von der Subklasse auf die Superklasse.

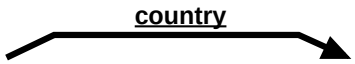
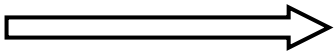
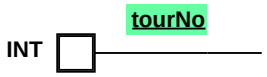
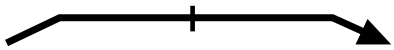
Klasse	
Referenz zwischen Klassen	
isA-Beziehung zwischen Klassen	
Basistypen	
Werteattribut mit Typangabe	
mengenwertige Referenz	
injektive Referenz	
partielle Referenz	
Indikator für nicht angezeigte Information	

Tabelle 2.1: OM1 Konzepte und deren Visualisierung

Drei weitere Symbolarten sind mit Referenzkanten kombinierbar, um Integritätsbediungen für Referenzen anzuzeigen.

- Das *Wagenrad*-Symbol symbolisiert graphisch die Mengenwertigkeit einer Referenz; auch in anderen Datenmodellen wird dieses Symbol für diesen Zweck verwendet (vgl. [Heu92] und /citeHuKi87). Es hebt durch seine auffällige Form eine mengenwertige Referenz optisch hervor.

- Der *Partialstrich* und die *Injektivspitze* sind übliche Symbole zur Kennzeichnung von Eigenschaften auf Funktionen (vgl. z.B. [A+91]).

- Das *Rauten*-Symbol wird in Anlehnung an aufklappbare *Black Boxes* verwendet, um auf versteckte graphische Information hinzuweisen, die ausgeblendet ist, aber interaktiv eingeblendet werden kann. Das Symbol wird zur Anzeige einer Diagrammsicht (der *zweiten Ebene* – siehe Abschnitt 2.3.2.3) verwendet.

Ein weiteres visualisierbares OM1 Konzept ist die *Schlüsseleigenschaft* von Attributen. Diese wird durch *unterstrichene Attributnamen* ausgedrückt. Die in der Tabelle 2.1 aufgeführten Attributnamen *country* und *tourNo* kennzeichnen die Referenz und das Werteattribut jeweils als Teil des Schlüssels.

Bei der Wahl der Basissymbole werden die Richtlinien aus Abschnitt 2.3.2.1 wie folgt beachtet. Durch unterschiedene Symbole für verschiedene OM1 Konzepte wird das semantische Überladen vermieden. Durch unterschiedliche Größen der Symbole besteht eine Gewichtung entsprechend der Relevanz der Konzepte. Die Kombinierbarkeit der Schlüsselvisualisierung bzw. des Mengensymbols sowohl mit Referenzkanten als auch mit Werteattributkanten erhöht die Lesbarkeit.[17] In [LM92] und im Kapitel 6 werden Visualisierungen weiterer OM1 Konzepte, z.B. Typkonstruktoren, beschrieben.

2.3.2.3 Diagramme

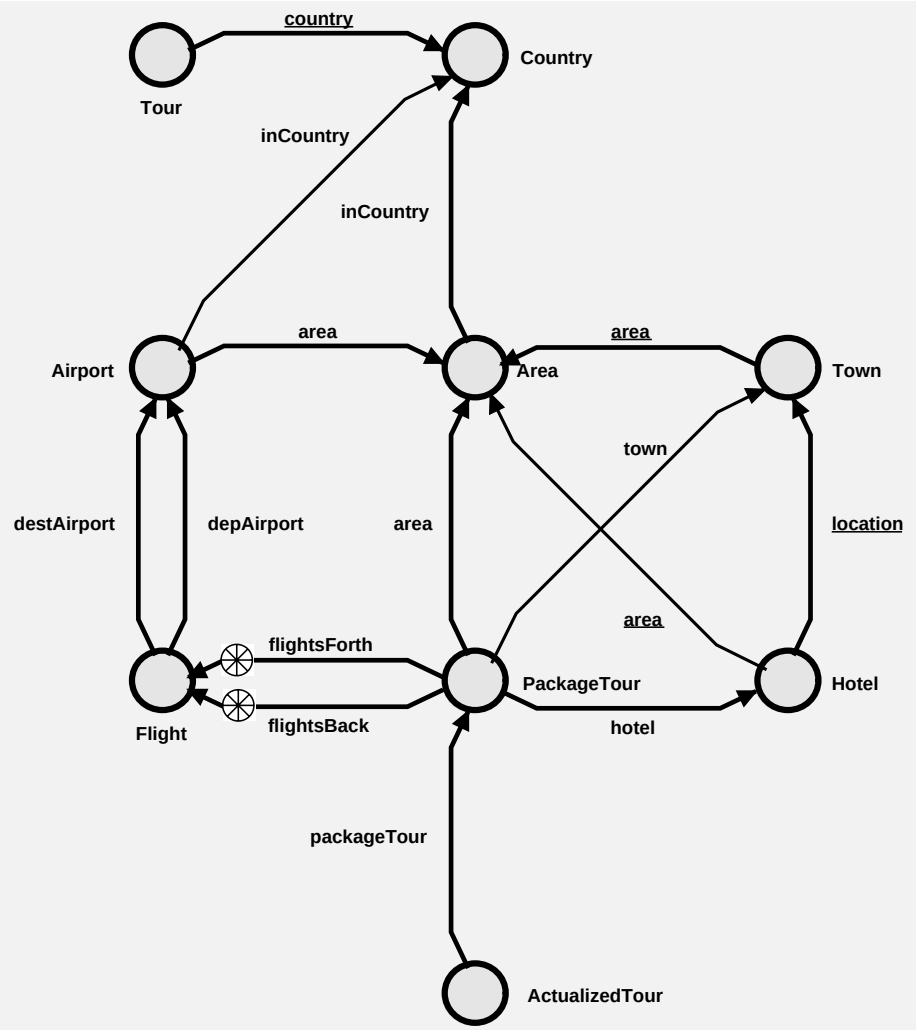


Abbildung 2.5: Visualisierung der TRACY Anwendung in der Ebene 1 (Standardsicht)

Im vorigen Abschnitt wurde gezeigt, daß durch die Wahl unterschiedlicher Symbole ein semantisches Überladen von Diagrammen vermieden wird. Bei der nun folgenden Vorstellung des Designs von Diagrammen, die sich aus den Basissymbolen zusammensetzen, soll verdeutlicht werden, wie die im Abschnitt 2.3.2.1 vorgestellten Richtlinien eingehalten und die formulier-

[17] Das Prinzip der orthogonalen Kombination von Symbolen verringert deren Anzahl und erhöht deren Wiedererkennung.

ten Lösungen konkret umgesetzt werden. Zunächst werden unterschiedliche Visualisierungssichten eines Schemas vorgestellt, dann wird die Trennung von anzuzeigender Information durch Einführung verschiedener Diagramme beschrieben.

Die Standardsicht (Ebene 1) in Abb. 2.5 veranschaulicht, daß es jeweils *eine Art* von Knoten und Kanten gibt. Diese Sicht soll einen leicht erfaßbaren Überblick über die Modellierung einer Anwendung geben. Es werden *Klassen* und *Referenzen* als wichtige Beziehungen zwischen Klassen dargestellt. *Werteattribute* und Vererbungsbeziehungen werden hier nicht repräsentiert.

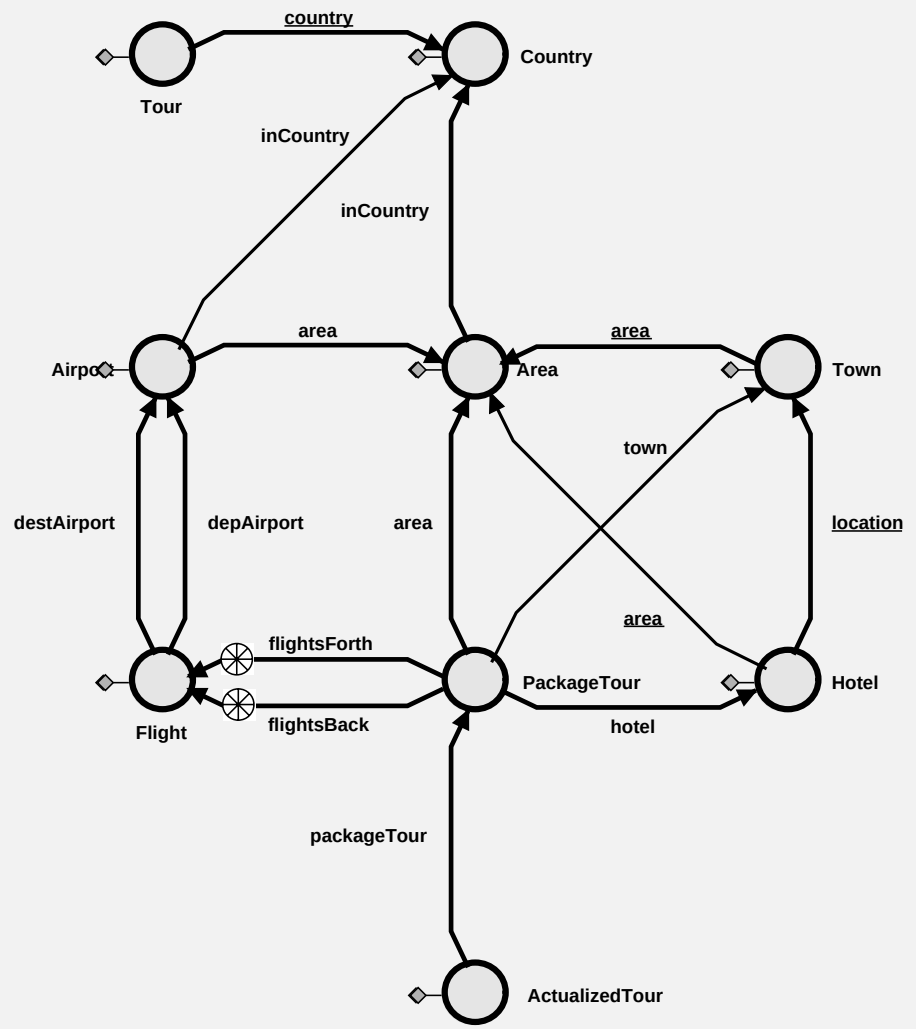


Abbildung 2.6: Visualisierung der TRACY Anwendung in Ebene 2

Durch *Beschränkung der topologischen Anordnung* ist das Diagramm übersichtlich gestaltet. Klassenknoten sind auf dafür vorgesehenen Rasterpunkten plaziert, Referenzkanten liegen zwischen im Raster *benachbarten* Klassenknoten. Den Vorteilen der leichten Erfaßbarkeit und des einfachen optischen Navigierens über Verbindungen des Diagramms steht der Nachteil der Restriktion in der Anordnung von Klassenknoten und Referenzkanten gegenüber. Durch die Alternative der *erweiterten Rasterung*[18] wird dieser Nachteil zum Teil wieder aufgehoben. Die

[18] Eine engere Rasterung mit mehr Rasterpunkten ermöglicht mehr Freiheitsgrade in der Anordnung von Klas-

Vor- und Nachteile von Restriktionen der Anordnung sowie der Alternativen dazu werden in der Bewertung im Abschnitt 2.3.2.4 gegenübergestellt.

Erweiterte Sichten (Ebene 2 und 3) zeigen Zusatzinformationen in unterschiedlicher Detaillierung. Abb. 2.6 zeigt die *Sicht der Ebene 2*. Es sind zusätzlich zur Standardsicht Rautensymbole vorhadnen, die auf nicht angezeigte Werteattribute von Klassen hinweisen. Sie sind einzeln aufklappbar. Dem Benutzer wird dadurch mitgeteilt, wo zusätzliche Informationen vorliegen.

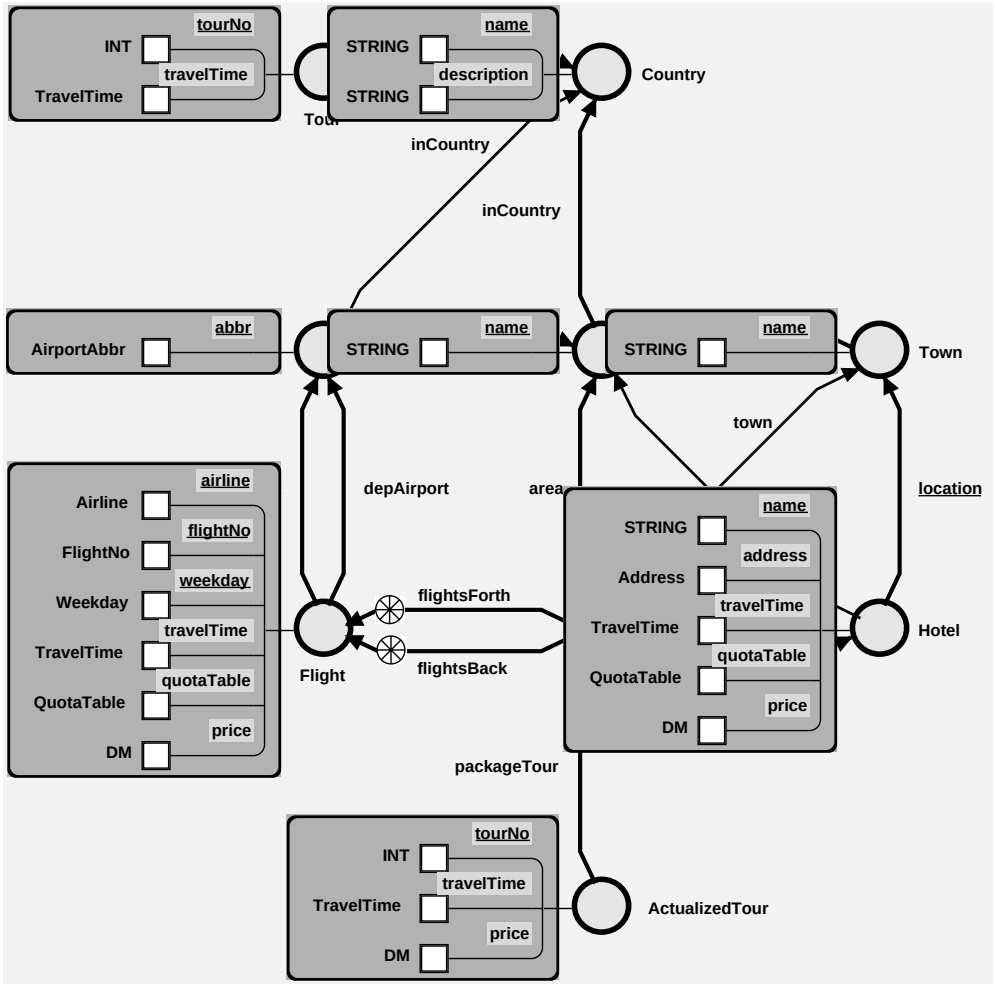


Abbildung 2.7: Visualisierung des TRACY Anwendung in Ebene 3

Die Abb. 2.7 stellt die *Sicht der Ebene 3* dar. Es werden neben Klassen und Referenzen die Werteattribute angezeigt. Um die einer Klasse zugeordneten Werteattribute als Gruppe wahrnehmbar und von denen anderer Klassen abgrenzbar zu visualisieren, sind sie durch Umrandung, einer fächerartigen Verbindung und Grautonuntergrund optisch zusammengefaßt. Das graphische Konstrukt für diese Zusammenfassung von Werteattributen einer Klasse wird in dieser Arbeit als *Werteknoten* bezeichnet. Diese Ebene enthält mehr Informationen, obwohl die opti-

senknoten und erhöht die Anzahl graphisch modellierbarer Referenzen einer Klasse. Ein Beispiel für erweiterte Rasterung ist im Anhang B.1 gezeigt.

sche Übersichtlichkeit und leichte Erfaßbarkeit vergleichsweise abnimmt.

Beim Vergleich der Sichten wird deutlich, daß sie in Bezug auf die Anforderungen Übersichtlichkeit und leichte Erfaßbarkeit auf der einen Seite und Vollständigkeit von semantischer Schemainformation auf der anderen Seite jeweils verschiedene Gewichtungen haben.

Die Trennung in verschiedene Diagramme verfolgt das Ziel, Diagramme lesbarer zu gestalten. Die Visualisierung verschiedener Kantenarten in einem Diagramm würde die optische Wahrnehmung und Interpretation seitens des Betrachters erschweren [Bal88] [Sta87]. Um dies zu vermeiden, werden Referenzen und Vererbungsbeziehungen jeweils in *verschiedenen Diagrammen* separat dargestellt. Das bei der Einführung der drei Ebenen verwendete Graphikbeispiel stellt ein *Referenzdiagramm* dar. Die Abb. 2.8 zeigt Vererbungsbeziehungen einer Anwendung[19] als *Vererbungsdiagramm*. Diese modelliert die Beziehungen zwischen Personen, Angestellten, Studenten und Hilfskräften einer Universität.

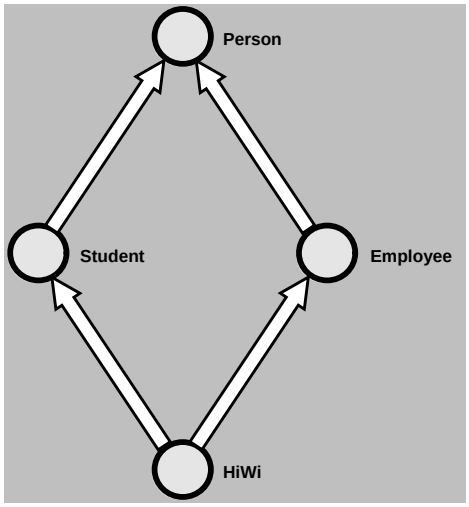


Abbildung 2.8: Visualisierung von ***isA***-Beziehungen eines Universitätsbeispiels

Auch im Vererbungsdiagramm sind Klassenknoten rasterartig angeordnet. Zwischen ihnen liegen Vererbungspfeile, um Subklassenbeziehungen anzuzeigen. Wie bei der Standardsicht des Referenzdiagramms ist die Graphik auf eine Art von Knoten und Kanten beschränkt. Dies gewährleistet eine leichte optische Wahrnehmung der visualisierten Vererbungsbeziehungen zwischen Klassen. Die beiden Diagramme können nebeneineinader angeordnet sein.

Eine integrative Sicht beider Diagrammarten in einer Graphik, um eine *Gesamtsicht* mit allen Schemainformationen zu erhalten, läßt sich durch *Overlaytechniken* realisieren, wie in einem Beispiel in Anhang B.2 gezeigt.

2.3.2.4 Zusammenfassung und Bewertung

In den beiden vorhergehenden Abschnitten wurden Basissymbole und Diagramme definiert, die den in Abschnitt 2.3.2.1 genannten Anforderungen an Visualisierung, bei Beachtung der dort

[19] Aus didaktischen Gründen bezieht sich dieses Beispiel nicht auf die Anwendung TRACY, da diese nur eine Vererbungsbeziehung besitzt.

genannten Richtlinien, in weiten Teilen genügen (vgl. [LM92]). Es stellt sich heraus, daß sich einige Ziele der Anforderungen gegenseitig ausgrenzen. Um Diagramme übersichtlich zu gestalten, wird auf die Darstellung aller schemarelevanter Information in einer Graphik verzichtet.
Es folgt eine Auflistung von Vorteilen und Schwächen der gewählten Diagrammdarstellung. Denkbare Alternativen zur Vermeidung der Schwächen werden am Ende erörtert. Eine übersichtliche Diagrammgestaltung wird durch folgende visuelle Konzepte erreicht.

Verschiedene Visualisierungen für die semantisch unterschiedlichen OM1 Konzepte *Klasse*, *Referenzattribut*, *Werteattribut* und *Vererbungsbeziehung* vermeiden ein *semantisches Überladen* einer Symbolart.

Die Separierung eines OM1 Schemas in zwei getrennte Diagramme, in das *Referenzdiagramm* (in der Standardsicht) und in das *Vererbungsdiagramm*, folgt der Richtlinie, ein Diagramm nur auf eine Knoten- und Kantenart zu begrenzen.

Die Rasteranordnung von Klassenknoten und die topologische Beschränkung der möglichen Verbindungskanten auf benachbarte Klassenknoten im Referenzdiagramm tragen zu einer leichten Erfassung von Klassen und Beziehungen einer Anwendung bei.

Eine Vervollständigung von Information wird durch verschiedene Sichten bewirkt. Schemainformation ist ein- und ausblendbar. Die dadurch alternativ wählbaren Granularitätsstufen der angezeigten Information lassen eine Wahl zwischen übersichtlicher und vollständiger Darstellung von Information zu.

Restriktionen der Symbolanordnung schränken die Freiheitsgrade in der Modellierung ein. Klassensymbole müssen so im Referenzdiagramm angeordnet sein, daß alle Referenzen des OM1 Schemas durch Verbindungen **benachbarter Klassensymbole** darstellbar sind. Die Anzahl der graphisch modellierbaren Referenzen einer (als referenziert oder referenzierend) involvierten Klasse beschränkt sich auf maximal acht. Ferner sind zwischen zwei Klassen maximal zwei definierte Referenzen im Diagramm anzeigbar. Zur Reduzierung dieser Einschränkungen werden zwei Alternativen von graphischer Modellierung vorgestellt.

Eine erweiterte Rasterung mit mehr zugelassenen Rasterpositionen erhöht die mögliche Anzahl topologisch benachbarter Klassenknoten. Besonders sogenannte *kaskadierende Beziehungen*[20] [LS87] sind als Baumstruktur übersichtlich zu visualisieren. Klassen derselben Kaskadenstufe haben keine Beziehungen untereinander und können optisch näher zueinander liegen (siehe dazu Beispieldiagramme in Anhang B.1. Ferner erhöht sich die Zahl der maximal modellierbaren Referenzen einer involvierten Klasse (von acht) auf 24. Falls die Beziehungsanzahl zum Modellieren nicht ausreicht, ist eventuell eine erneute Modellierung der Anwendung oder eine *Modularisierung* (s.u.) anzuwenden. Nachteilig an der erweiterten Rasterung ist eine mögliche Unübersichtlichkeit von Diagrammen durch Kantenüberschneidungen. Die übersichtliche Graphikgestaltung liegt in der Verantwortung des Modellierers.

Eine Modularisierung von Graphiken, die Diagramme nach semantischen Gesichtspunkten in mehrere Teilgraphiken gliedert, wird in [LM92] und in [Tha91b] vorgeschlagen: Eine Übersichtsgraphik zeigt nur die (aus Anwendungssicht) wichtigsten Referenzen zwischen Klassen an. In semantisch zusammenhängenden Teildiagrammen, die als graphische Module zu verstehen sind, werden Beziehungen jeweils detailliert dargestellt (siehe dazu die

[20] Ein Beispiel für kaskadierende Beziehungen sind *ist-Teil-von*-Beziehungen, die z.B. mehrstufig die Beziehungen von Fahrzeugteilen in einem Fahrzeug modellieren.

Beispiele in Anhang B.3 (aus [LM92]). Die Vorteile liegen darin, daß die Visualisierung bei steigender Zahl graphisch modellierbarer Referenzen zwischen Klassen übersichtlich bleibt.

Aus Anwendungssicht kann es wünschenswert sein, eine integrative Sicht aus dem Referenzdiagramm und dem Vererbungsdiagramm in einer Graphik zu erhalten. In [LM92] wird dafür eine *überlagernde Darstellung* beider Diagramme vorgeschlagen. Bei der Modellierung sollte es dem Benutzer überlassen werden, zwischen einer (durch Restriktionen des Systems unterstützten) übersichtlichen, aber in den Freiheitsgraden beschränkten Modellierung und einer mehr Freiheitsgrade zulassenden Modellierung, bei der der Benutzer für die Übersichtlichkeit Verantwortung trägt, zu wählen.

Kapitel 3

Design des Graphikeditors

In diesem Kapitel wird der Entwurf der Benutzerschnittstelle für den Graphikeditor beschrieben. Der Entwurf umfaßt die optische Gestaltung der Editorkomponenten sowie die Dialogführung zwischen Benutzer und Editor. Als Standard für den Schnittstellenentwurf wird OPEN LOOK[1] verwendet, welches das Aussehen und die Funktionalität (*look and feel*) von graphischen Benutzeroberflächen durch Stilrichtlinien genormt festlegt. Die in Abschnitt 2.3.2 spezifizierten Diagrammformen stellen optische Richtlinien zur übersichtlichen Diagrammgestaltung dar, die beim Entwurf der diagrammanipulierenden Operationen zu berücksichtigen sind.

In [Obe92] werden unterschiedliche Dialogarten für verschiedene Möglichkeiten und Alternativen von Dialagsteuerung in interaktiven Softwarewerkzeugen vorgestellt. Bei der Spezifikation der Dialoge der Schnittstelle werden jeweilige Stärken und Schwächen der Dialogarten berücksichtigt. Für einige Editoroperationen werden Wahlmöglichkeiten zwischen verschiedenen Dialogarten entworfen.

Ein nächster Schritt zur Überprüfung des Entwurfs bezüglich Benutzerfreundlichkeit ist dessen Evaluierung. Evaluationsverfahren haben u.a. zum Ziel, Hilfen bei Entwurfsentscheidungen zu geben und Qualitätsprüfungen an fertigen Produkten vorzunehmen [Bal88] [Obe92]. Die Evaluierung des Entwurfs ist eine komplexe Aufgabe und liegt außerhalb des Rahmens dieser Arbeit. Zur Evaluierung kann z.B. ein heuristisches Verfahren verwendet werden, bei dem Benutzer bei der Arbeit am Prototypen des Graphikeditors beobachtet werden. Die aus der Beobachtung resultierenden Ergebnisse können anhand verschiedener Kriterien der Benutzerfreundlichkeit (Aufgabenangemessenheit, Übersichtlichkeit, Erlernbarkeit, Steuerbarkeit etc.) ausgewertet werden [Obe92] [Rau89].

Dieses Kapitel gliedert sich in drei Abschnitte. Im ersten Abschnitt werden die Komponenten der OPEN LOOK Benutzeroberfläche vorgestellt. Thema des zweiten Abschnitts ist die Einführung verschiedener Dialogarten zur interaktiven Dialogsteuerung. Der dritte Abschnitt stellt die Benutzerschnittstelle des Graphikeditors vor. Die Gestaltung einzelner Editorkomponenten, ihre Funktionalität und die Art der Dialogführung werden ausführlich beschrieben.

3.1 Die OPEN LOOK Benutzeroberfläche

Um das Design des Graphikeditors definieren zu können, ist zunächst eine geeignete Benutzeroberfläche zu wählen. Aus folgenden Gründen fiel die Wahl auf die graphische Schnittstelle

[1] *OPEN LOOK* ist ein eingetragenes Warenzeichen der Firma *AT&T*.

OPEN LOOK UI[2].

- OPEN LOOK ist fensterorientiert und spezifiziert eine Oberfläche, die die *Schreibtischmetapher* (*desktop*) unterstützt. Damit ist sie auch für ungeübte Benutzer *leicht erlernbar* [Rau89].

- Eine Vereinheitlichung von OPEN LOOK Schnittstellen wird durch eine Standardisierung ihrer Komponenten unterstützt. Stilrichtlinien (*style guides*) legen die Gestaltung (*look*) und die Bedienung der Schnittstellenkomponenten (*feel*) über Eingabegeräte wie Maus und Tastatur fest (siehe [SM90b] und [SM90c]).

- OPEN LOOK ist auf der Hardwareplattform für die Implementation des Graphikeditors verfügbar. Die Harwareplattform besteht aus *Sparc*-Arbeitsplatzrechner unter *Sun/Unix*. Die darauf zur Verfügung stehende graphische Benutzerschnittstelle ist *OpenWindows 3.0*, eine Implementation des *OPEN LOOK UI* Standards.

- Die Implementation von OPEN LOOK Komponenten und anwendungsspezifischer Komponenten wird durch einen *Werkzeugkasten* (*toolkit*) unterstützt. Mit seiner Hilfe lassen sich alle Bildschirmkomponenten des Graphikeditors implementieren (vgl. Kapitel 4.1.3).

Relevante Komponenten und deren Stilrichtlinien werden in den folgenden Abschnitten eingeführt. Dabei wird jeweils das Aussehen und der Aufbau, die möglichen Ein- und Ausgaben sowie der vorwiegende Anwendungszweck einer Komponente erläutert. Zuvor sei zunächst auf die Spezifikation des Eingabegeräts *Maus* und deren Tasten in *OPEN LOOK* hingewiesen. Eine Maus ist ein zeigendes Eingabegerät, dem ein Mauszeiger[3] (*mouse cursor*) auf dem Bildschirm zugeordnet ist. Sie besitzt drei Knöpfe, denen verschiedene Funktionen zugeordnet sind. Der linke dient zum Auswählen (*select*) oder Drücken (*press*) von Bildschirmobjekten und Kontrollelementen (vgl. Abschnitt 3.1.2). Der mittlere Knopf dient zum Erweitern oder Reduzieren (im englischen als *Adjust* zusammengefaßt) von ausgewählten Objekten. Der rechte Knopf erfüllt die *Menü*-Funktion (*menu*): Zu dem unter dem Mauszeiger auf dem Bildschirm befindlichen Objekt öffnet sich ein dazu assoziiertes Menü.

3.1.1 Basisfenster

Ein Basisfenster dient meistens als Hauptinteraktionskomponente für eine Applikation. Beim Start der Applikation erscheint es auf dem Bildschirm, bei deren Beendigung verschwindet es. Es enthält folgende Komponenten (vgl. Abb. 3.1):

- *Kopfzeile* (*header*):
  Diese enthält einen *Fentstermenüknopf*, das ist ein Knopf, über den ein Menü zu öffnen ist. Knöpfe und Menüs werden im folgenden Abschnitt beschrieben. Außerdem sind der *Fenstertitel* mit dem Titel der Applikation sowie globale Mitteilungen in der Kopfzeile enthalten. Globale Mitteilungen geben Auskunft über den Zustand der Applikation oder die im Fenster dargestellte Sicht auf Informationen der Applikation.

- *Kontrollfeld* (*control* area):
  Hier sind interaktiv zu betätigende *Kontrollelemente* (*controls*) angeordnet (siehe den folgenden Abschnitt).

[2] *OPEN LOOK User Interface*

[3] Der Mauszeiger hat in der Regel eine Pfeilform, kann aber – applikationsgesteuert – andere Formen (z.B. die eines Stoppuhr-Symbols) annehmen.

- *Applikationsfläche* (*application pane*):
  In ihr befindet sich anwendungsspezifische Information, die angezeigt wird und die – anwendungsabhängig – eventuell manipulierbar ist.

- *Fußzeile* (*footer*):
  Sie dient zur Anzeige von Statusinformationen und Fehlermeldungen, die anwendungsspezifisch sind.

- *Rollbalken*(*scrollbar*):
  Ein horizontaler oder vertikaler Schiebebalken erlaubt es, durch Verschieben des Inhalts der Applikationsfläche bestimmte Teile der anwendungsspezifischen Information sichtbar zu machen (s.u.).

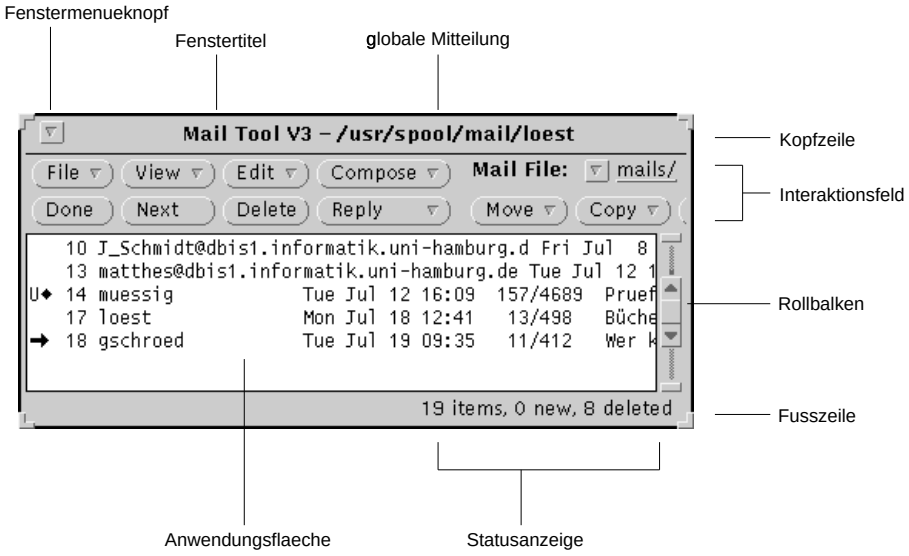


Abbildung 3.1: Komponenten eines Basisfensters in OPEN LOOK

Ein Basisfenster besitzt folgende Standardfunktionalität, die in Menüpunkten des Fenstermenüknopfs interaktiv gewählt werden kann. Es ist schließbar und wird dann nach dem Schließen als *Piktogramm* (*icon*) angezeigt. Es ist *entfernbar* vom Bildschirm entfernbar. Es ist in seiner Größe änderbar, verschiebbar und kann in den Vordergrund oder Hintergrund des Bildschirms gelegt werden. Die Elemente des Kontrollfelds werden im folgenden Abschnitt beschrieben.

3.1.2 Kontrollelemente

Im Kontrollfeld eines Basisfensters oder auch in Kommando- und Eigenschaftsfenstern (vgl. Abschnitt 3.1.3) befinden sich verschiedene *Kontrollelemente* (*controls*), die interaktiv zu betätigen sind und die Aufgaben der Anwendung steuern. In Abb. 3.2 (aus [Mü94b]) sind verschiedene Arten von Kontrollelementen[4] aufgeführt. Sie werden im folgenden erklärt.

[4] mit englischer Benennung

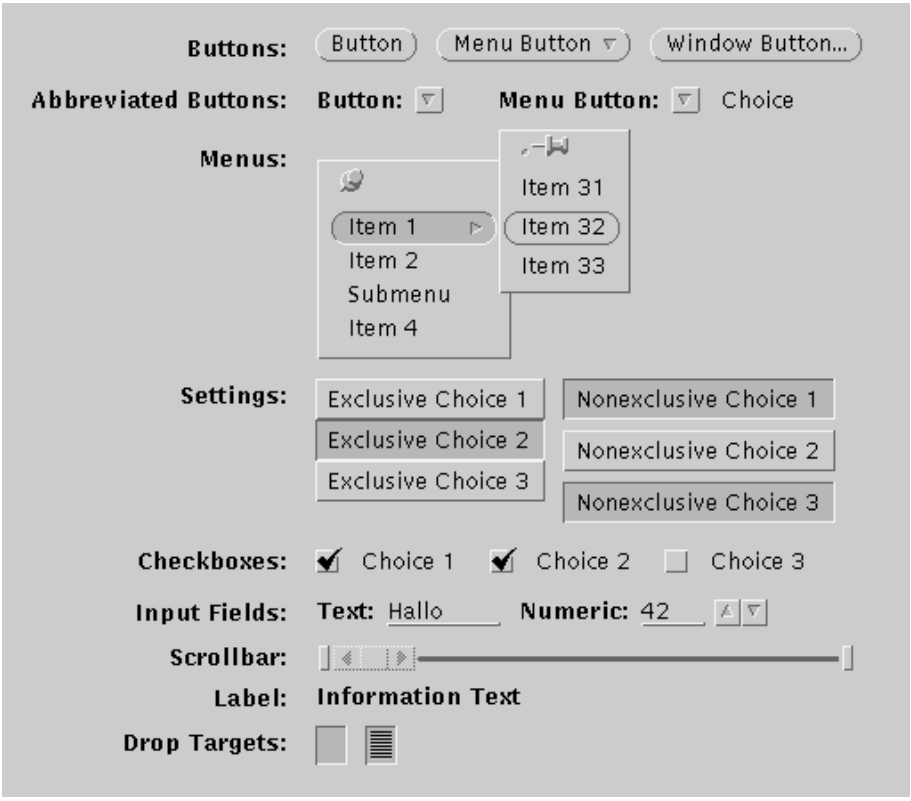


Abbildung 3.2: OPEN LOOK Komponenten

Knöpfe (*buttons*)[5] können über die *Select*-Maustaste gedrückt werden und lösen ein Kommando aus. Jeder Knopf hat eine Beschriftung (*label*), die das Kommando angibt, das jeweils durch Drücken des Knopfes ausführbar ist. Ein Menüknopf (*menu button*) enthält ein Dreiecksymbol; er kann über die *Menü*-Maustaste gedrückt werden, woraufhin sich ein (eventuell geschachteltes) Menü mit *Einträgen* (*items*) öffnet. Menüeinträge haben dieselben Eigenschaften wie Knöpfe.

Menüs (*menus*) sind in zwei Arten aufgeteilt. Menüknöpfe[6] und *Aufklappmenüs*[7] (*popup menus*).

Auswahlen (*settings*) sind Rechtecke mit Texten oder Symbolen als Inhalt, die in Gruppen vertikal oder horizontal angeordnet sind. Es gibt zwei Arten von Auswahlen: *exklusive Auswahlen* (*exclusive settings*) erlauben die alternative Selektion nur einer *Auswahl* aus einer Gruppe, *nicht exklusive Auswahlen* (*nonexclusive settings*) erlauben die Selektion mehrerer *Auswahlen*. Selektierte Auswahlen sind optisch hervorgehoben.

Rollbalken (*scrollbars*) dienen zum Verschieben eines Fensterinhalts und zur Anzeige der Proportion des gerade im Fenster sichtbaren Inhalts. Beim Drücken oder Ziehen des *Aufzugs*[8] (*elevator*) wird der Fensterinhalt um bestimmte Einheiten horizontal bzw. vertikal (je nach Rollbalkenart) verschoben.

[5] Knöpfe werden in ihrer allgemeinen Form auch als *Kommandokn'opfe* (*command buttons*) bezeichnet.
[6] siehe oben; – auf *abkürzende* Menüknöpfe (*abbreviated menu buttons*) wird hier nicht näher eingegangen.
[7] Diese sind jeweils Flächen oder Symbolen zugeordnet und werden über die *Menü*-Maustaste aufgeklappt.
[8] Das ist ein Teil eines Rollbalkens.

Numerische und Textfelder (*numeric fields/ text fields*) sind entweder nur lesbare oder editierbare Anzeigefelder. Editierbare Felder – auch *Eingabefelder* (*input fields*) genannt – können über Tatastureingabe mit Zeichen gefüllt werden. *Rollknöpfe* (*scroll buttons*) dienen dazu, nicht vollständig anzeigbare Texte über ein Textfeld zu schieben oder numerische Werte im numerischen Feld zu inkrementieren oder dekrementieren.

Ja/Nein-Felder (*check boxes*) sind zweiwertig und repräsentieren ein *Ja* oder *Wahr* bzw. ein *Nein* oder *Falsch*. Ein *Ja/Wahr* wird durch ein Haken im Feld dargestellt.

3.1.3 Aufklappfenster

Ein *Aufklappfenster*[9] (*popup window*) enthält links oben einen *Pinnknopf* (*pushpin*), der zwei visuelle Zustände einnehmen kann. Ist der Pinnknopf *fixiert* (*pinned*), bleibt das Fenster nach Ausführung der in ihm enthaltenen Kommandos angezeigt. Ist der Pinnknopf *unfixiert* (*unpinned*), verschwindet das Fenster nach der Kommendoausführung. Der Pinnknopf ist vom Benutzer setzbar. Es gibt drei Arten von Popupfenstern.

Kommandofenster (*command window*): In ihnen werden Parameter für einen Befehl gesetzt und der Befehl selbst durch Betätigung eines Knopfes ausgeführt (siehe Abb. 3.3).

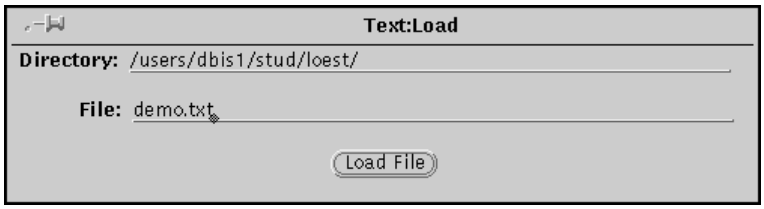


Abbildung 3.3: Ein Aufklapp-Kommandofenster mit unfixiertem Pinnknopf

Eigenschaftsfenster (*property window*): Ihr Inhalt ist formularartig gegliedert. Es können Eigenschaften von vorher selektierten Objekten durch Parameter verändert und durch einen *Ausführungsknopf* (*apply button*) auf die Objekte angewendet werden (siehe Abb. 3.4).

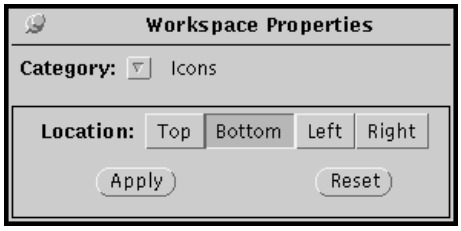


Abbildung 3.4: Ein Aufklapp-Eigenschaftsfenster mit fixiertem Pinnknopf

Hilfefenster (*help window*) und Notizfenster (*notice window*): Fenster dieser beiden Arten zeigen jeweils Text an und verschwinden bei Betätigung eines Knopfes. Wenn ein

[9] im folgenden auch als *Popupfenster* bezeichnet

einziger Knopf vorhanden ist, dient er als Bestätigungsknopf, bei mehreren vorhanden Knöpfen ist mit jedem ein bestimmtes Kommando verbunden. Solange das Fenster geöffnet ist, sind alle weiteren Interaktionsmöglichkeiten auf dem Bildschirm blockiert.

Ein Popupfenster kann nur geöffnet werden, wenn das ihm zugeordnete *Elternfenster* geöffnet ist.

3.2 Dialogarten

In diesem Abschnitt weren verschiedene Dialogarten für interaktive Dialoge zwischen Benutzer und Rechner vorgestellt. Es werden jeweils die Vor- und Nachteile einer Dialogart bezüglich ihrer Verwendung bei unterschiedlichen Benutzergruppen und Anwendungsaufgaben erörtert.. Die daraus gezogenen Erkenntnisse werden bei der Spezifikation von Dialogen, die bei der Benutzung des Graphikeditors vorkommen (siehe Abschnitt 3.3), berücksichtigt.

In den folgenden beiden Abschnitten werden zunächst zwei Dialogarten vorgestellt, die bei der interaktiven Benutzung von graphischen Schnittstellen als verschiedene Formen der Dialogführung zwischen Benutzer und System eingestuft werden können. Sie werden als *benutzergesteuerte* und *systemgesteuerte Dialoge* benannt [Rau89] [Obe92]. Ihre Charakteristika, ihre Erlernbarkeit im Zusammenhang mit verschiedenen Benutzergruppen und ihre Schnelligkeit werden in den beiden folgenden Abschnitten benannt. Am Ende werden Kriterien genannt, die für die Wahl der Dialogarten ausschlaggebend sind.

3.2.1 Benutzergesteuerte Dialoge

Diese werden nach zwei Arten von Eingabetechniken klassifiziert. Es gibt *Kommandodialoge* und *direkte Manipulation*. In Kommandodialogen werden *Kommandos* direkt in ein Befehlsfenster eingegeben. Vorteilhaft ist die hohe Flexibilität und Steuerbarkeit durch den Benutzer. Von Nachteil ist die längere Einarbeitungszeit sowie die hohen Fehlerraten bei der Eingabe. Die direkte Manipulation setzt folgendes voraus. Die zu manipulierenden Objekte und darauf auszuführende Operationen (Beispiel: *Löschoperation* als Papierkorbsymbol) werden fortwährend auf dem Bildschirm dargestellt. Operationen sind schnell, inkrementell und umkehrbar ausführbar. Die Ergebnisse sind unmittelbar sichtbar. Die Eingabe erfolgt ausschließlich über Zeigen (mit dem *Cursor*) und Auswählen uber Maustasten oder die Tastatur [Shn92]. Vorteile dieser Dialogform sind:

- Leichte Erlernbarkeit;
- niedrige Fehlerraten;
- relativ schnelle Durchführbarkeit;
- Steigerung des Kompetenzgefühls und der Akzeptanz seitens des Benutzers.

Nachteilig ist sind folgende Eigenschaften:

- Einige Operation sind schwierig und kompliziert durchführbar;
- die Benutzer dürfen keine visuellen oder motorischen Schwächen haben;
- verwendete Metaphern prägen den Benutzer hinsichtlich *einer* Sicht.

Zum Teil sind auch Menüeingaben in fensterorientierten Systemen mit Anzeige von mehreren Menüs in einem oder mehreren Fenstern als benutzergesteuert einzuordnen [Obe92].

3.2.2 Systemgesteuerte Dialoge

Es werden Formulare sowie Menüs, die geschachtelt und hierarchisch aufrufbar sein können, für Dialoge benutzt. Positiv zu bewerten ist die gute Benutzerführung, eine schnelle Einarbeitungszeit und kleine Fehlerraten bei der Benutzereingabe. Bei geeigneter Beschriftung ist die Semantik leicht verständlich. Formulare eignen sich besonders für stark strukturierte Datenerfassung und Setzen von Operationsparametern (z.B. *Druckformat* beim Drucken). Menüs unterstützen und strukturieren Planungs- und Entscheidungsprozesse. Negativ zu bewerten ist, daß diese Dialogarten umfangreicher und ineffizienter als die benutzergesteuerten sind. Sie haben nur eine geringe Flexibilität und lassen dem Benutzer wenig Handlungsspielraum. Menüs erfordern eine Navigation in der Menüstruktur sowie einzelnes Öffnen und Absuchen und sind deshalb weniger für geübte Benutzer geeignet [Obe92].

3.2.3 Wahl der Dialogart

Sieht man von der Kommandoeingabe als Dialogart einmal ab, so ist zu überlegen, welche Dialogarten in einer Anwendung unter *OPEN LOOK* zu bevorzugen sind: die benutzergesteuerte, direkt manipulative oder die systemgesteuerte, formular- und menübasierte.

Es sind auch Mischformen zulässig, die – je nach zu erfüllender Aufgabe – eine Kombination von *verschiedener Dialogarten* in einem Dialog, der aus mehreren Dialogschritten besteht, beinhalten. Wenn beispielsweise vom Benutzer beabsichtigt ist, auf verschiedenen Objekten dieselbe Operation nacheinander auszuführen (z.B. *Löschen von Diagrammobjekten einer Sorte*), ist es sinnvoll, per Menü zunächst die gewünsche Operation *einzustellen* und dann sukzessive die zu bearbeitenden Objekte auszuwählen [Obe92]. Wenn bestimmte Argumente für eine Operation benötigt werden, z.B. Drucken eines Dokuments, ist es sinnvoll, nach Auswahl der Operation ein Formular aufzuklappen, in das die Parameter eingegeben werden können.

3.3 Interaktionskomponenten des Graphikeditors

In diesem Abschnitt werden die OPEN LOOK Komponenten des Graphikeditors beschrieben. Es wird gezeigt, wie die Komponenten auf dem Bildschirm angeordnet werden. Desweiteren wird schrittweise – im Zusammenhang mit den Komponenten – der eigentliche Funktionalitätsumfang des Graphikeditors beschrieben. In diesem Zusammenhang werden Folgen von Interaktionen auf Bedienelementen des Graphikeditors und die jeweils damit verbundene anwendungsspezifische Funktionalität erklärt.

Die Komponentenbeschreibungen sind nach verschiedenen Kategorien von Funktionalität in zwei Abschnitte aufgeteilt. Der erste Abschnitt beschreibt die visuelle Gestaltung und die basisfensterbezogene Funktionalität des Graphiketors. Im zweiten Abschnitt werden Bedienelemente eingeführt, die editorspezifische Operationen realisieren. Die mit den Bedienelementen verbundenen Dialoge zwischen System und Benutzer werden genauer beschrieben.

3.3.1 Editorfenster

Gemäß der in Abschnitt 2.3.2 beschriebenen Aufteilung von OM1 Schemavisualierungen in zwei Diagrammarten besteht der Graphikeditor aus zwei Komponenten, die als Fenster auf dem Bildschirm erscheinen: Ein *Basisfenster* zeigt auf seiner Applikationsfläche das *Referenzdiagramm*

an, ein *Popupfenster*[10] stellt das *Vererbungsdiagramm* dar.

Das Basisfenster mit dem Referenzdiagramm ist die Hauptinteraktionskomponente des Graphikeditors und wird daher im folgenden als *Hauptfenster* bezeichnet. Beim Aufruf des Graphikeditors öffnet sich dieses zuerst. Von ihm aus kann das Aufklappfenster mit dem Vererbungsdiagramm (im folgenden *Vererbungsfenster* genannt) geöffnet werden. Durch Schließen des Hauptfensters wird der Programmlauf des Graphikeditors beendet. Beiden Fenstern gemeinsam ist folgende Funktionalität.

- OPEN LOOK *Fensterfunktionalität*:
  Die Fenster sind verschiebbar, schließbar und änderbar in Größe, Position und Überlappungsreihenfolge.

- Anzeige und graphisches Editieren von Diagrammen:
  Es sind verschiedene Ausschnitte des jeweiligen Diagramms – durch horizontales oder vertikales Verschieben (*scrolling*) von Rollbalken – anzeigbar. Das Editieren bezieht sich auf Einfügen und Löschen jeweiliger Diagrammkomponenten.

- Ausdrucken des jeweiligen Diagramms: Dies geschieht über ein spezielles Kommandofenster, welches in Abschnitt 3.3.2.3 näher beschrieben wird.

Darüber hinaus hat das *Hauptfenster* zwei weitere Kategorien von Funktionalität.

- *Spezifische Funktionalität* zur Handhabung des Referenzdiagramms:
  Es sind drei verschiedene Sichten der Diagrammanzeige[11] sowie zwei verschiedene Rasterarten[12] wählbar (vgl. Abschnitt 2.3.2).

- *Schemaweite Funktionalität*:
  Diese beinhaltet das Laden und Sichern von OM1 Visualisierungen (bezogen auf beide Diagrammarten), die Generierung von OM1 Schemadefinitionen in textueller Form und den damit verbundenen Aufrufen von OM1 Klasseneditoren.

Im folgenden werden exemplarisch die Fensterkomponenten des Hauptfensters beschrieben, die den – bereits in Abschnitt 3.1.1 erörterten – prinzipiellen Fensteraufbau in der Abb. 3.5 beispielhaft illustriert und kurz erläutert. Eine genauere Beschreibung einzelner Komponenten und Dialoge zur Steuerung anwendungsspezifischer Funktionen erfolgt im Anschluß an diesen Abschnitt (vgl. Abschnitt 3.3.2).

[10] Die Realisierung als Popupfenster erfolgt aus technischen Gründen. Es könnte auch ein Basisfenster sein.
[11] Die Sichten beinhalten verschiedene Granularität angezeigter Information.
[12] Die Rasterarten sind *Standardraster* und *erweitertes Raster*.

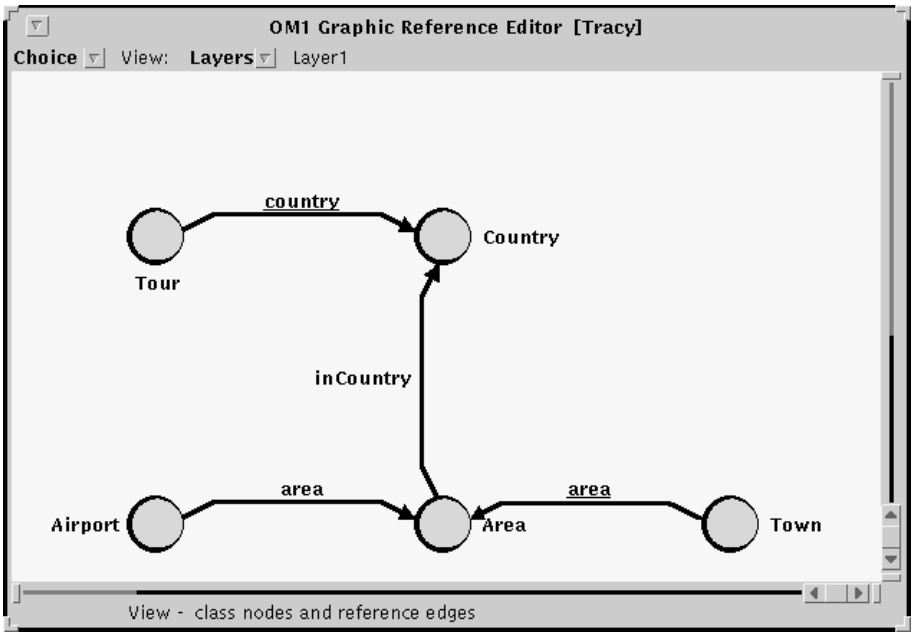


Abbildung 3.5: Das Hauptfenster des Graphikeditors

Der Fenstertitel benennt das gerade gewählte Werkzeug und das geladene Schema. Es wird unterschieden zwischen editierendem Werkzeug (*editor*) und anzeigendem Werkzeug (*viewer*) (vgl. Abschnitt 3.3.2.2). Im Beispiel der Abb. 3.5 heißt der Titel *Graphic Reference Viewer - Tracy*.

Das Kontrollfeld enthält Menüknöpfe mit werkzeugspezifischen Menüs, die später erläutert werden.

Die Anwendungsfläche enthält das *Referenzdiagramm* als Visualisierung eines OM1 Schemas. Es wird vollständig oder als Ausschnitt – je nach Größe – angezeigt.

Die Fußzeile kann drei Arten von Meldungen enthalten:

- Statusmeldungen über angezeigte Diagramminhalte in der Applikationsfläche:
  Im Beispiel der Abbildung 3.5 wird gemeldet, daß *Klassenknoten* und *Referenzkanten* angezeigt werden.
- Statusmeldungen über gerade laufende Systemaktivitäten:
  Dies sind zum einen Meldungen über Ladevorgänge beim Diagrammaufbau nach dem Starten des Editors. Weiterhin werden länger dauernde Vorgänge wie Schemasicherung, Generierungsaktivitäten und Erzeugung von Druckdateien durch Meldungen kommentiert.
- Aufforderungen für bestimmte Benutzereingaben:
  Diese werden in bestimmten Dialogzuständen des Editors angezeigt. Beispiele folgen in späteren Abbildungen (vgl. Abschnitt 3.3.2.4).

Zwei Rollbalken dienen zum horizontalen und vertikalen Verschieben des angezeigten Diagramms auf der Anwendungsfläche des Fensters.

3.3.2 Komponenten für editorspezifische Operationen

In diesem Abschnitt wird zunächst die Funktionalität von Graphikeditoren umrissen. Dann werden Dialoge in Zusammenhang mit den benutzten Interaktionskomponenten beschrieben, die zwischen Benutzer und System stattfinden und sich aus einzelnen Dialogschritten zusammensetzen. Ein Dialogschritt besteht jeweils aus einer Benutzereingabe auf einer Interaktionskomponente sowie aus einer Systemreaktion, die ein visuelles Feedback oder den Aufruf einer Operation beinhaltet. In dem Zusammenhang wird auch beschrieben, welche Anwenderfunktionalität die Operationen eines Dialogs bzw. Dialogschritts auslösen.

Die *Funktionalität* von Graphikeditoren bezieht sich einerseits auf Operationen, die den Zugriff, die Darstellung und die Ablage von Graphiken realisieren. Dies sind: *Laden*, *Anzeige*, *Sicherung* und Ausdruck von Diagrammen. Ferner gibt es graphikmanipulierende Operationen. Sie bewirken die *Erzeugung*, *Änderung* und das *Entfernen* von Objekten, aus denen die Graphiken zusammengesetzt sind.

3.3.2.1 Laden und Sichern von Diagrammen

Es gibt zwei Möglichkeiten zum *Laden* von Diagrammen. Zum einen wird vor jedem Start des Graphikeditors im *StyleTop* Editor (vgl. Abschnitt 1.2) ein Schemaname angegeben. Beim Start des Graphikeditors wird das Schema, wenn es nicht leer ist, in das Applikationsfenster geladen. Zum anderen gibt es die Möglichkeit, über den Menüknopf *Scheme* den Menüeintrag *Load Scheme* zu selektieren, woraufhin in einem Aufklappfenster interaktiv der Schemaname eingegeben wird. das System prüft, ob das Schema existiert. Bei Nichtvorhandensein kann in einem Notizfenster das Laden durch Knopfdruck bestätigt oder abgebrochen werden. Das Sichern von Diagrammen erfolgt über den *Scheme*-Menüknopf und Auswahl des Eintrags *Save Scheme*.

Sowohl während des systemseitigen Ladevorgangs als auch für die Dauer des Sicherns erfolgt eine visuelle Darstellung der gerade laufenden Systemaktivität. In der Fußzeile des Hauptfenserts erscheint eine entsprechende Meldung und der Mauszeiger nimmt die *Stoppuhr*-Form an. Das Hauptfenster ist für Eingabe nicht aktiv.

Dialoggestaltung: Das Laden und Sichern bezieht sich immer auf *beide Diagrammarten*, das *Referenzdiagramm* und das *Vererbungsdiagramm*. Nach dem Start des Editors ist zunächst nur das Hauptfenster mit dem *Referenzdiagramm* sichtbar.
Die Menüs und die Dialoge zwischen Benutzer und System entsprechen den Standards von verbreiteten Editoren, die unter OPEN LOOK laufen[13].

3.3.2.2 Diagrammanzeige

Für die Diagrammanzeige des *Referenzdiagramms* ist explizit der *Anzeigemodus* – alternativ zum *Editiermodus* – wählbar. Es ist sinnvoll, zwischen Anzeige- und Editiermodus zu unterscheiden. Im Anzeigemodus nicht benötige Kontrollknöpfe, die zum Editieren notwendig sind, werden im Anzeigemodus ausgeblendet. Das erhöht die Übersicht über Interaktionsmöglichkeiten und vermeidet ungewolltes, versehentliches Editieren.

[13] Als Beispiele sind die Editoren *Textedit* oder *IslandDraw*, die auf der Oberfläche *OpenWindows* arbeiten, zu nennen.

Nachdem der Anzeigemodus über den Menüknopf *Choice* und den Eintrag *View* im Hauptfenster gewählt ist, wird folgendes angezeigt: In der Kopfzeile erscheint der als globaler Text der Eintrag *Graphic Reference Viewer*. Außerdem erscheint im Kontrollfeld der Menüknopf *Layers*, über dessen Menüeintrage die Auswahl der anzuzeigenden Sicht des Referenzdiagramms (zu Sichten: vgl. Abschnitt 2.3.2) erfolgen kann. Die jeweils gewählte Sicht wird in der Applikationsfläche angezeigt, ihre Numerierung steht rechts neben dem Menüknopf. Im Beispiel in Abb. 3.5 ist die Sicht der *Ebene 1*[14] als *Layer 1* gekennzeichnet und die angezeigten Diagrammsymbole in der Fußzeile kommentiert.

Das **Vererbungsdiagramm** wird in einem Popupfenster angezeigt, welches über die Wahl des Menüeintrag *Inheritance Scheme* im Menüknopf *Scheme* des Hauptfensters aufgeklappt wird.

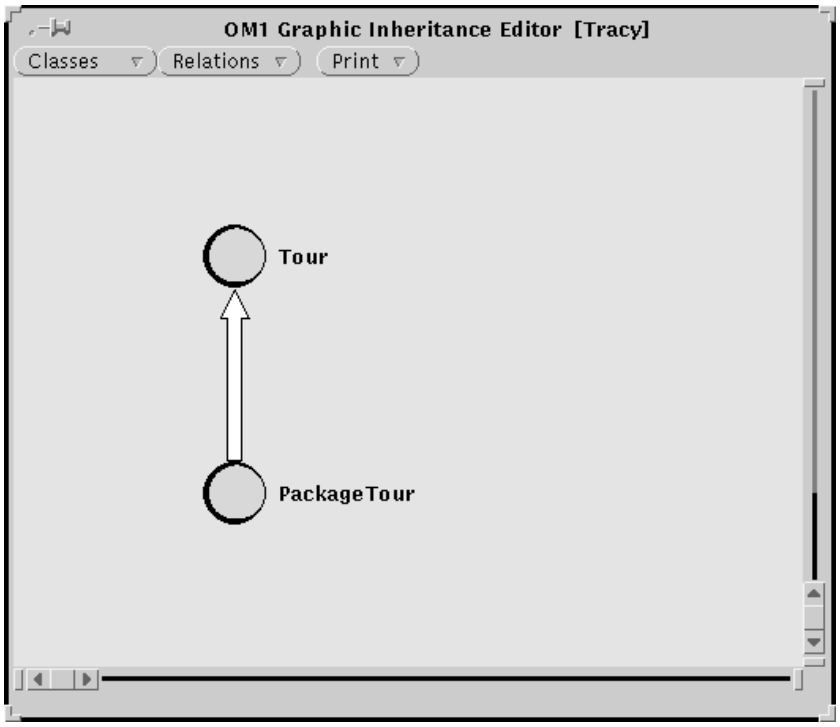


Abbildung 3.6: Das Vererbungsfenster des Graphikeditors

3.3.2.3 Ausdruck von Diagrammen

Sowohl im Hauptfenster als auch im Vererbungsfenster gibt es den Menüknopf *Print*, über dessen Eintrag *Diagram* jeweils ein Kommandofenster aufgeklappt wird, daß editierbare Felder für folgende Parameter enthält (vgl. Abbildung 3.7):

- Größeneinstellung: Es kann zwischen Din A4, Originalgröße und interaktiv einzugebender Größe gewählt werden.
- Formatrichtung: Es stehen Hoch- oder Querformat zur Verfügung.
- Ausgabemedium: Es sind alternativ Datei und Drucker – mit zu spezifizierendem Namen – wählbar.

[14] Referenzdiagramm ohne Anzeige von Werteattributen

Durch Drücken des *Apply*-Knopfes wird die gewählte Druckoption ausgelöst.

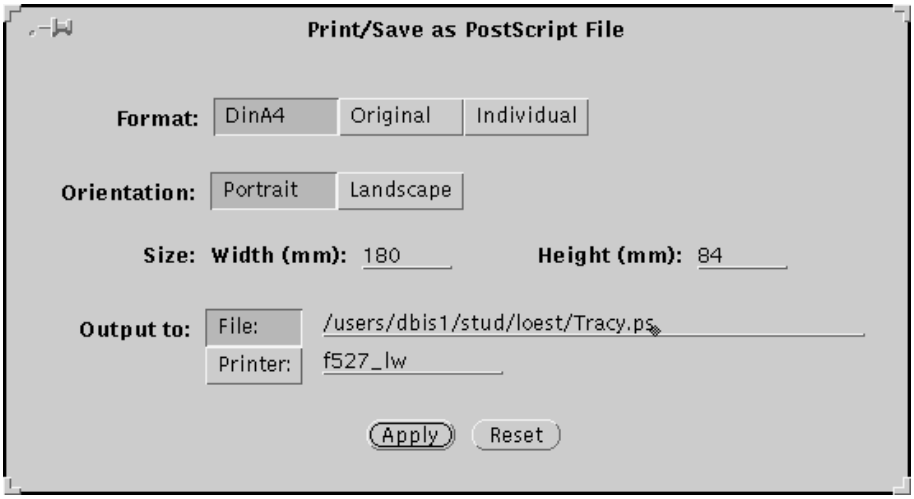


Abbildung 3.7: OPEN LOOK Komponenten

Dialogform im Kommandofenster: Das Layout des Kommandofensters ist nach OPEN LOOK Richtlinien gestaltet [SM90b]. Die einstellbaren Parameter sind eine Teilmenge der in Druckoptionen anderer Softwarewerkzeuge vorhandenen (vgl. z.B. Druckoptionen des kommerziellen Produkts *IslandDraw*).

3.3.2.4 Manipulation des Referenzdiagramms

Für die Diagrammanipulation des Referenzdiagramms ist explizit der *Editiermodus* – alternativ zum *Anzeigemodus* – wählbar. Dies erfolgt über den Menüknopf Choice im Hauptfenster, wo der Eintrag *Edit* gewählt wird. Danach wird folgendes angezeigt. In der Kopfzeile erscheint der als globaler Text der Eintrag *Graphic Reference Editor*. Im Kontrollfeld werden drei Menüknöpfe mit den Beschriftungen (*Classes*, *References* und *Value Attributes*) eingeblendet. Jedes der drei Menüs enthält die Einträge *Insert* und *Delete*. Damit sind sechs graphikmanipulierende Operationen ansteuerbar: Jeweiliges Einfügen und Löschen von Klassenknoten, Referenzkanten und Werteattribute im Referenzdiagramm. Das Menü *Classes* enthält einen weiteren Menüeintrag mit dem Namen *Single Class*. Die damit im Zusammenhang stehende Funktionalität wird in Abschnitt 3.3.2.6 vorgestellt. In den drei folgenden Unterabschnitten werden Dialogarten und einzelne Dialogschritte zum Einfügen von Symbolen in Diagrame beispielhaft aufgeführt.

Einfügen von Diagrammelementen: Die folgenden drei Beispiele stellen anhand von bildhaften Szenarien jeweils Dialoge zum Einfügen einer Symbolart vor. In den Beschreibungen der Dialogschritte werden Benutzeraktivitäten durch das Wort **Benutzer** eingeleitet, Systemaktivitäten durch das Wort **System**. Die Systemaktivitäten beschränken sich beim Einfügen von Symbolen auf Veränderungen der Anzeige, die die Kopfleiste, die Applikationsfläche und die Fußleiste des Hauptfensters betreffen. Anwendungsspezifische Operationen werden nicht ausgelöst.

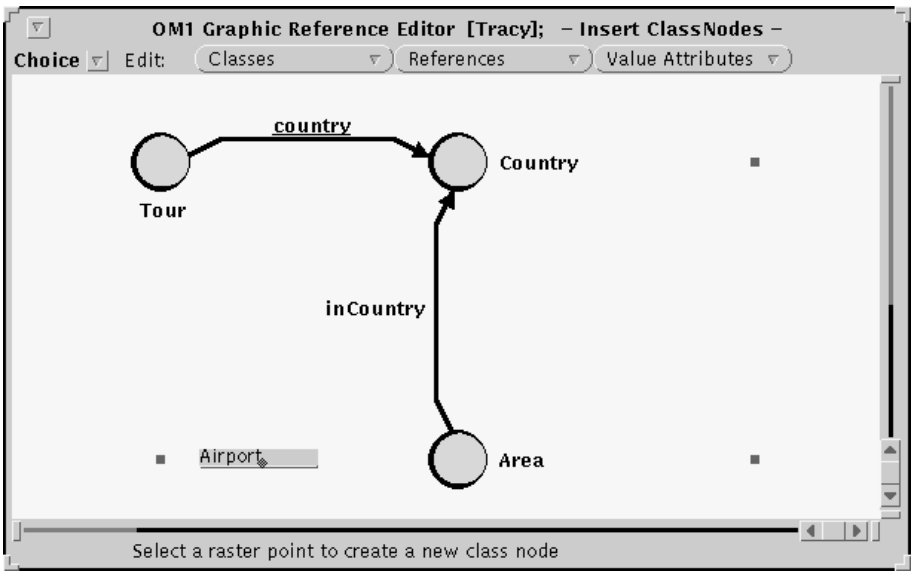


Abbildung 3.8: Einfügen eines Klassenknotens in ein Referenzdiagramm

Abbildung 3.8 beinhaltet die angezeigten Komponenten beim Einfügen von Klassenknoten im Referenzdiagramm. Der Dialog besteht aus den folgenden Dialogschritten.

Benutzer: Auswahl des Menüeintrags *Insert* im Menü *Classes*.

System: Am Ende der Kopfzeile erscheint der Text *Insert class nodes*. In der Applikationsfläche werden Klassenknoten und Referenzkanten sowie Rasterpunkte an freien Positionen, die für das Einfügen verfügbar sind, angezeigt. In der Fußleiste erscheint die Eingabeaufforderung *Select a grid point to insert a new class node*.

Benutzer: Selektieren eines leeren Rasterpunktes.

System: Anzeige eines editierbaren Textfeldes in Rasterpunktnähe mit Textcursor.

Benutzer: Eintragen eines Namens über die Tastatur. Dieser Dialogschritt ist in Abb. 3.8 dargestellt.

System: Anzeige des neu erzeugten Klassenknotens mit dem vorher eingegebenen Namen als Beschriftung.

Dialoggestaltung: Die Einfügeoperation wird zunächst über das Menü gesteuert. Damit erfolgt eine *Feineinstellung* des Editors auf die Handhabung *Einfügewerkzeug für Klassenknoten*. Das eigentliche Einfügen erfolgt durch *direkte Manipulation* der Diagrammfläche seitens des Benutzers. Diese Vorgehensweise ist eine sinnvolle Aufteilung zwischen *menügesteuerter Interaktion* und *direker Manipulation* [Obe92]. Fehlerhafte Benutzereingaben werden abgefangen. Neue Klassenknoten sind nur auf nicht besetzten Rasterpositionen im Diagramm erzeugbar.

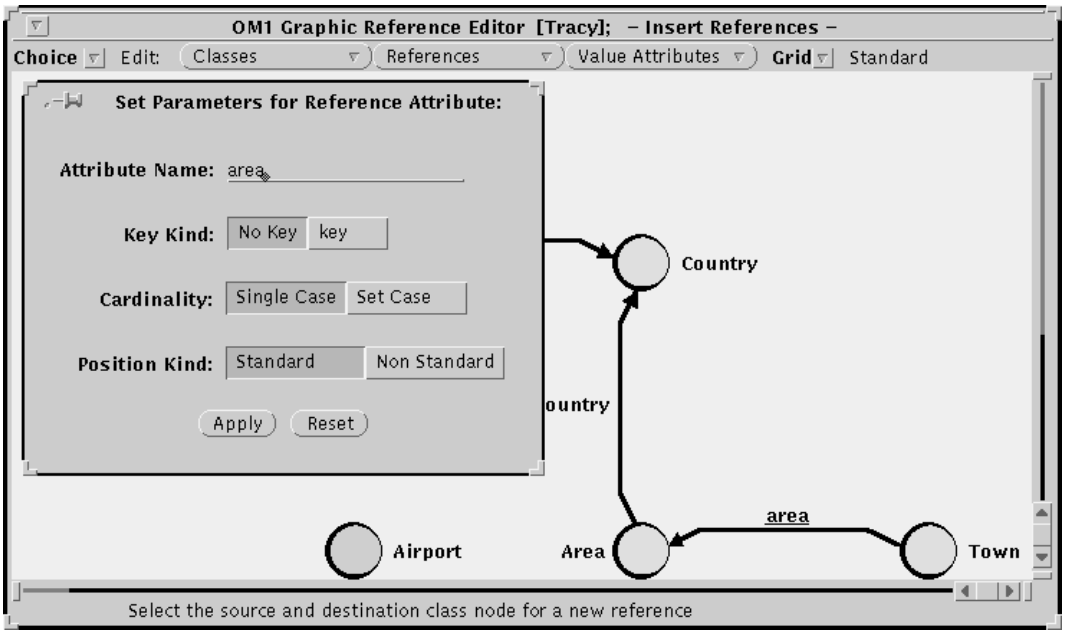


Abbildung 3.9: Einfügen einer Referenzkante ins Referenzdiagramm

Die Abbildung 3.9 stellt die die Bildschirmanzeige während des Einfügens von Referenzkanten im Referenzdiagramm dar. Die ersten Dialogschritte zur Auswahl der Feineinstellung des Werkzeugs vollziehen sich analog zu denen im vorigen Dialogbeispiel. Es schließen sich die folgenden Schritte an.

Benutzer: Selektieren zweier benachbarter Klassenknoten im Diagramm;

System: Jeder der selektierten Klassenknoten wird durch farbliche Hervorhebung angezeigt. Ein Eigenschaftsfenster wird aufgeklappt. Darin sind Eingabefelder und Auswahlen formularartig angeordnet, durch die der Name, die Kardinalität, die Schlüsseleigenschaft und die Ausrichtung der Referenzkante[15] bestimmbar sind.

Benutzer: Setzen der Referenzkanteneigenschaften im Eigenschaftsfenster und Drücken des *Apply*-Knopfes – siehe dazu Abb. 3.9.

System: Anzeige der neu im Diagramm (zwischen den beiden Klassenknoten) erzeugten Referenzkante mit den entsprechenden Eigenschaften.

[15] Es kann alternativ zwischen zwei – zueinander spiegelverkehrten – Knickrichtungen der Referenzkante gewählt werden.

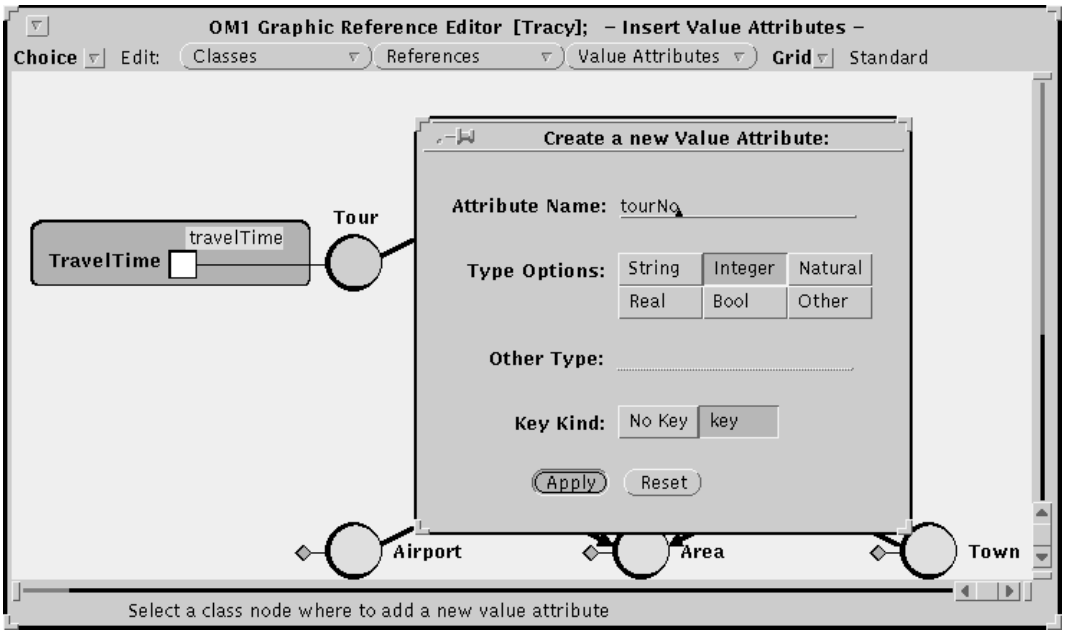


Abbildung 3.10: Einfügen eines Werteattributs in ein Referenzdiagramm

Die Abbildung 3.10 zeigt die die Bildschirmanzeige während des Einfügens von Werteattributen im Referenzdiagramm. Der Dialog zwischen Benutzer und System vollzieht sich analog zu dem oben beim Einfügen von Referenzkanten genannten. Analog zum Einfügen von Referenzkanten werden in einem Eigenschaftsfenster der Name, die Kardinalität, der Wertetyp[16] und die Schlüsseleigenschaft des Werteattributs spezifiziert.

Dialoggestaltung bei Einfügeoperationen: Fehlerhafte Eingaben, die die Rasterung des Schemas verletzen würden, werden abgefangen. Beispielsweise wird zwischen Klassenknoten, die nicht benachbar sind oder zwischen denen keine Referenzkante mehr paßt, keine neue Referenzkante eingefügt. Für verschiedene Teilaufgaben beim Einfügen stehen verschiedene, für die jeweilige Tätigkeit *adäquate Dialogarten* zur Verfügung.

- Für die Feineinstellung des Editors als *Einfügewerkzeug* wird der entsprechende Menüknopf im Kontrollfeld des Hauptfensters benutzt.
- Für die *Positionsangabe* der des einzufügenden Symbols steht die Diagrammfläche mit per Maus selektierbaren Klassenknotens zur Verfügung.
- Für die Festlegung von Eigenschaften von Referenzkanten und Werteattributen werden Eigenschaftsfenster benutzt, welches in ihrem Layout den OPEN LOOK Richtlinien entsprechen.

Die verschiedenen Tätigkeiten beim Einfügen von Symbolen sind in ihrer Reihenfolge signifikant und durch unterschiedliche Interaktionsarten geprägt. Daher ist mit einer gewissen Einarbeitunszeit bis zur Beherrschung der Einfügeoperationen zu rechnen.

[16] Ein Wertetyp kann durch als *Auswahl* von vordefinierten Typbezeichnern oder durch Editieren in einem Texteiingabefeld spezifiziert werden.

Löschen von Diagrammelementen: Um Klassenknoten, Referenzkanten und Werteattribute aus dem Referenzdiagramm zu entfernen, wird zunächst der jeweilige Menüeintrag gewählt, für Klassenknoten beispielsweise der Eintrag *Delete* des Menüknopfs *Classes*. Damit wird die zu löschende Symbolart bestimmt. Das System gibt daraufhin in der Kopf- und Fußzeile entsprechende Meldungen aus. Anschließend erfolgt durch Anklicken von Symbolen der gewählten Art das Entfernen. Das angeklickte Symbol verschwindet vom Bildschirm.

Dialoggestaltung: Die vor dem eigentlichen Löschen über ein Menü zu treffende Auswahl der Symbolart hat folgende Vorteile.

- Die Symbole der gewählten Art sind nicht durch andere Symbolarten verdeckt. Werteknoten, die andere Symbolarten verdecken können, werden nur angezeigt, wenn sie vorher zum Löschen ausgewählt wurden.
- Fehler des Benutzers beim Anklicken im Diagramm werden reduziert. Es sind nur die Symbole entfernbar, die zu der gewählten Art gehören.
- Das Menü muß nur einmal gewählt werden, bevor mehrere Symbole derselben Art durch Anklicken im Diagramm entfernt werden.

Ändern von Diagrammelementen: Wünschenswert sind Operationen für die Änderung von schemarelevanten Eigenschaften wie Klassennamen, Attributnamen und Typnamen in den korrespondierenden Visualisierungen[17] sowie welche für Layoutänderungen durch Verschieben von Symbolen im Diagramm [18] Attributeigenschaften wie Schlüsseleigenschaft, Kardinalität.

3.3.2.5 Manipulation des Vererbungsdiagramms

Im Vererbungsfenster gibt es – im Unterschied zum Hauptfenster – nur einen Bearbeitungsmodus: den des Editierens. Ein Anzeigemodus entfällt hier, weil es keine verschiedenen Sichten auf das Vererbungsdiagramm gibt.

In der Kopfzeile erscheint als globaler Text der Eintrag *Graphic Inheritance Editor*. Im Kontrollfeld werden zwei Menüknöpfe mit den Beschriftungen *Classes* und *Relations* angezeigt. Die Menüs enthalten dieselben Einträge wie die entsprechenden im Hauptfenster: jeweils *Insert* und *Delete*. Dadurch werden folgende Operationen eingeleitet: Einfügen und Löschen von Klassenknoten sowie von Vererbungspfeilen im Vererbungsdiagramm.

Auf die Beschreibung von Dialogen und Dialogschritten zur Steuerung dieser Operationen wird hier verzichtet, weil diese denen im Referenzdiagramm sehr ähneln. Das Einfügen von Klassenknoten erfolgt über dieselben Interaktionskomponenten wie im Referenzdiagramm. Nur die Entfernungen zwischen Rasterpositionen sind in beiden Diagrammen unterschiedlich.

Die Auswahl von Ursprungs- und Zielklassenknoten für einen einzufügenden Vererbungspfeil geschieht durch deren Selektion durch den Benutzer im Diagramm – wie beim Einfügen von Referenzkanten im Referenzdiagramm. Das Entfernen der beiden Symbolarten geschieht analog zum Entfernen von Symbolarten im Referenzdiagramm.

[17] das sind Klassenknoten, Referenzkanten und Werteattribute

[18] eine – unter Einhaltung des Rasterlayouts graphisch und algorithmisch anspruchsvolle – Operation ist z.B. das Verschieben von Klassenknoten unter Mitführung von Werteknoten und Referenzkanten

Dialoggestaltung: Die, im Vergleich zum Hauptfenster, ähnlichen bis identischen Operationen zur Diagrammanipulation vereinfachen den Lernaufwand und erfordern beim Wechsel zwischen Hauptfenster und Vererbungsfenster weniger Umdenken seitens des Benutzers.

3.3.2.6 Direkte Manipulation

Um dem Paradigma der *Direkten Manipulation* [Obe92] [Shn92] von Graphiksymbolen entgegen zu kommen, sind Klassenknoten im Referenzdiagramm *Aufklappmenüs* zugeordnet. Zu jedem Klassenknoten können über das mit ihm assoziierte Menü ihn betreffende Operationen gewählt werden (siehe Abb. 3.11). Dies sind:

Ändern der Lage der Beschriftung: Mit der Maus kann zwischen vier Punkten gewählt werden, wohin der Name des Klassennamens versetzt werden soll: rechts, links, oberhalb oder unterhalb des Klassenknotens.

Löschen des Klassenknotens: Der Klassenknoten verschwindet vom Bildschirm.

Ein- und Ausblenden von Werteattributen: Dies betrifft die Anzeige des Werteknotens, der mit dem Klassenknoten assoziiert ist.

Öffnen eines Klasseneditors für die textuelle Modellierung: Zum Klassenknoten kann ein Klasseneditor geöffnet werden, um die korrespondierende textuelle OM1 Klassendefinition sichtbar und editierbar zu machen. Die damit verbunde Funktionalität wird in Abschnitt 5.1 ausführlicher erläutert.

Generierung von textueller Beschreibung in OM1: Es kann aus der graphischen Modellierung einer Klasse eine textuelle Spezifikation in OM1 Kode in interner, abstrakter Repräsentation generiert werden. Dieser Aspekt wird in Abschnitt 5.1 näher beschrieben.

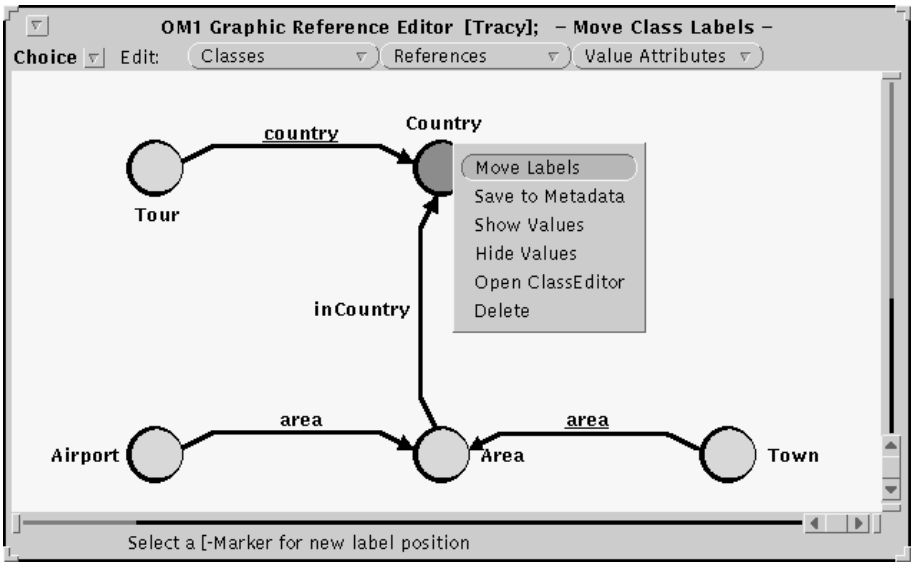


Abbildung 3.11: Darstellung eines Aufklappmenüs für Klassenknoten

Dialoggestaltung: Die den Klassenknoten zugeordneten Aufklappmenüs ermöglichen ein schnelles Aufrufen aufeinander folgender verschiedenartiger Operationen – jeweils nur durch eine Maustastenbetätigung, ohne weitere Dialogschritte. Dieses unterstützt ein objektorientiertes Vorgehen seitens des Benutzers. Erst wird das Objekt gewählt und dann werden die objektspezifischen Funktionen aufrufen.

3.3.2.7 Auswahl verschiedener Rastergrößen

Wie in Abschnitt 2.3.2 spezifiziert, gibt es zwei Rastermodi im Referenzdiagramm: den Standardrastermodus und den *erweiterten Rastermodus*. Der letztere läßt enger liegende Positionen für Klassenknoten sowie weitere Referenzkantenarten zwischen benachbarten Klassenknoten zu. Die Wahl zwischen beiden Rasterarten erfolgt im Hauptfenster über den Menüknopf *Grid*. Dadurch ist es möglich, Klassenknoten enger im Referenzdiagramm zu plazieren und mehr Referenzkanten zwischen ihnen zu erzeugen. Die Vor- und Nachteile des erweiterten Rastermodus bezüglich Diagrammübersichtlichkeit und Freiheitsgraden der Modellierung werden in Kapitel 2.3.2 näher benannt.

Kapitel 4

Realisierung des Graphikeditors

Bei der Realisierung des Graphikeditors sind die Anforderungen bezüglich des zu unterstützenden OM1 Graphikmodells (vgl. Abschnitt 2.3.2) und des Entwurfs der Benutzerschnittstelle (vgl. Kapitel 3) zu berücksichtigen. Ziel dieses Kapitels ist, das schrittweise Vorgehen und angewandte Verfahrensweisen während der Realisierung zu beschreiben.

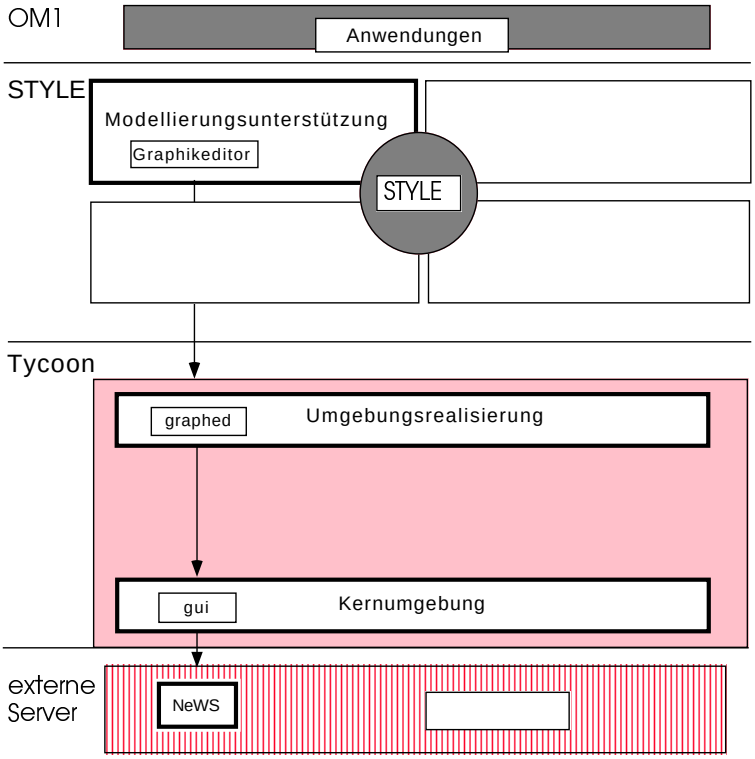


Abbildung 4.1: Architektur der Entwicklungsumgebung für den Graphikeditor

In Abschnitt 4.1 werden die vorhandenen externen Dienste und Werkzeuge des *NeWS* Bildschirmservers vorgestellt. Außerdem wird die Tycoon Bibliothek *newsenv* [Mü94b], die die externen Dienste in das Tycoon System einbindet, beschrieben. Sie bildet die Plattform für die Erstellung von OPEN LOOK Benutzerschnittstellen (*gui*) in Tycoon.

In Abschnitt 4.2 werden Erweiterungen externer Dienste beschrieben, die zur Realisierung von Symbolen und Diagrammen des OM1 Graphikmodells erforderlich sind. In dem Zusammenhang werden Verfahrensweisen zur effizienten Implementierung der Erweiterungen vorgestellt.

Um die Erweiterungen externer Dienste in das Tycoon System einzubinden, ist eine Erweiterung der Bibliothek *newsenv* erforderlich. Die damit verbundenen Aufgaben werden in Abschnitt 4.3 beschrieben.

Die eigentliche Erzeugung der Benutzerschnittstelle, die den Graphikeditor realisiert, wird in Abschnitt 4.4 beschrieben. Den Schwerpunkt des Abschnitts bildet die Schilderung der Verwendung von Modulen der erweiterten Tycoon Bibliothek *newsenv* zur Erzeugung von OPEN LOOK Komponenten, zur Integration von OM1 Graphikkomponenten und zur Implementation von menügesteuerter Funktionalität.

4.1 Verwendete Dienste

Die in den Abschnitten 4.1.1 bis 4.1.3 eingeführten Dienste des NeWS Servers unterstützen die Erstellung graphischer Benutzerschnittstellen für die *OpenWindows*[1] Anwendungsumgebung. Die Seitenbeschreibungssprache *PostScript* [AS90] dient als Basis für das netzwerkfähige Fenstersystem NeWS[2]. Der *Werkzeugkasten* TNT (*The NeWS Toolkit*) definiert unter Verwendung von PostScript und NeWS eine Klassenbibliothek, die die Implementation von Benutzerschnittstellen der OPEN LOOK Spezifikation unterstützt.

In Abschnitt 4.1.4 wird die Tycoon Bibliothek *newsenv* vorgestellt, die die Funktionalität des TNT Werkzeugkastens in Tycoon einbindet, vorgestellt.

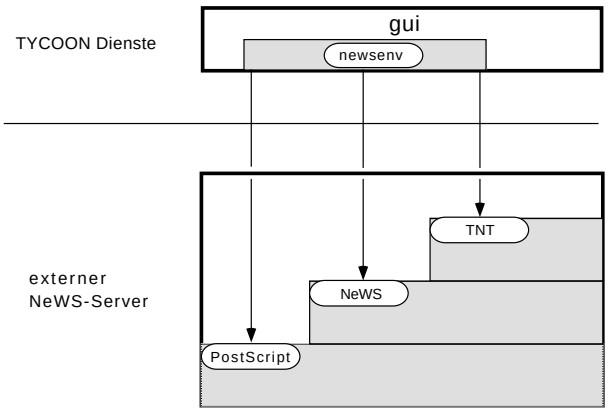


Abbildung 4.2: Einbindung von Diensten des NeWS Servers in Tycoon

4.1.1 PostScript

PostScript ist eine interpretative, kellerspeicherbasierte Programmiersprache, deren Verwendung unter verschiedenen Gesichtspunkten betrachtet werden kann: Als generelle Programmier-

[1] OpenWindows ist eine auf der Hardware des Arbeitsbereichs DBIS installierte graphische Benutzeroberfläche, die dem OPEN LOOK Standard folgt.

[2] Network extensible Window System

sprache, die graphische Primitive enthält; als geräteunabhängige Seitenbeschreibungssprache, wobei die Seiten Texte, geometrische Formen und Bilder enthalten; als interaktives System, welches Seiten auf Druckern oder Bildschirmen ausgibt; und als Schnittstelle zur Formatkonvertierung für Graphikdaten. Der Abschnitt 4.1.1.1 beschreibt Konzepte der Sprache, die für die allgemeine Programmierung wichtig sind. In Abschnitt 4.1.1.2 werden Konzepte zur Graphikerzeugung vorgestellt.

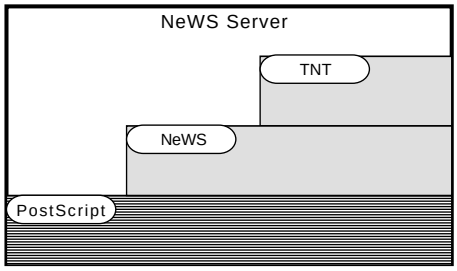


Abbildung 4.3: Dienste des NeWS Servers: PostScript

4.1.1.1 Algorithmische Vollständigkeit

PostScript ist eine allgemeine (*general purpose*) Programmiersprache. Sie umfaßt Datentypen wie ganze und reelle Zahlen, Boole'sche Typen, Zeichenketten und Felder. Zu den Datentypen gibt es Operatoren, die teilweise polymorph anwendbar und dynamisch erweiterbar sind. Es folgt eine Aufzählung weiterer, charakteristischer Datenypen.

Ein Wörterbuch (*dictionary*) besteht aus einer Tabelle, die Schlüssel-Wert Paare enhält. Durch die Tabelle werden Daten und Prozeduren strukturiert und benannt. Jedes Wörterbuch stellt einen geschlossen Sprachraum dar. Ein Beispiel ist das Zeichensatzwörterbuch (*font dictionary*: Es definiert einen Zeichensatz, wobei jeder Buchstabe durch einen Schlüssel und eine dazugehörige Prozedur definiert ist.

Ein Name (*name*) wird als Identifikator für einen Wert oder eine Prozedur benutzt und ist als *Objekt erster Klasse* einzustufen.

Eine Datei (*file*) ist ein flüchtiges oder persistentes Objekt, das Zeichenströme enthält.

Ein Kellerspeicher (*stack*) wird vom *PostScript Interpreter* verwendet, um zu jedem Ausführungszeitpunkt einer Anwendung den Zustand der Ausführung zu definieren. Es gibt drei verschiedene Kellerspeicher.

- Der Operandenkellerspeicher (*operand stack*) dient als Ablage für Ein- und Ausgabeparameter für Operatoren sowie für Ein- und Rückgabeparemeter von Prozeduren (siehe unten).
- Der Wörterbuchkellerspeicher (*dictionary stack*) enthält alle zum Ausführungszeitpunkt geladenen Wörterbüche. Er fungiert als Suchraum, z.B. für Prozeduraufrufe, d.h. ausführbare Namen, die als Schlüssel-Wert Paare gespeichert sind.
- Auf dem Ausführungskellerspeicher (*execution stack*) liegen alle aufgerufenen, ausführbaren Objekte, hauptsächlich Prozeduren und Dateien.

Ferner gibt es *Kontrollstrukturen*, wie bedingte Anweisungen, Schleifen und Prozeduren. Im Programm definierte Prozeduren können weitere Operatoren und bereits vorhandene Prozeduren enthalten. Sie sind über ihren Namen aufrufbar und ausführbar. Es existieren polymorphe Operanden, wie z.B. *forall*, der als Iterator sowohl für Felder als auch für Wörterbücher verwendet werden kann. Durch das Kellerspeicherkonzept bedingt, sind Aufrufe von Prozeduren und Operanden in *Postfix*-Notation zu tätigen.

Als *Objekte* werden alle vom Programm aus zugreifbaren Daten angesehen. Beispielsweise sind auch Prozeduren Objekte. Jedes Objekt hat einen Typ, eine bestimmte Anzahl von Attributen und einen Wert. Es gibt atomare und zusammengesetzte Objekte. Objekte sind entweder *literal* (z.B. Konstanten) oder *ausführbar* (z.B. Prozeduren).

4.1.1.2 Konzepte zur Graphikerzeugung

Eine Seitenbeschreibung in PostScript ist geräteunabhängig. Ihre wichtigsten Elemente werden kurz eingeführt.

Zeichenprimitive zur Definition von geometrischen Formen, wie Linien, Winkel, Rechtecke, Polygone und Ellipsen.

Operationen zum Setzen von Zeichenattributen, die das Aussehen der geometrischen Formen beeinflussen. Zeichenattribute sind unter anderem: Liniendicke, Linienfarbe, Füllfarbe und Füllmuster sowie Ausschneidepfade (*clipping paths*). Es werden verschiedene Farbmodelle[3] unterstützt.

Texte werden in Standardzeichensätzen beschrieben oder werden benutzerseitig definiert. Sie sind voll graphisch integriert, d.h. alle graphischen Operationen können auf Texte angewendet werden.

Bilder werden punktweise selbst definiert oder beispielsweise durch Scanner eingelesen. Verschiedene Auflösungen und Farbmodelle für Bilder werden unterstützt.

Koordinatensysteme können durch die Operationen Translation, Skalierung und Rotation transformiert werden. Diese Operationen können auf sämtliche Elemente der darzustellenden Seite, auch einzeln, angewendet werden.

PostScript ist hoch flexibel. Es können Seitenbeschreibungen mit komplizierten graphischen Komponenten auf einfache, strukturierte Weise programmiert werden. Durch orthogonale Kombination vorhandener Komponenten können verschiedenartige, komplexen graphischen Formen erzeugt werden. Neu definierte Prozeduren, die wiederum graphische Formen manipulieren können, erweitern die Funktionalität. Mittel zur Strukturierung und Modularisierung von Programmen sind Benennung, Schachtelung und Parametrisierung von Prozeduren. Eine im PostScript Programm definierte Seitenbeschreibung wird durch den Interpreter des jeweiligen Ausgabegeräts in gerätespezifische Ausgabedaten konvertiert und ausgegeben.

4.1.2 NeWS

NeWS ist eine auf PostScript basierende Programmiersprache, die Konzepte für die Bildschirmausgabe in einem verteilten Fenstersystem bereitstellt. Sie ist Teil des X11/NeWS Servers, welcher zur *OpenWindows Version 3* Umgebung gehört [SM92a].

[3] z.B. RGB-, HSB- und das Graustufen-Modell(vgl.[AS90])

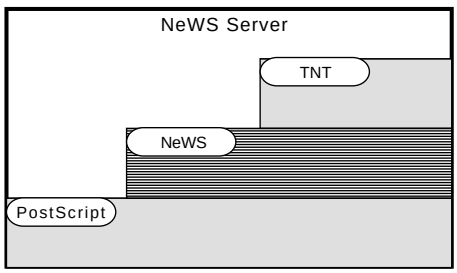


Abbildung 4.4: Dienste des NeWS Servers: NeWS Sprachschicht

NeWS verwendet PostScriptOperatoren, um Text, Bilder und graphische Gebilde auf Bildschirmen auszugeben. Darüberhinaus definiert es weitere Operatoren und Typen, die Mehrbenutzerfähigkeit (*multitasking*) sowie Interaktionsmöglichkeiten im Fenstersystem unterstützen. Im folgenden Abschnitt 4.1.2.1 werden Konzepte zur Erzeugung und Positionierung von Flächen auf dem Bildschirm sowie zur Verwaltung von Interaktion vorgestellt. Im Abschnitt refsecnewsserv4 wird der X11/NeWS Server, der NeWS Programme interpretiert und den eigentlichen Dienst zur Bildschirmausgabe erbringt, beschrieben.

4.1.2.1 Konzepte

Wichtige Erweiterungen von NeWS gegenüber PostScript sind *NeWS Typen* und *NeWS Operatoren*. Mithilfe dieser Konzepte lassen sich *NeWS Objekte* erzeugen. Ein wichtiges NeWS Objekt ist das *Canvas*, welches als Basisbaustein für die Darstellung von Fensteroberflächen verwendet wird. Ereignisse (*rvents*) sind relevant für die Interaktionssteuerung. Die Konzepte werden im folgenden erläutert.

NeWS Typen stellen eine Typerweiterung gegenüber PostScript dar. Sie stellen Datenstrukturen für die Definition von multiplen Bildschirmflächen, für Benutzerinteraktionen, Mehrbenutzerverwaltung und weitere Erfordernisse eines Fenstersystems zur Verfügung. Sie sind in PostScript Wörterbüchern (*dictionaries*). Jeder Typ ist durch Eigenschaften (*properties*) definiert, die über das Wörterbuch zugreifbar und teilweise überschreibbar sind. Beispiele für Typen sind *CanvasType* zur Verwaltung von Flächen auf dem Bildschirm und *EventType* zur Verwaltung von Ereignissen, die durch Benutzereingabe oder Prozesse erzeugt werden und vom NeWS X11 Server „interessierte“ Prozesse weitergeleitet werden.

NeWS Operatoren beziehen sich im wesentlichen auf Interaktionen mit Fensteroberflächen und auf die Definition und Manipulation von Objekten, die durch NeWS Typen definiert sind. Beispiele für Operatoren zur Erzeugung und Positionierung von Objekten vom Typ *CanvasType* auf dem Bildschirm sind *newcanvas*, *movecanvas* und *canvastotop*.

Canvases sind Objekte vom Typ *CanvasType*. Ein Canvas ist eine (nicht notwendigerweise) rechteckige Fläche auf einem Bildschirm. Sie dient als Basisfläche, in der Anwendungen Texte und Graphik anzeigen können. Die Kombination von Flächen (*canvases*) dient u.a. zur Erzeugung von Fensteroberflächen auf dem Bildschirm. Eigenschaften, die teilweise über Operatoren änderbar sind, bestimmen unter anderem die Lage, die Schachtelungsreihenfolge und die Sichtbarkeit einer Fläche auf dem Bildschirm. Weitere Ausführungen zu Canvases sind im Abschnitt 4.1.3 zu finden, wo im Rahmen eines *Klassenkonzepts* die Klasse *ClassCanvas* eingeführt wird, deren Attribute und Methoden auf den hier beschriebenen Eigenschaften und Operatoren aufbauen.

Ereignisse (*events*) sind Objekte zum Nachrichtenaustausch zwischen NeWS Prozessen. Sie stellen einen generischen Mechanismus zur Interprozeßkommunikation dar. Es gibt drei Arten von Ereignissen,

- Eingabeereignisse (*input events*) werden meist durch Maus- oder Tastatureingaben erzeugt.
- Dienstereignisse (*server events*) benachrichtigen über den Zustand von Objekten und Prozessen. Sie finden z.B. bei *garbage collection* Verwendung.
- Synthetische Ereignisse (*synthetic events*) werden von NeWS Prozessen erzeugten, im Server an einen Verteilungsmechanismus geschickt und von dort an „interessierte" Prozesse weiterverschickt.

Beispiele und nähere Ausführungen zu Ereignissen sind in Abschnitt 4.1.3.2 zu finden.

NeWS stellt ferner ein objekorientiertes Klassenkonzept zur Verfügung, welches die Definition von *NeWS Klassen* unter Ausnutzung von Vererbungsmechanismen ermöglicht. Diese konzeptionelle Erweiterung gegenüber NeWS Typen wird vom Werkzeugkasten *The NeWS Toolkit* (kurz: *TNT*) umfangreich genutzt und in Abschnitt 4.1.3 ausführlich beschrieben.

4.1.2.2 Der X11/NeWS Server

Der X11/NeWS Server [SM90a] ist ein auf einem Rechner laufender Prozeß. Er hat die Aufgabe, NeWS Programme zu interpretieren und auszuführen, Ereignisse von Eingabegeräten wie Maus und Tastatur zu verwalten und Bildschirmausgaben zu tätigen. Die Architektur folgt dem *Client-Server* Modell [Sin92] (vgl. Abb. 4.5 aus [Mü94b]). Es wird strikt zwischen dem Diensterbringer (*server*) und dem Dienstabnehmer (*client*) unterschieden, die entkoppelt sind und über ein Protokoll miteinander kommunizieren.

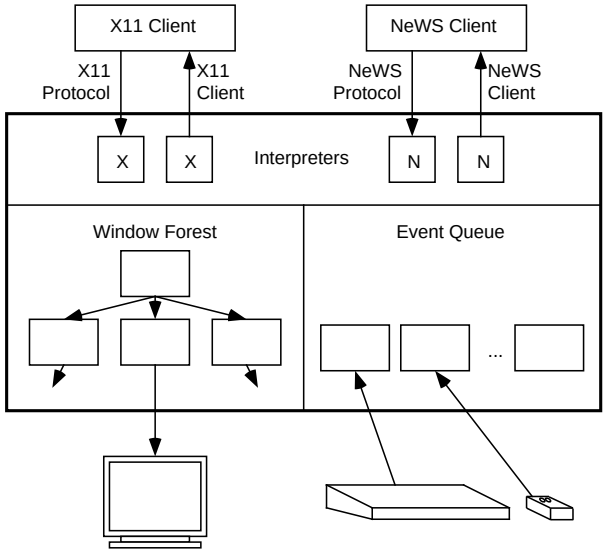


Abbildung 4.5: Architektur des X11/NeWS Servers

Alle Prozesse, die im Adreßraum des Servers und unter seiner Kontrolle laufen, werden als leichte Prozesse (*lightweight processes*) bezeichnet. Sie verbrauchen nur wenig Ressourcen bezüglich

Dauer und Speicherplatz und dienen im wesentlichen zur Kommunikation zwischen NeWS Objekten im Server. Weitergehende Ausführungen zum X11/NeWS Server sind in [SM90a] und [Mü94b] zu finden.

4.1.3 Das NeWS Toolkit (TNT)

Das NeWS Toolkit [SM92a] [SM92b] (*The NeWS Toolkit*, kurz: *TNT*) ist ein Werkzeugkasten (*toolkit*) zur Implementation von in OPEN LOOK spezifizierten Benutzerschnittstellen. Es stellt eine Erweiterung von NeWS dar und besteht im wesentlichen aus einer hierarchisch gegliederten Menge von Klassendefinitionen, die in Vererbungsbeziehungen zueinander stehen. Desweiteren bietet es den *Wire Service*, einen Dienst zur Benachrichtung und zum Verbindungsaufbau zwischen NeWS Server und Anwendungsprogrammen, an. Eine weitere Komponente, der *Jot Service*, dient zur interaktiven Erstellung von Texten.

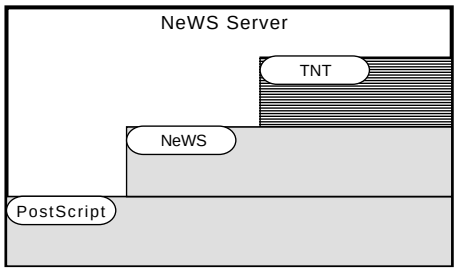


Abbildung 4.6: Dienste des NeWS Servers: TNT Werkzeugkasten

Flexibles Programmieren in TNT wird durch seine Architektur gefördert. Alle Sprachkonzepte von NeWS und PostScript (Operatoren und Typen) sind in TNT verfügbar. Ferner bieten die generellen Klassen eine hinreichend abstrakte Funktionalität, um neben der in Subklassen vorhandenen OPEN LOOK Implementierung neue Subklassenhierarchien zu definieren, die andere, von OPEN LOOK verschiedene, graphische Spezifikationen implementieren. Methoden von existierenden Klassen sind in den Klassen selbst oder in Subklassen überschreibbar. Mithilfe dieser Flexibilität lassen sich anwendungsspezifische Schnittstellen implementieren, die über den OPEN LOOK Standard hinausgehen. In Abschnitt 4.2 wird die Nutzung dieser Flexibilität für die Realisierung der Erzeugung und Verwaltung von OM1 Diagrammen (vgl. Abschnitt 2.3.2) ausführlich beschrieben.

Im folgenden Abschnitt wird der Aufbau der Klassenhierarchie und die damit im Zusammenhang stehenden Begriffe erläutert, und es werden exemplarisch relevante Klassen der Klassenhierarchie beschrieben. In einem weiteren Abschnitt wird der Umgang mit TNT bezüglich Nachrichtenaustausch zwischen Bildschirmobjekten, Interaktionsmöglichkeiten der Benutzeroberfläche sowie Kommunikationsaufbau zwischen NeWS Server und Anwendungsprogramm erklärt.

4.1.3.1 Aufbau und Konzepte

Durch hierarchischen Aufbau von Klassen, verbunden mit Konzepten wie *Vererbung* und *Kapselung*, stellt TNT zu ein mächtiges Werkzeug dar, das Vorteile objektorientierter Sprachen, wie Wiederverwendbarkeit und einfache Änderbarkeit von Programmkode, besitzt. Wichtige Begriffe bezüglich des hierarchischen Aufbaus werden im folgenden erklärt.

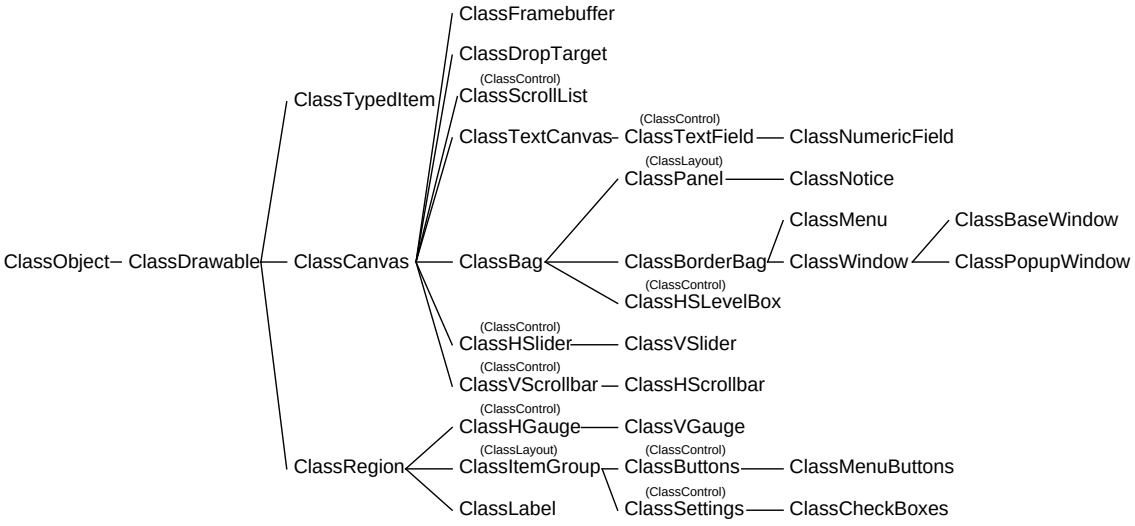


Abbildung 4.7: Die TNT Klassenhierarchie

Klassenhierarchie: In Abb. 4.7 (entnommen aus [SM92b]) sind alle in TNT definierten Klassen mit ihren Vererbungsbeziehungen als Baumstruktur dargestellt. Die weiter links angeordneten sind die *generelleren*, die weiter rechts angeordneten die *spezielleren* Klassen, die OPEN LOOK Komponenten definieren.

Vererbung: Eine *Subklasse* erbt von der *Superklasse* alle Attribute[4] und Methoden. Es gibt *multiple Vererbung*, d.h. eine Klasse kann von mehreren Superklassen erben. Dabei ist die Reihenfolge der Superklassen in einer Liste exakt definiert und signifikant. Bei Namensgleichheit von Attributen oder Methoden von Superklassen derselben Hierarchiestufe entstehen keine Konflikte im Vererbungsmechanismus, weil von der zuerst in der Liste gefundenen Superklasse, die das jeweilige geerbte Attribut bzw. die Methode enthält, geerbt wird. *Methoden*, die von Superklassen geerbt werden, sind auf zweierlei Arten in einer Klasse redefinierbar. Sie können entweder erweitert oder überschieben werden.

Abstrakte Klassen: Eine Klasse wird als *abstrakt* bezeichnet, wenn von ihr keine Instanzen erzeugt werden können. Beispiele dafür sind *ClassObject* und *ClassDrawable*. Sie fassen alle Variablen und Methoden zusammen, die in sämtlichen Subklassen benötigt werden. *ClassLayout* und *ClassControl* sind weitere abstrakte Klassen, die aber nur an ausgewählte Subklassen vererbt werden. Deshalb sind sie im Hierarchiebaum nicht eindeutig positionierbar. Klassen mit diesen Eigenschaften werden in [SM92b] als *Mixin Classes* bezeichnet. In [KGZ93] werden diese Arten von Klassen *Aspektklassen* genannt.

Klasseninstanzen: Von jeder nicht abstrakten Klasse der TNT Hierarchie (das sind *ClassCanvas* und ihre Subklassen) lassen sich über die Methode *NewInit* Instanzen erzeugen, die auf dem Bildschirm anzeigbare und manipulierbare Objekte darstellen. Es gibt zwei Kategorien von Attributen, deren Werte für die Instanzen einer Klasse einen unterschiedlichen Sichtbarkeitsbereich haben. *Klassenattribute* enthalten Werte, die in allen Instanzen einer Klasse identisch sind. Eine Änderung ihrer Werte wird in alle Instanzen propagiert. *Instanzattribute* enthalten Werte, die für jede Instanz explizit änderbar sind.

[4] Attribute werden in [SM92b] als *variables* bezeichnet.

Durch Anwendung der multiplen Vererbung, verbunden mit Methodenerweiterung und -überschreibung in Subklassen, einerseits, sowie der Unterscheidung zwischen Klassen- und Instanzattributen, andererseits, ist Vererbung in Subklassen der TNT Klassenhierarchie sehr flexibel definierbar. Die Wiederverwendbarkeit von Klassen wird durch die granular einstellbare Vererbungsfunktionalität erhöht.

Es folgt eine Auflistung einiger *genereller* Klassen der TNT Hierarchie. Sie sind durch neue Subklassenhierarchien flexibel, d.h. für die Implementation von Flächen und Symbolen *beliebiger Form*, erweiterbar.

ClassObject ist die Wurzel des Klassenbaums. Sie enthält Hilfsfunktionen zur Instanzenerzeugung.

ClassCanvas definiert eine (nicht notwendigerweise) rechteckige Fläche als Verallgemeinerung für die meisten sichtbaren Bildschirmregionen. Ihre Struktur enthält die meisten Methoden und Attribute, um das Aussehen der Bildschirmobjekte und deren Reaktion auf Benutzereingaben (*look and feel* des OPEN LOOK Standards) zu implementieren.

ClassBag ist eine Subklasse von ClassCanvas, die Methoden zur Erzeugung und Verwaltung einer Kollektion von weiteren Flächen (*canvases*) bereitstellt. Eine Instanz dieser Klasse ist somit eine Fläche (*canvas*), die als Behälter fungiert: Auf ihr liegen in bestimmter Anordnung weitere Canvases, die als ihre *Klienten* bezeichnet werden.

ClassPanel ist eine Subklasse von ClassBag. Sie stellt bestimmte Layout Protokolle für mögliche Anordnungen beliebig vieler Klienten zur Verfügung. Sie eignet sich zum Beispiel zur Definition von Diagrammflächen, die aus weiteren, auf ihnen geordnet liegenden Komponenten bestehen. Diese Klasse wird (durch Subklassenerweiterungen) in dieser Arbeit intensiv für die Definition von OM1 Diagrammen (siehe Abschnitt 4.2) verwendet.

Speziellere Klassen der TNT Hierarchie erben von den generellen und implementieren Komponenten der OPEN LOOK Benutzeroberfläche. Es werden exemplarisch einige Klassen vorgestellt. Beispiele für das Aussehen der Komponenten sind in Abbildungen des Abschnitt 3.1 zu finden.

ClassMenuButtons implementiert Menüknöpfe (*menu buttons*).

ClassScrollbar implementiert Rollbalken (*scrollbars*).

ClassBaseWindow implementiert Basisfenster (*base windows*).

4.1.3.2 Umgang mit dem Toolkit

In diesem Abschnitt werden zunächst Formen des Nachrichtenaustauschs zwischen Klasseninstanzen und Klassen vorgestellt. Anschließend wird anhand eines Szenarios die serverseitige Abwicklung von Bildschirmeingaben gezeigt. Zuletzt werden weitere Dienste und deren Verwendung für den Kommunikationsaufbau zwischen NeWS Server und Anwendungsprogramm erklärt.

Kommunikation zwischen Klassen und Instanzen Die Kommunikation erfolgt durch Botschaften (*messages*), die von Klasseninstanzen an beliebige Klassen oder Klasseninstanzen geschickt werden. Sie enthalten Methodenaufrufe oder lesende Zugriffe auf Attribute. In Methodendefinitionen können wiederum weitere Methodenaufrufe auftreten. Es folgen Kodebeispiele.

/map myCanvas ***send***
20 40 /move myCanvas ***send***
/ForegroundColor ClassCanvas ***send***

Die ersten beiden Botschaften sind Methodenaufrufe, die an eine mit dem Namen *myCanvas* assoziierte Klasseninstanz geschickt werden. Die parameterlose Methode */map* bewirkt das Erscheinen des mit der Instanz assoziierten Bildschirmobjekts, die Methode */move* hat zwei Eingabeparameter als Präfix und verschiebt das Objekt auf die entsprechende Position des Bildschirms. Die dritte Botschaft wird an die *ClassCanvas* geschckt und ist ein lesender Zugriff auf ein Attribut, das die (Standard-)Vordergrundfarbe enthält. Der Rückgabewert liegt auf dem Operandenkellerspeicher (*operand stack*, vgl. Abschnitt 4.1.1.1) des NeWS Servers.

Interaktion zwischen Eingabegeräten und TNT. TNT unterstützt die Definition von Methoden zur Verarbeitung von Tastatur- und Maustasteneingaben. Der NeWS Server ist für die Verwaltung von Benutzereingaben un das Senden der damit verbundenen Methodenaufrufe an die „richtigen" Instanzen zuständig. Dieser komplexe Vorgang wird anhand eines Szenarios illustriert, das sich aus den Schritten *Eingabe*, *Serveraktivität* und *Methodenaufruf* zusammensetzt.

Ausgangssituation: Auf dem Bildschirm wird ein kreisförmiges Objekt angezeigt, das eine Instanz der Klasse *ClassCircle*, einer Subklasse von *ClassCanvas*, ist. In der Klasse ist eine Methode definiert, die beim Anklicken des Objekts durch eine Maustaste aufgerufen wird und die ein Verschwinden des Objekts vom Bildschirm bewirkt.

Eingabe: Es wird über dem Kreis, der auf dem Bildschirm angezeigt wird, eine Maustaste gedrückt.

Serveraktivität: Der NeWS Server transformiert die Mauseingabe in ein *Ereignis* (*event*), welches die Koordinaten der Mauseingabe enthält, und schickt es an einen Ereignisverwalter (*event manager*), einen Prozeß innerhalb des Servers. Dieser aktiviert alle Bildschirmobjekte, die sich unter den Mauskoordinaten befinden – also auch das Kreisobjekt – und schickt das Ereignis an eines von ihnen. Alle „nicht interessierten" Objekte geben das Ereignis weiter. Das Kreisobjekt ist an dem Ereignis „interessiert", weil in seiner Klasse die auf das Ereignis reagierende Methode definiert ist.

Methodenaufruf: Die Methode der Klasse *ClassCircle*, in der in diesem Fall eine Löschoperation definiert ist, wird auf die Instanz – das kreisförmige Objekt – angewendet. Das Objekt verschwindet vom Bildschirm.

In TNT sind Methodennamen für Methoden, die auf bestimmte Ereignisse reagieren, vorgegeben. Im Beispiel des Szenarios ist dies die Methode *TrackStop*. Jede Subklasse von *ClassCanvas*, in der eine Methode dieses Namens definiert ist, kann auf Ereignisse reagieren, die durch Maustasten ausgelöst werden. Beim Aufruf der Methode *TrackStop* wird der jeweils in der Methode definierte Kode, im Beispiel die Methode *destroy* zum Entfernen des Objekts, ausgeführt.

Integration von TNT und Anwendungsprogramm Die Aufteilung der Funktionalität einer Anwendung folgt dem *Client-Server Modell*.

Der Server ist der *NeWS X11 Server*, ein laufender Prozeß auf dem für die graphische Ausgabe zuständigen Rechner. Sein Adreßraum besteht aus Wörterbüchern (*dictionaries*), in denen NeWS Klassen – einschließlich der TNT Klassenhierarchie und der möglichen

weiteren, benutzerdefinierten Subklassen – und Instanzen der Klassen definiert sind. Diese Wörterbücher werden bei Bedarf in dafür vorgesehende Kellerspeicher *dictionary stacks* geladen. Der Server hat die Eigenschaften eines *PostScript Interpreters*: Er arbeitet mit weiteren Kellerspeichern (*Operanden-* und *Ausführungskellerspeicher* (vgl. Abschnitt 4.1.1.1). Seine Aufgabe besteht darin, PostScript- und NeWS Operationen (z.B. zur Manipulation von Bildschirmobjekten) zu interpretieren und auszuführen.

Der Client ist ein laufendes Anwendungsprogramm, welches in einer höheren Programmiersprache, z.B. C oder Tycoon, implementiert ist und die eigentliche anwendungsspezifische Funktionalität besitzt. Es kann auf einem anderen Rechner ausgeführt werden als der Server. Sein Adreßraum ist von dem des Servers disjunkt.

Bei der Entwicklung einer Anwendung ist zu überlegen, welche zusätzliche anwendungsspezifische Funktionalität die Bildschirmoberfläche (UI) benötigt. Diese kann durch *NeWS Erweiterungen* wie *Neudefinieren von Subklassen* in der TNT Klassenhierarchie implementiert werden. Subklassenbildung wird mit Beispielen ausführlich in Abschnitt 4.2 behandelt. Die neu definierten Klassen werden zur Laufzeit in den Adreßraum des Servers geladen (durch den *Wire Service* – s.u.) und verleihen ihm damit die gewünschte Funktionalität. Für die eigentliche *Kopplung* zwischen Client und Server ist ein spezieller Verbindungsaufbau, der *Wire Service*, zuständig. Er kann folgende Operationen ausführen.

- Öffnen einer Verbindung vom Client zum Server durch Erzeugen einer neuen *Leitung* (*wire*);
- Laden von NeWS Erweiterungen[5] (*downloading*) in den Adreßraum des Servers;
- Aufrufe von NeWS Methoden vom Client zum Server durch
  - Angabe des Empfängers, das ist eine Klasse oder eine Instanz;
  - Verwendung von entsprechenden Eingabeparametern;
  - Synchronisation bei Methodenaufrufen, die Rückgabewerte liefern;
  - Auslesen der vom Server erzeugten Rückgabewerte in formatierter Form;
- Senden von PostScript Prozeduren (vom Client zum Server), die vom Server direkt ausgeführt werden;
- Installation von Aktionen (*call backs*) mit Hilfe eines Benachrichtigungsmechanismus' (*notifier*), um z.B. vom Server aus Funktionen des Anwendungsprogramms aufzurufen.

4.1.4 Die Tycoon Bibliothek newsenv

Die Tycoon Bibliothek *newsenv* [Mü94b] dient zur Einbindung der in dem Werkzeugkasten TNT (vgl. Abschnitt 4.1.3) definierten NeWS Klassen (kurz: *TNT Klassen*) und deren Methoden in die Tycoon Entwicklungsumgebung. Weitere Basisdienste der Bibliothek realisieren eine effiziente Handhabung des Verbindungsaufbaus und Nachrichtenaustauschs zwischen TL Anwendungsprogrammen und NeWS Servern.

[5] NeWS Erweiterungen sind z.B. neu definierte Subklassen in der TNT Hierarchie mit neuen/erweiterten Methoden. Sie liegen als PostScript Daten vor.

Diese Bibliothek stellt somit einen Dienst zur vollen Nutzung der TNT Funktionalität (vgl. Abschnitt 4.1.3) unter Tycoon dar. Sie ist leicht erweiterbar, weil die Verwendung der Basisdienste den Programmieraufwand reduziert.

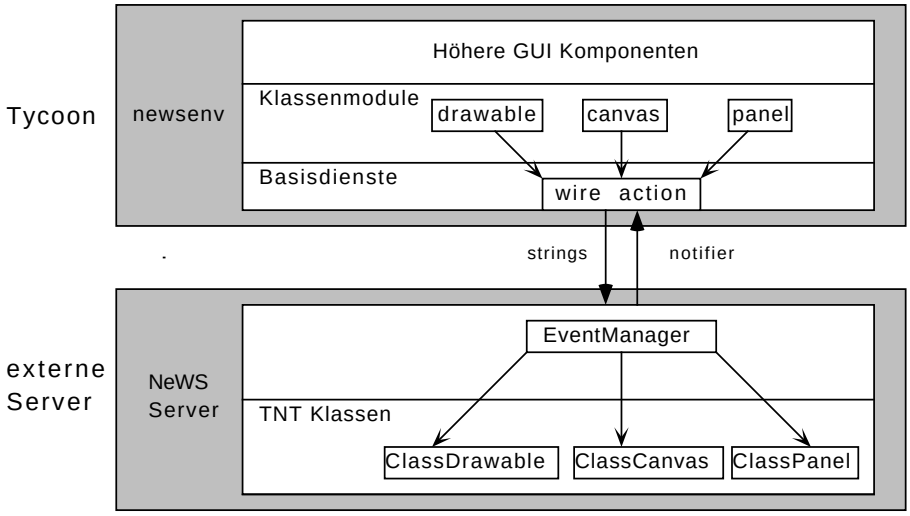


Abbildung 4.8: Dienste Bibliothek *newsenv* zur Einbindung von Diensten des NeWS Servers in Tycoon

Im Abschnitt 4.1.4.1 werden die Basisdienste, die die Kommunikation zwischen TL und TNT realisieren, erläutert. Der Abschnitt 4.1.4.2 beschreibt beispielhaft die Modulen und Funktionen, die die Funktionalität der TNT Klassen in TL einbinden.

4.1.4.1 Basisdienste für die Kommunikation

Die TL Module *wire* und *action* erbringen Dienste für den Kommunikationsaufbau und Nachrichtenaustausch zwischen TL Anwendungsprogramm, das die Rolle des *Dienstnehmers* (*client*) einnimmt, und dem externen NeWS Server, dem *Diensterbringer* (siehe Abb. 4.8). Die Module werden im einzelnen vorgestellt.

Das Modul *wire* bietet Funktionalität für den Verbindungsaufbau und Nachrichtenaustausch zwischen Client und Server an. Die Funktionalität ist zum großen Teil identisch mit der des *Wire Service* unter TNT (vgl. Abschnitt 4.1.3.2). In der Implementation des Moduls *wire* wird auf diesen Dienst über eingebundene *C* Funktionen zugegriffen. Mithilfe dieses Dienstes ist es möglich, vom TL Anwendungsprogramm aus neue NeWS Klassen in den Server zu laden, entfernte Methodenaufrufe mit typisierten Eingabeparametern an den NeWS Server zu schicken und vom Server typisierte Rückgabewerte von ausgewerteten Methodenaufrufen zu erhalten. Es existieren Möglichkeiten zur synchronen und asynchronen Kommunikation. Nähere Ausführungen sind in [Mü94b, S. 72 f.] nachzulesen. Ein Kodebeispiel, das die Benutzung des Moduls *wire* darstellt, ist in Abschnitt 4.3.2 zu finden. Dienste des Moduls *wire* werden sowohl für die Realisierung von Modulen der Bibliothek *newsenv* (vgl. Abschnitt 4.1.4.2) als auch für die Realisierung der *Erweiterung* des *newsenv* zur Einbindung neuer TNT Klassen (vgl. Abschnitt 4.3) verwendet .

Beim *Starten* eines TL Anwendungsprogramms, welches das Modul *wire* benutzt, wird implizit ein Verbindungsaufbau zum NeWS Server, der dem Tycoon Programm zugeordnet ist, getätigt, und es werden die NeWS- und TNT Umgebungen in den Server geladen. Danach können durch Aufrufe der Funktion *wire.loadPostScript* vom Programm benötigte *benutzerdefinierte* TNT Klassen, welche *Erweiterungen* der TNT Klassenhierarchie sind, in den NeWS Server geladen werden.

Das Modul *action* dient zur Ereignisverwaltung. Eine Aufgabe ist das Erzeugen von *Aktionen*, die TL Funktionen vom NeWS Server aus, z.B. nach Benutzereingaben auf Interaktionskomponenten der Benutzerschnitstelle, (als *call backs*) aufrufbar machen. So wird es beispielsweise ermöglicht, beim Drücken von Knöpfen über Maustasteneingaben auf dem Bildschirm Aufrufe von TL Funktionen zu tätigen. Eine zweite Aufgabe besteht in der Verteilung von ankommenden Ereignissen aus dem NeWS Server, die vom TL Anwendungsprogramm zur Laufzeit verarbeitet oder weitergereicht werden. Dieser Dienst wird durch den Funktionsaufruf *action.waitAndDispatch()* erbracht. Genauere Ausführungen zur Ereignisverwaltung werden in [Mü94b] gemacht.

4.1.4.2 Klassenschnittstellen zu TNT

Zu jeder Klasse der TNT Klassenhierarchie gibt es eine TL Schnittstelle und ein Modul, in dem die meisten Methodenaufrufe der Klasse durch jeweils korrespondierende TL Funktionen implementiert sind (vgl. Abb. 4.8). Dadurch wird in Tycoon die volle Funktionalität der TNT Klassenhierarchie zur Erzeugung und Manipulation von OPEN LOOK Benutzerschnittstellen im Tycoon System verfügbar gemacht. Die Benutzung der TL Funktionen für die Methodenaufrufe hat folgende Vorteile.

- Eingabeparameter der Methoden können durch Funktionsparameter definiert werden. Dadurch wird die Syntax der Aufrufe vereinfacht.
- Die Typisierung von Funktionsparametern läßt eine Typüberprüfung zur Übersetzungszeit zu und fördert somit die Typsicherheit.
- Aufwendige Protokolle für die Kommunikation zwischen Client und Server bleiben verborgen.
- Eine Applikation kann *eine* Programmiersprache, nämlich TL, sowohl für die Programmierung der Bildschirmgraphik als auch für die Implementation weiterer, anwendungsspezifischer Funktionalität verwenden. Insbesondere lassen sich komplexe Graphikdaten zur Laufzeit von der TL Seite über den NeWS Server in die Tycoon Umgebung laden, um dort effizient verwaltet zu werden.

Im folgenden Kodebeispiel wird die Syntax von TL Funktionsaufrufen dargestellt. Es werden Instanzen der Klassen *Canvas* und *Panel*[6] erzeugt[7] und ihre Größe erfragt.

```
let myCanvas = canvas.create(object.framebuffer)
let myPanel = panel.createCalculated(object.framebuffer)
let canvasSize = drawable.size(myCanvas)
let panelSize = drawable.size(myPanel)
```

[6] Bei der Erzeugung einer Instanz von ClassPanel wird ein *kalkuliertes Layoutprotokoll* für die Anordnung von Klienten definiert.

[7] Jedem Bildschirmobjekt wird bei seiner Erzeugung der *FrameBuffer*, das ist die Hindergrundfläche des Bildschirms, als Untergrundfläche zugeordnet.

Es wird deutlich, daß Methodenaufrufe, die in der TNT Hierarchie in Superklassen definiert sind und von Subklassen geerbt werden, auch im *newsenv* nur *einmal* als Funktion definiert sind und auf Instanzen verschiedener Klassen als Parameter angewendet werden können. Die Funktion

drawable.size(d :Drawable.T) :Drawable.Size

ist beim Aufruf sowohl durch die *ClassCanvas* Instanz *myCanvas* als auch durch die *ClassPanel* Instanz *myPanel* parametrisierbar. Dies ist aufgrund der Subtypbeziehungen von *Canvas.T* und *Panel.T* zum Typ *Drawable.T*. In [Mü94b] werden Beispiele gegeben, die die Umsetzung der vielfältigen Methodenaufrufe des TNT Werkzeugkastens durch korrespondierende TL Funktionen ausführlicher beschreiben.

4.2 Realisierung der Diagramme durch neue TNT Klassen

Benutzerschnittstellen und deren Komponenten, deren Aussehen und Funktionalität (*look and feel*) im OPEN LOOK Standard festgelegt sind, sind durch spezielle Subklassen in der TNT Klassenhierarchie implementiert. Die in Abschnitt 2.3.2 erfolgte Spezifikation von *OM1 Diagrammen* setzt eigene, von OPEN LOOK verschiedene Stilrichtlinien. Um diese zu implementieren, ist eine *Erweiterung der TNT-Klassenhierarchie* durch neue Subklassen erforderlich.

Die Erweiterung umfäst einerseits die Definition der auf dem Bildschirm anzuzeigenden Form (*look*) von Basissymbolen und von aus Basissymbolen zusammengesetzten OM1 Diagrammen und andererseits die Defintion von Funktionalität zur *Erzeugung und Manipulation* (*feel*) von Basissymbolen in den Diagrammen, die vom Graphikeditor aufrufbar sein soll. Beim Entwurf und der Realisierung von neuen Klassenkonzepten werden die folgenden Verfahrensweisen angewendet.

Generalisierung und Spezialisierung durch Erweiterung der TNT Hierarchie um Zwischenebenen, wobei Vorteile der (multiplen, mehrstufigen) Vererbung genutzt werden können.

Dekomposition von komplexen Bildschirmobjekten in weniger komplexe Teilobjekte. Mehrere Klassen, die jeweils eine einfache Fläche definieren, werden zur Erzeugung eines geschachtelten Bildschirmobjekts – durch Übereinanderlegen der Flächen – benutzt.

Kategorisierung der Klassen nach dem Leitbild der *Werkzeug-Material* Metapher (siehe Abschnitt 4.2.1). Durch diese Metapher werden Klassen aus semantischer Sicht nach Art ihres Umgangs unterschieden. Diese Art der Konzeptbildung unterstützt die Übersichtlichkeit und Änderbarkeit des Systems.

Zunächst wird im Abschnitt 4.2.1 das Leitbild *Werkzeug-Material* motiviert. In den weiteren Abschnitten werden Erweiterungen der TNT-Klassenhierarchien systematischen eingeführt, jeweils unter Berücksichtigung der oben erwähnten Verfahrensweisen. Im Abschnitt 4.2.2 werden die *Materialklassen* vorgestellt, die die Basissymbole der OM1 Diagramme definieren. Der Abschnitt 4.2.3 beschreibt die *Materialklassen* zur Definition von OM1 Diagrammen. Im Abschnitt 4.2.4 folgt die Beschreibung der *Werkzeugklassen*, die Dienste zur Erzeugung und Manipulation bereitstellen.

4.2.1 Werkzeug und Material als Leitbild

Um die Funktionalität zur Erzeugung, Anzeige und Manipulation von Diagrammen in effizient strukturiert zu implementieren, eignet sich das in [BKKZ92] beschriebene Leitbild *Werkzeug*

und Material als Metapher für die objektorientierte Softwareentwicklung.

Editoren sind interaktive Anwendungssysteme und lassen sich als Werkzeuge und Materialien in einer Arbeitsumgebung modellieren [BKKZ92]. Der Graphikeditor ist als *reaktives System* aufzufassen. Eine Benutzereingabe, die als Ereignis an das System weitergereicht wird, aktiviert dort Komponenten, die eine Dienstleistung erbringen. Danach wartet das System auf die nächste Benutzereingabe. Die Funktionalität, die durch die Dienstleistung erbracht wird, ist somit *benutzergesteuert* [BKKZ92]. Im Gegensatz dazu gibt es *Routinesysteme*[8], die meist über eine Folge von Menümasken eine rein sequentielle Bearbeitung vorschreiben: Der Arbeitsablauf ist *systemgesteuert* [Obe92] [BKKZ92] [Flo94].

Um die Werkzeug-Material Metapher sinnvoll einzusetzen, ist der Vergleich eines reaktiven Systems mit einer *Werkstatt* zweckmäßig. In einer Werkstatt werden Werkzeuge benutzt, mit denen Arbeitsgegenstände als Material bearbeitet werden. Im folgenden werden Werkzeugklassen, Materialklassen sowie *Aspektklassen*, die von Materialklassen abstrahieren, charakterisiert.

Werkzeugklassen beschreiben die Umgangsformen eines Benutzers mit einer Anwendung. Sie realisieren die interaktive Komponente eines Systems; durch sie werden *Softwarewerkzeuge* definiert.

Materialklassen definieren die Eigenschaften von Arbeitsgegenständen[9], die in einem Softwaresystem durch Softwarewerkzeuge bearbeitet werden. Die Eigenschaften geben nicht nur Auskunft über Form und Zustand des Arbeitsgegenstands, sondern auch über die Bearbeitbarkeit durch bestimmte Werkzeuge.

Aspektklassen fassen jeweils genau diejenigen Eigenschaften von Materialien zusammen, die eine Bearbeitung durch ein bestimmtes Werkzeug ermöglichen. Diese Eigenschaften (Aspekte) betreffen jeweils meist verschiedene Materialklassen. Eine Aspektklasse entsteht also durch *Generalisierung* von gemeinsamen Eigenschaften mehrerer Materialklassen.

Es folgen einige Beispiele für die Aufteilung der Komponenten des Graphikeditors in Werkzeug und Material.

Materialklassen zuzuordnen sind:

- OM1 Diagramme, wie Referenzdiagramm und Vererbungsdiagramm;
- Basissymbole, die in den Diagrammen vorkommen;
- Sichten des Referenzdiagramms;
- Dateien, wie druckbare PostScriptdateien, und Dateien, die Graphiken abstrakt beschreiben.

Aspektklassen zuzuordnen sind:

- druckbare Dokumente;
- größenskalierbare Dokumente.

[8] Als Anwendungsbeispiel ist die Scheckbearbeitung in einer Bank zu nennen, bei der die Reihenfolge der Arbeitsschritte vorgegeben ist.

[9] Gemeint sind sowohl physisch vorhandene Arbeitsgegenstände, wie z.B. Ausdrucke, als auch virtuelle, wie z.B. ein Formular auf einer Bildschirmoberfläche.

Werkzeugklassen zuzuordnen sind:

- der Graphikeditor;
- Editoren zum Editieren des OM1 Referenzdiagramms und des OM1 Vererbungsdiagramms, jeweils mit Subkomponenten wie *Anzeiger* und *Entferner*;
- Sichtenanzeiger für das Referenzdiagramm;
- Druckdatei-Formatierer;
- Textgeneratoren;
- Drucker.

Die Klassenstrukturierung nach der Werkzeug-Material Metapher erweist sich bei der Systementwicklung aus folgenden Gründen vorteilhaft.

- Vorgabe einer *sinnvollen Modularisierung* Die Modellierung der Werkzeugklassen erfolgt nach anwendungsspezifischen Kriterien. Ihre Funktionalität wird durch den Umgang bestimmt, sie lassen sich semantisch klar voneinander kapseln. Sowohl Werkzeugklassen als auch Materialklassen eines Systems können durch Vererbungshierarchien (in Super- und Subklassen) strukturiert sein.
- Leicht zu handhabende *Änderbarkeit* des Systems Durch die *Trennung* von Werkzeugen und Materialien in verschiedene Klassen bleiben Änderungen des Systems lokal begrenzt.
  - Materialien sind im allgemeinen relativ stabil. Beispielsweise Scheckformulare in einer Bank oder ER-Diagramme bei der Datenbankmodellierung ändern sich selten.
  - (Software-)Werkzeuge hingegen werden häufiger in ihrer Funktionalität den Bedürfnissen und Arbeitsweisen von Anwendern angepaßt. Bei einer Änderung muß nur die jeweilige Werkzeugklasse (und jede der eventuell vorhandenen Subklassen) geändert werden. Alle *Materialklassen* bleiben unverändert [BKKZ92].
- *Modulare Erweiterbarkeit* Neue Werkzeugklassen, die bei der Systementwicklung hinzukommen, lassen sich einfügen, ohne bereits vorhandene Werkzeugklassen oder Materialklassen ändern zu müssen.

4.2.2 Materialklassen für Basissymbole

Basissymbole sind die graphischen Komponenten, aus denen sich die OM1 Diagramme zusammensetzen (vgl. Abschnitt 2.3.2). Das OM1 Referenzdiagramm besteht aus Klassenknoten, Referenzkanten und Werteknoten. Das OM1 Vererbungsdiagramm enthält Klassenknoten und Vererbungspfeile. Das Aussehen und die Semantik der Basissymbole wird in Abschnitt 2.3.2 erklärt. Zum Klassenentwurf werden, wie erwähnt, folgende Verfahrensweisen angewendet.

- Generalisierung und Spezialisierung;
- Dekomposition von komplexen Bildschirmobjekten in weniger komplexe Teilobjekte;
- Kategorisierung der Klassen nach dem Leitbild der *Werkzeug-Material* Metapher.

Nach der *Werkzeug-Material* Metapher sind die Basissymbole als *Material* zu kategorisieren. In jedem der folgenden Abschnitte werden die Klassenkonzepte für eine Basissymbolart unter Einbeziehung der oben dargelegten Verfahrensweisen eingeführt. Die durch die Klassen erzeugbaren Objekte werden jeweils durch eine Abbildung veranschaulicht, die Vererbungsbeziehungen der Klassen durch die Anzeige des für sie relevanten Teilbaums der erweiterten TNT-Klassenhierarchie. Wichtige Attribute und Methoden der Klassen werden exemplarisch beschrieben.

Vorweg seien die *Gemeinsamkeiten* der Definition neuer Subklassen für Basissymbole erwähnt. Wie in Abschnitt 4.1.3.1 beschrieben, werden alle anzuzeigenden Bildschirmobjekte durch *Flächen* realisiert, die weitgehend durch Methoden der TNT-Klasse *ClassCanvas* definiert sind. Alle Klassen, die *speziellere* Bildschirmobjekte definieren, nutzen oder überschreiben die Attributwerte und Methoden von *ClassCanvas*, bzw. fügen neue Attribute und Methoden hinzu. Somit sind alle für die Basissymbole relevanten Klassen (direkte oder indirekte) Subklassen von *ClassCanvas*.

4.2.2.1 Klassenknoten

Abbildung 4.9: TNT Klassen zur Erzeugung von Klassenknoten

Ein Klassenknoten besteht aus einem beschrifteten Kreissymbol. Zwei Alternativen von Füllfarben stellen optisch dar, ob ein Klassenknoten für bestimmte Editoroperationen[10] *selektiert* wurde oder nicht (Abb. 4.9 links und rechts). Beim Erzeugen eines Klassenknotens wird der Name in einem editierbaren Textfeld vom Benutzer eingegeben.

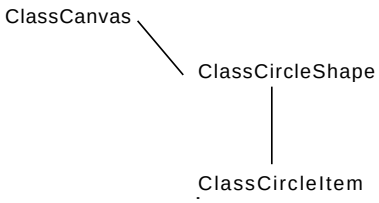


Abbildung 4.10: Hierarchieausschnitt: TNT Subklassen zur Klassenknotendefinition

Bei der Realisierung wird nach den folgenden Verfahrensweisen vorgegangen. *Generalisierung/Spezialisierung* wird für die Definition des Kreissymbols verwendet. In *ClassCircleShape* wird die äußere Form einer Fläche durch die Methode *path* als kreisförmig definiert. Die gleichnamige Methode der Superklasse *ClassCanvas*, die eine rechteckige Form definiert, wird überschrieben, wie das folgende Beispiel von Kodefragmenten verdeutlicht.

[10] Z.B. Einfügen einer Referenzkante oder eines Werteattributs zu einem selektierten Klassenknoten.

```
/ClassCanvas ClassDrawable
    /path {
        rectpath
    } def
    ...

/ClassCircleShape ClassCanvas
    /path {
        ...
        0 360 arc
    } def
```

In der Klasse *ClassCircleItem* (ohne Kodebeispiel), einer Subklasse von *ClassCircleShape*, existiert das boolesche Attribut *Selected?*, welches durch Werkzeugklassen gesetzt wird, bestimmt die Füllfarbe des Kreissymbols. Die überschriebene Methode *Paint* schließlich „malt“ den Kreis unter Berücksichtigung der kreisförmig definierten Formde in *path* und der im Attribut *Selected?* gesetzten Füllfarbe.

Durch *Dekomposition* teilt sich der Klassenknoten in ein *Kreissymbol*, welches von der Klasse *ClassCircleItem* erzeugt wird und einen *Label*, der durch die Klasse *ClassTextLabel* als unsichtbare Fläche mit einer zentrierten Zeichenkette definiert wird. Eine Kreissymbolinstanz hat „Kenntnis“ von der ihr zugeordneten Labelinstanz durch das Attribut *CurrentLabel*, das die Labelinstanz assoziiert und bei der Erzeugung gesetzt wird.

4.2.2.2 Referenzkante

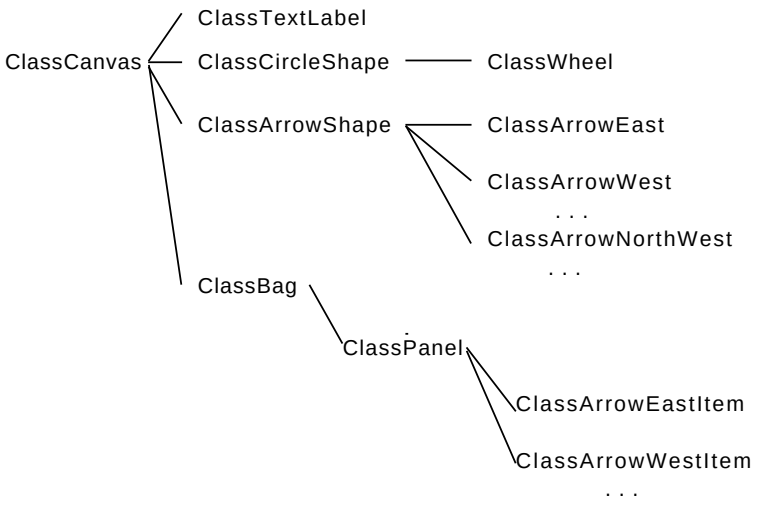


Abbildung 4.11: Hierarchieausschnitt: TNTSubklassen zur Referenzkantendefinition

Es müssen 16 verschieden gerichtete Referenzkanten definiert werden, umdie möglichen acht Verbindungsrichtungen[11] zwischen benachbarten Klassenknoten mit je zwei Varianten zu realisieren. Für Verbindungen im *Erweiterten Rastermodus* (vgl.Abschnitt 3.3.2) existieren weitere 16 Referenzkantenarten.

[11] Die Richtungen – je zwei waagerechte, senkrechte und vier diagonale – sind nach den Himmelsrichtungen benannt: *North*, *NorthWest* etc.

In der Klasse *ClassArrowShape* werden die Grundformen der Pfeile sowie Drehungen der Grundformen durch PostScript-Malprozeduren definiert. Durch *Spezialisierung* von *ClassCanvas* und *ClassArrowShape* wird für jede Kantenrichtung eine Subklasse definiert, in der jeweils eine spezielle Malprozedur in der Methode *Paint* und eine spezielle äußere Form in der Methode *Path* spezifiziert werden. Die möglichst genaue Festlegung der äußeren Form ist Voraussetzung für die interaktive *Selektierbarkeit* einer Referenzkante. Die Default-*Path* Methode würde eine viel zu große rechteckige Fläche definieren, die die selektive Auswahl von sich überschneidenden Referenzkanten unmöglich macht.

Eine Referenzkante ist ein geknickter, beschrifteter Pfeil, der von einem Klassenknoten auf einen zweiten, im gerasterten Diagramm benachbarten, Klassenknoten weist.

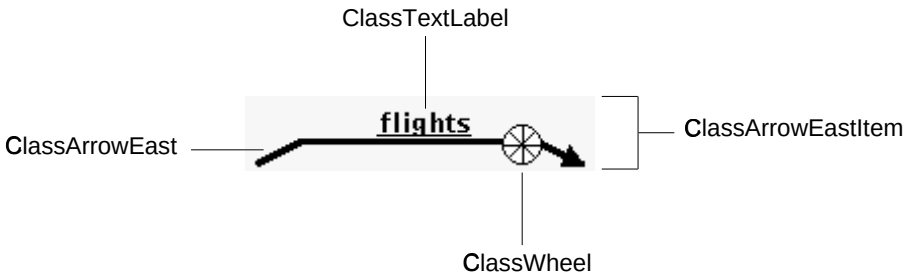


Abbildung 4.12: TNT Klassen zur Erzeugung einer Referenzkante

Durch *Dekomposition* werden die Referenzkante, ihr zugehöriges Label und ein eventuell vorhandenes *Wagenradsymbol*[12](für die Kennzeichnung von Mengenwertigkeit) von verschiedenen Klassen erzeugt und in einer (unsichtbaren) *Behälterfläche* durch entsprechende Layoutprotokolle positioniert. Im Beispielt der Abb. 4.12 sind die Referenzkante und die Behälterfläche durch die Klassen *ClassArrowEast* und *ClassArrowEastItem* definiert. Die Methode *setUnderline* in *ClassTextLabel* wird beim Erzeugen der Referenzkante aufgerufen, um die Darstellungsalternativen des Labels (*unterstrichen/ nicht unterstrichen*) zu setzen.

4.2.2.3 Werteknoten

Ein Werteknoten ist ein Kasten, der durch eine Kante mit einem Klassenknoten verbunden ist. In diesem Kasten wird jedes Werteattribut durch eine Linie mit einem Quadrat am Ende dargestellt. Die Linie ist mit dem Attributnamen beschriftet, das Quadrat mit dem Typnamen. Die Linien der Werteattribute laufen gabelförmig zusammen.

[12] Das Wagenradsymbol wird durch die Klasse *ClassWheel*, einer weiteren Subklasse der kreisdefinierenden Klasse *ClassCircleShape*, erzeugt.

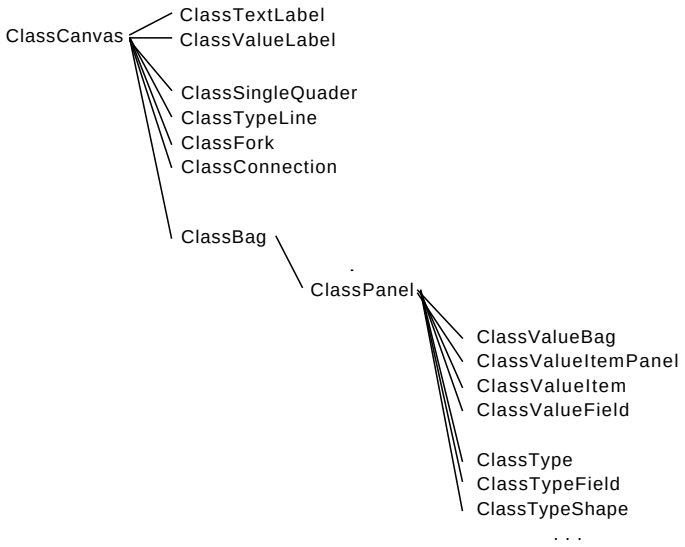


Abbildung 4.13: Hierarchieausschnitt: TNT Subklassen zur Werteknotendefinition

Bei der Realisierung werden keine neuen Zwischenebenen in der Vererbungshierarchie eingeführt. Ein Werteknoten besteht aus einfachen Flächen, deren definierende Klassen direkte Subklassen von *ClassCanvas* sind, und aus *Behälterflächen*, die weitere Flächen als Komponenten enthalten. Behälterflächen sind durch direkte Subklassen von *ClassPanel*[13] definiert.

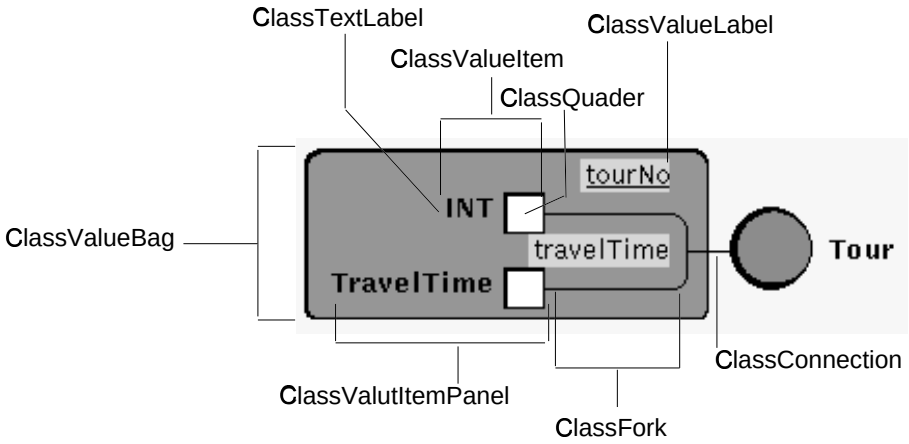


Abbildung 4.14: TNT Klassen zur Erzeugung von Werteknoten

Die *Dekomposition* ist hier das wesentliche Verfahren zur effizienten Realisierung von Werteknoten. Die nachfolgenden Beschreibungen der Komponenten sind ausführlich, um zu verdeutlichen, wie *komplexe Graphiksymbole* durch intensive Nutzung der *Dekomposition* von Flächen definiert und erzeugt werden können.

[13] *ClassPanel* stellt vor allem *Layoutprotokolle* als Methoden zur Anordnung von Komponenten zur Verfügung.

Die alle weiteren Komponenten beinhaltende Komponente ist der äußere Rahmen des Werteknotens, dessen Größe sich nach der Größe und Anzahl der in ihm enthaltenen Atttibutzeilen richtet. Er ist ein Behälter (definiert in *ClassValueBag*), der zwei weitere Komponenten enthält, eine Fläche mit einem gabelförmigen Symbol (Klassenname: *ClassFork*) und einen Behälter (Klassenname: *ClassValueItemPanel*) für die beschrifteten Werteattribute. Jedes Werteattribut (Klassenname: *ClassValueItem*) ist wiederum ein Behälter, der jeweils ein Label (Klassenname: *ClassTextLabel*) für den Attributnamen und den Attributtyp sowie die geometrische Form, Quadrat mit Verbindungslinie (definiert in den Klassen *ClassQuader* und *ClassTypeLine*, enthält (vgl. Abb. 4.14).

Die die Komponenten erzeugenden Klassen werden im einzelnen beschrieben.

ClassValueBag definiert den Rahmen des Werteknotens als kastenförmige Behälterfläche. Die wichtigsten Attribute und Methoden werden im folgenden beschrieben.

- *ShowStatus*: Dieses Attribut gibt an, ob ein Werteknoten angezeigt wird oder nicht; das Attribut wird bei der Ebenenauswahl durch den Benutzer gesetzt (vgl. die Spezifikation der Ebenen in Abschnitt 2.3.2).
- *Paint*: Nur wenn das Attribut *ShowStatus* den Wert „sichtbar“ (*visible*) besitzt, wird der Werteknoten angezeigt. Dafür wird die Größe neu berechnet (durch Abfrage der Größe des Behälters, der die Werteattribute enthält), eine abgerundete Umrahmung gezeichnet und die Hintergrundfläche mit grün (bzw. grau auf einem Schwarzweißbildschirm) gefüllt. Anschlïeßend werden die Komponenten gezeichnet (durch Aufruf der geerbten Methode *PaintChildren*).

ClassValueItemPanel definiert den „Wertebehälter“, der als Komponenten vertikal übereinander liegend die Werteattribute enthält. Die Höhe des Behälters wird aus der Anzahl der Werteattribute berechnet, die Breite ist die des breitesten Werteattributs. Layoutprotokolle definieren die zueinander relative Anordnung der Komponenten.

ClassFork definiert eine Gabelform. Die Höhe einer Gabelinstanz bzw. die Anzahl der anzuzeigenden Kanten werden aus der Anzahl der im Wertebehälter enthaltenen Werteattribute errechnet und in der Methode *Paint* beim Zeichnen der Gabelform verwendet.

ClassValueItem definiert ein Werteattribut, welches sich aus weiteren Komponenten und Subkomponenten zusammensetzt, die kurz, unter Angabe der sie definierenden Klassen, beschrieben werden

- *ClassValueLabel* definiert ein Label, das den (eventuell unterstrichenen) Attributnamen[14] anzeigt.
- *ClassTextLabel* definiert ein Label, das den Wertetypnamen anzeigt.
- *ClassTypeShape* ist eine Behälterklasse. Die Komponentenklasse *ClassTypeLine* definiert eine Attributkante, *ClassQuader* ein Quadrat.[15]

ClassConnection definiert eine Kante, die den Werteknoten mit dem Klassenknoten verbindet.

[14] Ein unterstrichener Attributname bedeutet, daß es sich um ein Schlüsselattribut handelt.
[15] *ClassTypeShape* und *ClassTypeLine* werden in der Abb. 4.14 nicht angezeigt.

4.2.2.4 Vererbungspfeil

Ein Vererbungspfeil ist ein breiter, weißer Pfeil, der vom erbenden zum vererbenden Klassenknoten weist. Er kann beliebig lang sein und in alle nicht abwärts weisenden Richtungen zeigen.

Die in vorigen Abschnitten zur Realisierung verwendeten Verfahrensweisen *Generalisierung/Spezialisierung* und *Dekomposition* sind für die Definition von Vererbungspfeilen nicht von Bedeutung. Ein Vererbungspfeil besteht aus einer einzigen Fläche und enthält keine weiteren Komponenten. Um unterschiedlich lange, in verschiedene Richtungen weisende Pfeilsymbole zu erzeugen, ist, im Gegensatz zu den TNT Hierarchieerweiterungen für die Referenzkanten, nur *eine neue Subklasse* von *ClassCanvas*, *ClassInheritArrow*, zu definieren. Die Vielfalt der Pfeilformen wird durch Methoden dieser Klasse und der Werkzeugklasse *ClassInsertInheritArrows* (siehe dazu Abschnitt 4.2.4.5) erzeugt. Wesentliche Vorgehensweisen für die Methodendefinitionen sind

1. Algorithmische Berechnung von Pfeilkantenlängen und Drehungen unter Verwendung von in PostScript definierten geometrischen Funktionen;

2. Erzeugung von Instanzattributen (vgl. Abschnitt 4.1.3.1), die für jede erzeugte Instanz individuelle, nur für diese Instanz geltende Werte enthalten.

Die algorithmische Berechnung von Länge und Neigungswinkel eines zu erzeugenden Vererbungspfeils geschieht bereits in der Werkzeugklasse *ClassInsertInheritArrows* in Form von geometrischer Auswertung der Koordinaten der zu verbindenden Klassenknoten. Dies wird in Abschnitt 4.2.4.5 beschrieben. Nach der Berechnung werden die Start- und Zielkoordinaten, die Länge und der Neigungswinkel als Eingabeparamater der Methode *NewInit* verwendet, die an die Klasse *ClassInheritArrow* gesendet wird und einen neuen Vererbungspfeil erzeugt. Die Methode *NewInit* wird im folgenden anhand eines Kodebeispiels näher erläutert. Die Methoden *path* und *Paint* werden anschließend erklärt.

```
/NewInit { % rotate scale x y
   /NewInit super send
   /Y exch promote /X exch promote
   ScaleVal exch promote
   Angle exch promote
      ....
}def
```

NewInit erzeugt einen Vererbungspfeil als neue Instanz. Sie erweitert die gleichnamige in *ClassCanvas* definierte Methode um folgende Befehle.

- */NewInit* ***super send*** ruft die von *ClassCanvas* geerbete Methode *NewInit* auf. Es wird der *Framebuffer* als „Elternfläche“ initialisiert.
- Die weiteren Befehle dienen zur Initialisierung von Instanzattributen durch die Eingabeparameter, die im Kodebeispiel als Kommentar hinter dem %-Zeichen der ersten Zeile angegeben sind. Es sind
  - */Y* ***exch promote*** */X* ***exch promote***: Initialisieren der Attribute *X* und *Y* mit den Startkoordinaten;
  - */ScaleVal* ***exch promote***: Setzen des Attributs *ScaleVal* mit dem Skalierungsfaktor;

– */Angle* ***exch promote***: Setzen des Attributs *Angle* mit dem Winkel.

Der NeWSBefehl ***promote*** bewirkt, daß eine Kopie eines Klassenattributs angelegt wird, die nur für eine Klasseninstanz (in diesem Falle: für diesen Vererbungspfeil) sichtbar ist. Semantisch gesehen wird das Klassenattribut in ein Instanzattribut umgewandelt.

path benutzt die in *NewInit* gesetzten Instanzvariablen *ScaleVal* und *Angle*, um die individuelle Form des Pfeils durch Skalierung und Drehung zu definieren.

Paint wiederum benutzt *path*, um die Pfeilform anzuzeigen und mit weißer Farbe zu füllen.

4.2.3 Materialklassen für Diagramme

Wie in Abschnitt 2.3.2 spezifiziert, gibt es in der graphischen Modellierung von OM1 zwei Diagrammarten, die getrennt angezeigt werden, das *Referenzdiagramm* und das *Vererbungsdiagramm*. Bei dem Entwurf und der Realisierung von Klassen für die Diagrammarten werden die in Abschnitt 4.2 aufgeführten Verfahrensweisen verwendet. In Anwendung der *Werkzeug-Material* Metapher Diagramme als *Material* zu kategorisieren. Sie werden durch *Werkzeuge* (Editoren) bearbeitet (erzeugt, angezeigt, manipuliert etc.). Die Gemeinsamkeit der neu definierten Klassen in der TNTKlassenhierarchie besteht darin, daß sie alle Subklassen von *ClassPanel* sind, also jeweils eine *Behälterfläche* definieren, in der die Anordnung der in ihr enthaltenen Komponenten durch Layoutprotokolle bestimmt ist. Die Methoden von *ClassPanel* werden für das Layout der Basissymbole in den Diagrammen verwendet.

Die Klassenkonzepte werden in drei Abschnitten, jeweils unter Bezugnahme auf eine Verfahrensweise, eingeführt. Im ersten Abschnitt wird die *Dekomposition* von Diagrammen in Teildiagramme behandelt. Der zweite Abschnitt stellt *Generalisierung/Spezialisierung* durch geeignete neue Teilhierarchien von TNT Klassen dar. Zuletzt folgt *Werkzeug-Material* Metapher als Verfahren zur Realisierung der Konzepte.

4.2.3.1 Dekomposition von Diagrammen

In diesem Abschnitt wird das Hauptgewicht auf die Erläuterung der spezifischen Komponenten des *Referenzdiagramms* gelegt. Die Komponenten des *Vererbungsdiagramms* werden nur kurz am Ende erwähnt. Sie überschneiden sich teilweise mit denen des Referenzdiagramms oder werden analog zu ihnen gebildet.

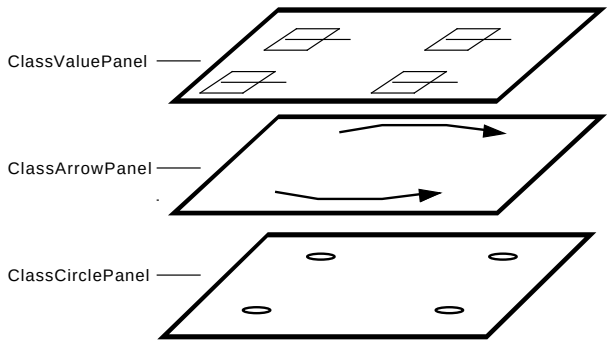


Abbildung 4.15: TNT Klassen zur Erzeugung von Flächen für Teildiagramme

Durch *Dekomposition* wird das Referenzdiagramm in drei übereinander liegende Komponentenflächen aufgeteilt, die jeweils als Teildiagramm nur eine Art von Basissymbolen enthalten (vgl. Abb. 4.15). Es gibt also eine Fläche mit Klassenknoten, eine mit Referenzkanten und eine mit Werteknoten. Diese drei Flächen haben jeweils einen durchsichtigen Hintergrund, sind gleich groß und werden bei der Anzeige des Diagramms übereinander auf dem Bildschirm dargestellt. Es entsteht optisch der Eindruck *eines* Diagramms. Die Teildiagramme und die das Gesamtdiagramm bildende Behälterfläche werden im folgenden beschrieben. Dabei werden die sie definierenden Klassen mit jeweils nur teildiagramm-spezifischen Attributen und Methoden erläutert, weil die meisten Methoden für die Diagrammverwaltung in der Superklasse *ClassGraph* definiert sind (siehe Abschnitt 4.2.3.2).

Das Klassenknoten-Teildiagramm enthält die Klassenknoten mit jeweiligen Labeln als Komponenten und wird durch die Klasse *ClassCirclePanel* definiert. Im folgenden werden relevante Methoden und Attribute genannt.

GetSelectedCircle referenziert im Diagramm den Klassenknoten, der den Status *ausgewählt* hat und sich optisch in der Darstellung von anderen Klassenknoten unterscheidet (vgl. die Beschreibung von Klassenknoten in Abschnitt 4.2.2.1).

ItemClass ist ein Beispiel für ein *redefiniertes* Attribut, das in der Superklasse *ClassGraph* nur einen Nullwert enthält. Sein Wert ist hier auf *ClassCircleItem* gesetzt und spezifiziert somit die Komponentenart *Klassenknoten* der vom Diagramm zu verwaltenden Komponenten.

Das Referenzkanten-Teildiagramm enthält die Referenzkanten als Komponenten und ist durch die Klasse *ClassReferencePanel* definiert. Es folgt die Beschreibung relevanter Attribute und Methoden.

getItemUnderPoint hat als Eingabeparameter eine Bildschirmposition und referenziert die dort befindliche Referenzkante. Diese Methode redefiniert die gleichnamige aus *ClassGraph* geerbte, weil für das „Finden“ von Referenzkanten ein spezielles Suchverfahren[16] erforderlich ist.

getClientRefKinds liest die Kardinalitätswerte aller Referenzkanten eines Diagramms, indem sie für jede Refeenzkante das zugeordnete Instanzattribut aufsucht. Die gelesenen Werte werden in einem *Array* abgelegt. Diese Methode dient (neben anderen) zur Datenextraktion für Speicher- und Transformationszwecke (siehe dazu Abschnitt 5.1). Die Diagrammart wird, analog zum Vorgehen bei der Definition der Klasse *CirclePanel*, im Attribut *ItemClass* spezifiziert. Der dem Attribut zugeordnete Klassenname *ClassArrowItem* definiert die Referenzkanten als Komponenten des Diagramms.

Das Werteknoten-Teildiagramm enthält die Werteknoten mit den Werteattributen eines OM1 Referenzdiagramms. Es wird durch die Klasse *ClassValuePanel* definiert. Wie in Abschnitt 2.3.2 beschrieben, gibt es drei Ebenen, in denen die Werteknoten eines Diagramms graduell unterschiedlich visualisiert werden. In der ersten Ebene sind die Werteattribute nicht sichtbar. In der zweiten Ebene werden rautenförmige Symbole statt der Werteknoten angezeigt. Diese sind ebenfalls im Teildiagramm als Komponenten enthalten. Die dritte Ebene zeigt alle Werteknoten

[16] Das Suchverfahren berücksichtigt den Umstand, daß die Referenzkanten teilweise auf übereinander geschachtelten Untergrundflächen liegen, die nacheinander durchsucht werden müssen.

an. Für jede Ebenenanzeige ist eine Methode verantwortlich, die den Anzeigestatus (*Showstatus*) aller Komponenten des Teildiagramms setzt, der deren Ein- oder Ausblenden bewirkt (vgl. Abschnitt 4.2.2.3). Die Methoden *setAllVisible*, *setAllHidden* und *setAllInvisible* setzen den jeweiligen Anzeigestatus der verschiedenen Symbolarten für die jeweilige Ebenen. Die Methode *Paint* berücksichtigt beim Malen des Diagramms die verschiedenen Anzeigestati.

Das Gesamtdiagramm ist eine Behälterfläche, die die vorher beschriebenen Teildiagramme sowie die in Abschnitt 4.2.4 beschriebenen *Werkzeugflächen* als Klienten enthält. Es wird durch die Klasse *ClassGraphPanel* definiert. *Set*-Methoden dienen dazu, erzeugte Instanzen von Teildiagrammklassen untereinander (und den Instanzen der Werkzeugklassen) „bekanntzumachen". Es werden spezifische Attribute gesetzt, über die vorhandene Instanzen referenziert werden können. Vor allem für die *Werkzeuginstanzen* ist es relevant, die durch sie bearbeiteten *Materialinstanzen* zu kennen. Die Methode *getClientsBbox*, die von der *Aspektklasse ClassPSPrint* geerbt wird (siehe Abschnitt 4.2.3.3), dient zur Berechnung der Gesamtgröße aus den Teildiagrammen. Sie wird beim Drucken von Diagrammen (vgl. Abschnitt 4.2.3.3) verwendet.

Die Teilflächen des Vererbungsdiagramms werden im folgenden kurz beschrieben. Das *Klassenknoten-Teildiagramm* unterscheidet sich nur durch andere Rastergrößen von dem des *Referenzdiagramms*. Seine definierende Klasse ist *ClassInheritCirclePanel*. Das *Vererbungsteildiagramm* enthält die Vererbungspfeile als Komponenten und ist in der Klasse *ClassInheritArrowPanel* definiert.

4.2.3.2 Generalisierung und Spezialisierung

Durch *Generalisierung* werden Gemeinsamkeiten der Teildiagramme in neuen abstrakten Superklassen definiert. Die Gemeinsamkeiten können jeweils auch als *generische Diagrammfunktionalität* bezeichnet werden.

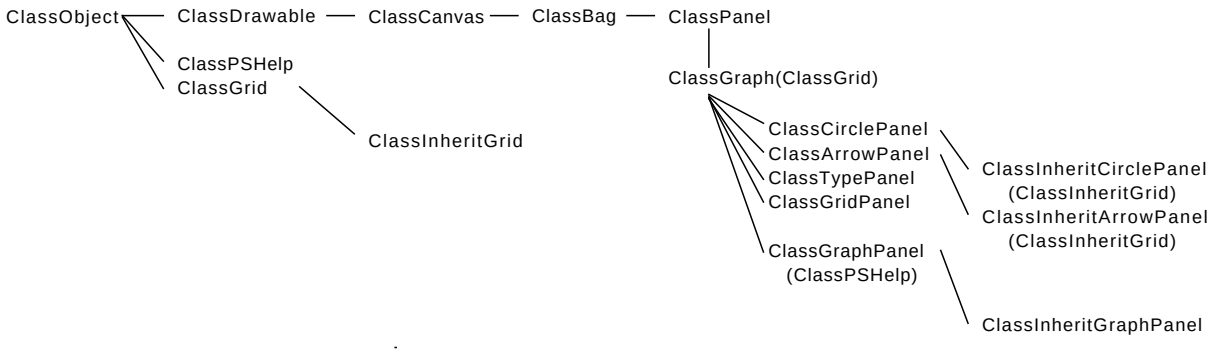


Abbildung 4.16: Hierarchieausschnitt: TNT Subklassen zur Definition von Teildiagrammen

Die Abb. 4.16 zeigt die neu eingeführten Hierarchieebenen. Nach der Erläuterung der generischen Komponentenverwaltung in der Klasse *ClassGraph* und der für die Rasterung zuständigen generischen Klasse *ClassGrid* folgt eine kurze Erörterung der Vorteile der Bildung von Subhierarchien.

Generische Komponentenverwaltung ist durch *generische Attribute und Methoden* der Klasse *ClassGraph* definiert, die jedes (Teil-)Diagramm, welches *eine Art von Basissymbolen* als

Komponenten besitzt, für die Verwaltung benötigt. Die Funktionalität von relevanten Attribute und Methoden ist:

- Instanziierung der Komponentenart durch Setzen des Attributs *ItemClass*;
- Auswahl einer Komponente unter einer (beispielsweise interaktiv selektierten) Position durch die Methode *getItem*;
- boolesche Abfragen der Position bezüglich ihres Ortes (innerhalb/außerhalb eines Diagramms) oder bezüglich ihrer Vakanz (mit einer Diagrammkomponente besetzt oder frei); sie werden beispielsweise von *Einfügewerkzeugen* beim Einfügen von neuen Komponenten getätigt;
- Auflistung aller im Diagramm enthaltenen Komponenten durch die Methode *getItemList*. Wird beispielsweise diese Methode auf das Teildiagramm, welches die Kreisknoten enthält, angewendet, so gibt sie ein *Array* zurück, welches alle Klassenknoten enthält.

Rasterung von Diagrammkomponenten wird durch die Klasse *ClassGrid* definiert. Dort sind Rastergrößen spezifiziert, die neu gesetzt werden können. Weitere Methoden dienen

- zur Rundung von Koordinaten zum nächstgelegenen Rasterpunkt im Diagramm;
- zum Ein- und Ausschalten des *erweiterten Rastermodus* (vgl. Abschnitt 3.3.2);
- zur Umrechnung von Koordinaten auf Zeilen-/Spaltenindizes, die zur Ermittlung benachbarter Diagrammpositionen verwendet werden;
- zum Setzen von Rastergrößen (in x/y-Werten).

Diese geerbte Funktionalität wird unter anderem zum Skalieren der Rastergrößen (vgl. Abschnitt 3.3.2) und zum punktgenauen Positionieren der jeweiligen Komponenten (z.B. der Klassenknoten) auf gültige Rasterpositionen in den Diagrammen benutzt.

Die Vorteile der Bildung neuer Hierarchieebenen in TNT liegen in der flexiblen Nutzung von Vererbung. Es ist hervorzuheben, daß *generische, datenmodellunabhängige Konzepte* auch für die Implementation anderer Datenmodelle, z.B. des *Entity Relationship* Modells (vgl. Abschnitt 2.1) genutzt werden können. Ferner stellt die hier als *Subklassengenerierung* bezeichnete Anwendung multipler Vererbung eine vereinfachte Vorgehensweise für die Definition neuer Subklassen bereit. Dies belegt anschaulich folgendes Kodebeispiel einer Klassendefinition.

```
/ClassInheritCirclePanel [ClassInheritGrid ClassCirclePanel] []
    classbegin
classend def
```

Im Beispiel definiert die Klasse *ClassInheritCirclePanel* die Teilfläche eines *Vererbungsdiagramms*, die die Klassenknoten enthält. Die Klasse erbt alle Eigenschaften der Klassen *ClassCirclePanel* und *ClassInheritGrid*. Allein durch diese Kombination der geerbten Eigenschaften wird diese Klasse vollständig definiert. Der Kodeumfang der Klassendefinition wird auf ein Minimum reduziert.

4.2.3.3 Kategorisierung der Klassen nach dem Leitbild der Werkzeug-Material Metapher

Diagramme sind als *Material* zu kategorisieren. Das in Abschnitt 4.2.1 eingeführte Leitbild beinhaltet neben Werkzeug- und Materialklassen auch *Aspektklassen*. Eine Aspektklasse faßt bestimmte Eigenschaften mehrerer Materialklassen zusammen, die für ein bestimmtes Werkzeug relevant sind. Beisielsweise faßt der Aspekt *Druckbares Diagramm* alle Eigenschaften zusammen, die für das *Werkzeug Drucker*[17] relevant sind. Die das Drucken betreffenden Eigenschaften sind für das Referenzdiagramm wie für das Vererbungsdiagramm (bzw. deren Teildiagramme) in der Aspektklasse *ClassPSPrint* zusammengefaßt. Wesentliche Funktionen dieser Klasse bestehen in:

- der Berechnung der für den Drucker relevanten Dimension des Dokuments (*bounding Box*) aus den Koordinaten der Diagrammkomponenten;
- der Kombination verschieden dimensionierter Teildiagramme;
- den Methoden zur eigentlichen Erzeugung eines druckbaren Dokuments.

Der letzte Punkt wird verdeutlicht. In der Superklasse *ClassCapture*, die zur TNT Klassenbibliothek gehört, sind Methoden zur *Transformation* von NeWS Klassenmethoden, die die Bildschirmausgabe von NeWS Instanzen spezifizieren, in PostScript-Seitenbeschreibungen vorhanden. Durch Anwendung dieser Transformationsfunktionalität auf Instanzen von angezeigten Bildschirmobjekten werden druckbare PostScript Daten erzeugt. Eine weitergehende Funktionalität, um – mit Hilfe von TL Funktionen – Dateien in EPSF-Format[18] zu erzeugen, wird in Abschnitt 4.4.1.2 ausführlich beschrieben.

4.2.4 Werkzeugklassen für die Bearbeitung von Diagrammen

Im Gegensatz zu den in den vorigen Abschnitten behandelten Materialklassen spielen bei der Realisierung der Werkzeugklassen für die Diagrammerzeugung und -manipulation die in der Einleitung von Abschnitt 4.2 eingeführten *Verfahrensweisen* eine untergeordnete Rolle. Dekomposition wird nicht angewendet. Sie bezieht sich nur auf sichtbare Bildschirmobjekte, die in diesem Abschnitt definierten Werkzeuge sind *unsichtbar*. *Generalisierung* von Werkzeugeigenschaften war nur in beschränktem Maße möglich. Das liegt an der Unterschiedlichkeit der zu vollziehenden Bearbeitungsschritte bei der *Manipulation verschiedenartiger Basissymbole*[19]. Die Kategorisierung nach der *Werkzeug-Material-Metapher* spielt eine Rolle bei der Definition von Methoden in Werkzeugklassen. Bei der Diagrammbearbeitung verwenden die Werkzeugmethoden Methodenaufrufe von *Materialklassen*. Das Vorgehen beim Entwurf geeigneter Klassenkonzepte richtet sich vielmehr nach der erwünschten *unterschiedlichen Funktionalität* der einzelnen Softwarewerkzeuge.

Im Abschnitt 4.2.4.1 wird die Funktionalität des Graphikeditors in Bezug auf Diagrammbearbeitung beschrieben. Die dazu entwickelte Teilhierarchie neuer TNT-Klassen wird kurz vorgestellt. Der Abschnitt 4.2.4.2 geht auf die Realisierung genereller Funktionalität zur *benutzergesteuerten Auswahl von Bildschirmobjekten*, wie sie von allen Werkzeugen benötigt wird, ein. Im

[17] Der *Drucker* kann auch ein Softwarewerkzeug sein, welches beispielsweise eine druckbare PostScriptDatei erzeugt und an einen „realen“ Drucker schickt.

[18] EPSF heißt: *Extended PostScript Format* – vgl. [AS91]

[19] Beispielsweise muß beim Einfügen eines Klassenknotens ein leerer Platz im Diagramm selektiert werden, beim Einfügen einer Referenzkante müssen zwei existierende Klassenknoten vom Benutzer ausgewählt werden.

den Abschnitten 4.2.4.3 bis 4.2.4.6 werden die Realisierungen einzelner Werkzeuge durch neue TNT Subklassen genauer beschrieben. Es wird beispielhaft das Zusammenspiel von Werkzeug- und Materialklassen anhand von Methodenaufrufen gezeigt, die durch Folgen von interaktiven Benutzereingaben ausgelöst werden. Der Abschnitt 4.2.4.7 gibt eine Zusammenfassung der den vorgestellten Werkzeugklassen gemeinsamen Charakteristik.

4.2.4.1 Funktionalität der Softwarewerkzeuge

Wie in Abschnitt 3.3 bereits erwähnt, umfaßt die Diagrammbearbeitung das Erzeugen und Löschen von Klassenknoten, Referenzkanten, Vererbungspfeilen und Werteattributen. Eine wichtige Rolle spielt die Funktionalität zur interaktiven Auswahl von Bildschirmobjekten durch den Benutzer. Im folgenden werden die möglichen *Bearbeitungstätigkeiten*, die in Abschnitt 3.3.2.4 ausführlich spezifiziert sind, mit jeweiligen Benutzeraktivitäten aufgelistet. Der verwendete Begriff *Auswahl* bedeutet, daß ein Bildschirmobjekt mit der linken Maustaste angeklickt wird, das *Eintragen* von Texten ist eine interaktive Benutzereingabe über die Tastatur.

Das Einfügen von Klassenknoten erfolgt durch Auswahl eines leeren Rasterpunktes und Eintragen des Klassennamens.

Das Einfügen von Referenzkanten erfolgt durch Auswahl des referenzierenden (Start-) Klassenknotens und des zu referenzierenden (Ziel-)Klassenknotens. Der Attributname und die Kardinalität werden separat in einem Aufklappfenster eingetragen.

Das Einfügen von Vererbungspfeilen erfolgt durch Auswahl des die *erbende Klasse* repräsentierenden (Start-)Klassenknotens und des die *vererbende Klasse* repräsentierenden (Ziel-)Klassenknotens.

Das Einfügen von Werteattributen erfolgt durch Auswahl eines Klassenknotens. Der Attributname und der Typ werden in einem Aufklappfenster eingetragen.

Das Löschen von Basissymbolen[20] geschieht durch jeweilige Auswahl des zu löschenden Objekts.

In Abb. 4.17 sind die Klassendefinitionen für die Einfüge- und Löschwerkzeuge und deren Stellung in der Vererbungshierarchie dargestellt. Die Klassen ClassInsertCircles/-Arrows/-Values/--Inheritances definieren die Einfügewerkzeuge für Klassenknoten, Referenzkanten, Werteattribute und Vererbungspfeile, die Klassen ClassDeleteClasses/-Arrows etc. definieren die jeweiligen Löschwerkzeuge.

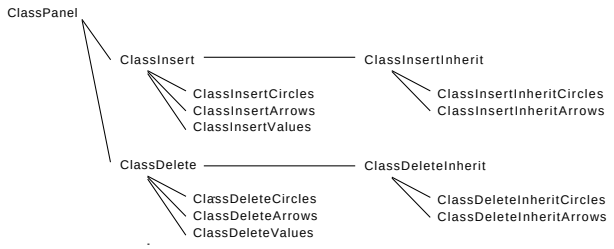


Abbildung 4.17: Hierarchieausschnitt: TNT Subklassen zur Definition von Diagrammwerkzeugen

Diese Werkzeugklassen werden zur besseren Unterscheidung *Diagrammwerkzeugklassen* genannt, die durch sie realisierten Softwarewerkzeuge heißen *Diagrammwerkzeuge*. *ClassUpdate* und *ClassDelete* sind generelle Werkzeugklassen, die gemeinsam nutzbare Methoden ihrer Subklassen (der Diagrammwerkzeugklassen) zusammenfassen.

4.2.4.2 Realisierung genereller Funktionalität zur interaktiven Auswahl von Basissymbolen

Unter Einbeziehung der *Werkzeug-Material* Metapher sowie Nutzung von in TNT definierten Methoden, die von Maustasten ausgelöste Ereignisse auswerten können, werden für die Diagrammwerkzeuge *gemeinsame Charakteristika* festgelegt, die ihre äußere Form und die Realisierung ihrer Funktionalität betreffen.

Die Form eines Diagrammwerkzeugs ist durch eine auf dem Bildschirm unsichtbare *Werkzeugfläche* gegeben, die als *Interaktionsfläche* für Benutzereingaben fungiert und dieselbe Größe wie das Diagramm hat, welches bearbeitet werden soll.

Die Funktionalität eines Diagrammwerkzeugs ist durch Methoden realisiert, die im folgenden näher erklärt werden. Für die *Ereignisverarbeitung* gibt es in der Klasse *ClassCanvas* vordefinierte *Ereignismethoden*, die nach Eintritt von durch Maustasten ausgelösten Ereignissen (Mausereignissen) mit Ereignisdaten als Eingabeparameter aufgerufen werden. Jede Subklasse von *ClassCanvas*, die in der Lage sein soll, Mausereignisse zu verarbeiten, signalisiert dies durch das Setzen des booleschen Attributs *Trackable* auf den Wert *True*. Die eigentliche Funktionalität der Ereignismethoden muß aber in jeder Subklasse von *ClassCanvas* realisiert werden. Dieser Mechanismus der Implementation von Methoden in Subklassen, die bereits in Superklassen spezifiziert sind, wird in [SM92b] als *Strict Subclass Responsibility* bezeichnet.

Die wichtigsten Ereignismethoden sind *TrackStart*, die beim Drücken der linken oder mittleren Maustaste aufgerufen wird, und *TrackStop*, die beim Loslassen einer Maustaste aufgerufen wird. Beim Methodenaufruf werden die Position und die Maustastenart als Eingabeparameter angegeben. Die jeweilige Subklasse implementiert weiteren Kode in den Ereignismethoden zur Auswertung der Positionen und zur Realisierung werkzeugspezifischer Reaktion.

Mithilfe des Positionierens des Werkzeugs auf dem Bildschirm und dem servergesteuerten Aufruf von Ereignismethoden wird die Auswahl eines Basissymbols durch folgende *Abfolge von Operationen*[21] realisiert.

1. Positionieren des Werkzeugs:
   Die Interaktionsfläche eines vom Benutzer gewählten Werkzeugs wird auf dem Bildschirm deckungsgleich vor das Diagramm gelegt. Da sie unsichtbar ist, wird das Diagramm weiterhin angezeigt.

2. Warten auf das Mausereignis:
   Die Interaktionsfläche des Werkzeugs ist nun für Eingaben des Benutzers empfangsbereit, da sie als *vorn liegende Fläche* das erste Bildschirmobjekt ist, das ein durch ein Mausklick ausgelöstes Ereignis empfängt.

[21] Die im folgenden verwendeten relativen Positionsangaben wie *vorn* etc. beziehen sich auf die imaginäre *Z-Ebene* der dreidimensionalen Einteilung des Bildschirms.

3. Auswertung des Ereignisses:
   Ein vom Benutzer ausgelöstes Mausereignis enthält unter anderem die Koordinaten des Ereignisses, die in den Ereignismethoden des Werkzeugs (*TrackStart*/*TrackStop*) ausgewertet wird. In diesen Methoden wird geprüft, ob im betreffenden (Teil-)Diagramm auf der Ereigniskoordinate ein Basissymbol liegt. Im positiven Falle bedeutet das, daß der Benutzer dieses Basissymbol mit der Maus ausgewählt hat.

4. Ausführen von werkzeugspezifischen Aktionen:
   Weitere werkzeugtypische Aktionen, wie z.B. Löschen des Basissymbols, wenn es sich um ein Löschwerkzeug handelt, folgen jeweils.

Die hier eher generelle Beschreibung der Operationenfolge zur Steuerung von Interaktion wird bei der Vorstellung der spezifischen Diagrammwerkzeuge in den folgenden Abschnitten anhand von Beispielen konkretisiert. Der jeweilige Ablauf wird zunächst exemplarisch in einem Szenario beschrieben. Danach wird die Realisierung des Ablaufs erklärt, um die dahinter stehende Methodik zu demonstrieren. Es soll gezeigt werden,

- welche Schritte von der Zustandsabfrage des Diagramms bis zur Manipulation des Diagramms auf Serverseite nötig sind;

- wie die Schritte durch Methodenaufrufe der Werkzeugklassen, die zu Methodenaufrufen der Materialklassen führen, ausgeführt werden;

- wie sich die Topologie und das Aussehen der Bildschirmobjekte bei jedem Schritt ändert; dies wird durch Graphiken illustriert.

4.2.4.3 Das Einfügewerkzeug für Klassenknoten

Die definierende Klasse ist *ClassInsertCircles*. Die von ihr erzeugte Instanz, die als Bildschirmobjekt die Werkzeugfläche darstellt, wird der Einfachheit halber im folgenden *InsertCircles* genannt.

Beschreibung des Szenarios: Der Benutzer wählt im Diagramm einen Rasterpunkt, auf dem noch kein Klassenknoten ist. Es erscheint eine Textzeile, in der interaktiv der Klassenname eingetragen wird. Der neue Klassenknoten wird auf dem Rasterpunkt mit dem Klassennamen als Label angezeigt.

Es folgt die Auflistung der aus dem Szenario resultierenden *Abfolge von Aktivitäten*, die sich aus Benutzeraktivitäten (**Benutzer**) und reaktiven Systemaktivitäten (**System**) zusammensetzt. Die für die Realisierung der Systemaktivitäten relevanten Klassenmethoden werden dabei jeweils erläutert. Die Abbildung 4.18 skizziert die Schritte.

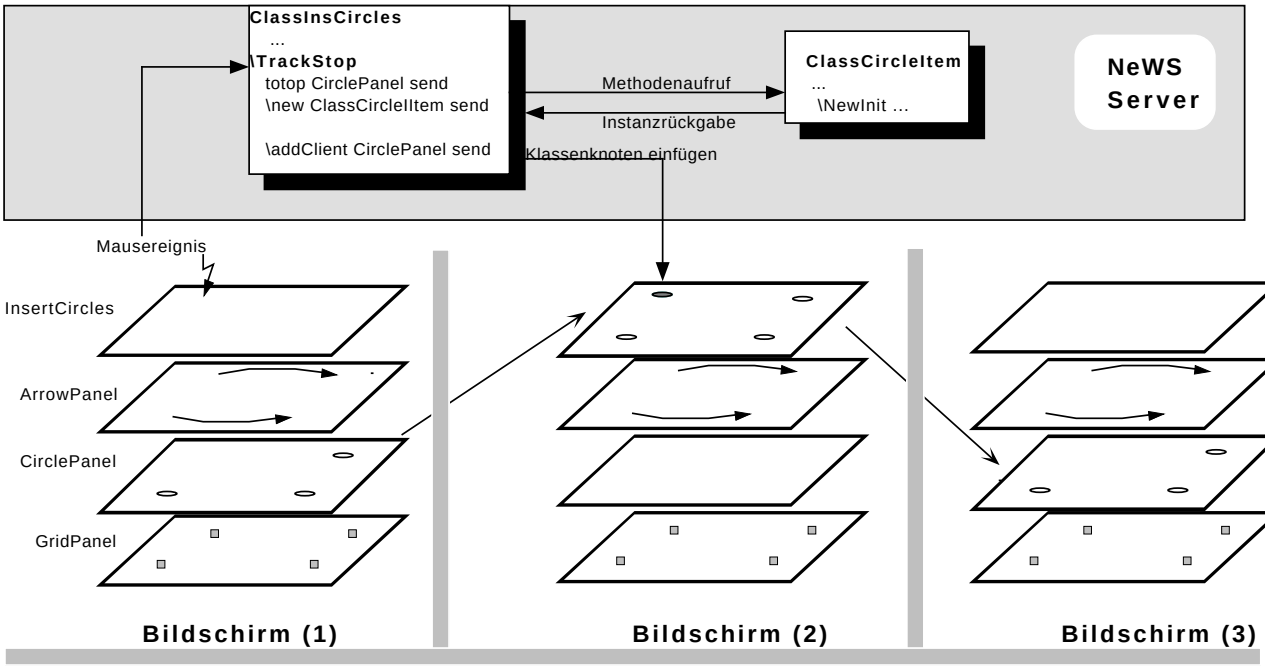


Abbildung 4.18: Szenario: Systemaktivitäten beim interaktiven Einfügen eines Klassenknotens

Benutzer: Auswahl des Einfügewerkzeugs. Dies geschieht über Editormenüs, auf die hier nicht näher eingegangen wird (vgl. Abschnitt 3.3.2).

System: Einblendung einer Rasterpunktfläche (*GridPanel*), die die Rasterpunkte anzeigt, und Einblendung und Positionierung der Werkzeugfläche (*InsertCircles*) als vorderste, diagrammdeckende Fläche auf dem Bildschirm. Die Topologie der Flächen – inklusive der das Diagramm bildenden zwei Teildiagrammflächen für Klassenknoten und Referenzkanten (*CirclePanel* und *ArrowPanel*) ist in Abb. 4.18, links illustriert. Das Werkzeug ist jetzt empfangsbereit für ein Mausereignis.

Benutzer: Selektieren eines leeren Rasterpunkts im Diagramm mit der linken Maustaste.

System: Die Werkzeugfläche (*InsertCircles*) empfängt das Mausereignis (siehe Abb. 4.18, links). Der NeWS Server ruft die Methode *TrackStop* der das Einfügewerkzeug definierenden Klasse *ClassInsertCircles* auf. Von dort werden mehrere Operationen durch Methodenaufrufe ausgeführt, deren Funktionalität kurz aufgelistet ist.

1. Runden der Mauskoordinaten auf die exakte Rasterpunktposition;
2. Abfrage, ob das Ereignis innerhalb des Diagramms ist;
3. Nachvornelegen der Klassenknotenfläche und Abfrage, ob die gewählte Position *unbesetzt* ist. Wenn ja, wird ein neuer Klassenknoten – als Instanz der Klasse *ClassCircleItem* mit leerem Label und Texteingabefeld erzeugt und in das Klassenknotendiagramm *CirclePanel* auf der gewählten Position eingefügt (vgl. Abb. 4.18, mitte). Es wird das Texteingabefeld für Mauseingaben *aktiviert*, alle anderen Flächen auf dem Bildschirm sind *deaktiviert* und für Eingaben nicht empfänglich.

Benutzer: Eingabe des Klassenknotennamens über Tastatur in das Texteingabefeld. Die Eingabe wird durch die *Return*-Taste abgeschlossen. (Dieser Vorgang wird in der Abb. 4.18 nicht dargestellt.)

System: Durch die *Return*-Taste wird in *ClassCircleItem* eine Benachrichtigungsmethode[22] (*notifier*) ausgeführt. Sie setzt den Namen in das Label, löscht das Texteingabefeld, zeigt den (nun vollständigen) Klassenknoten mit Label an und legt die Werkzeugfläche nach vorn (siehe Abb. 4.18, rechts). Das System ist jetzt bereit für die nächste Maustasteneingabe des Benutzers.

Die hier vorgestellte Funktionalität bezieht sich zwar auf das *Referenzdiagramm*, wird aber genauso im *Vererbungsdiagramm* angewendet.

4.2.4.4 Das Einfügewerkzeug für Referenzkanten

Die definierende Klasse ist *ClassInsertArrows*. Die von ihr erzeugte Instanz, die als Bildschirmobjekt die Werkzeugfläche darstellt, wird der Einfachheit halber im folgenden *InsertArrows* genannt.

Beschreibung des Szenarios: Der Benutzer wählt im Diagramm den Startklassenknoten und den Zielklassenknoten für die Referenzkante. Die neue Referenzkante wird eingefügt.

Die entsprechenden Systemaktivitäten erfolgen analog zu den im vorigen Abschnitt – für das Einfügen von Klassenknoten – genannten. Der Unterschied liegt darin, daß nach Auswahl der Start- und Zielpositionen auf der *Klassenknotenfläche* die *Referenzkantenfläche* nach oben auf den Bildschirm gelegt wird, um zu prüfen, ob die spezifizierte Position zum Einfügen einer neuen Referenzkante frei ist. Ist dies der Fall, so wird eine neue Referenzkante eingefügt. Auf die Darstellung der Systemaktivitäten im Zusammenhang mit dem Aufklappfenster wird hier aus Gründen der Übersichtlichkeit verzichtet.

4.2.4.5 Weitere Einfügewerkzeuge

Es folgt die Beschreibung zweier weiterer Werkzeuge, wobei jeweils nur die Unterschiede zu den vorher beschriebenen genannt werden.

Das Einfügewerkzeug für Werteattribute im Referenzdiagramm ist in der Klasse *ClassInsertValues* definiert. Die Systemaktivitäten zur Auswahl eines Klassenknotens und anschließendem Nachvornlegen der Werteattributfläche sind analog zu denen in den vorher beschriebenen Abschnitten. Wenn ein neues Werteattribut eingefügt werden soll, wird geprüft, ob bereits ein Werteknoten für den gewählten Klassenknoten vorhanden ist. Ist dies der Fall, so wird ein neues Werteattribut – durch die Methode *NewInit* der Klasse *ClassValueItem* – erzeugt und anschließend die Methode *resize* der Klasse *ClassValueBag* aufgerufen, um den Werteknoten und dessen Komponenten optisch zu vergrößern. Wenn noch kein Werteknoten vorhanden ist, wird dieser mit seinen Komponenten vor Ausführung der vorher beschriebenen Schritte zunächst erzeugt, indem transitiv die *NewInit* Methoden der Komponentenklassen des Werteknotens aufgerufen werden (vgl. die in Abschnitt 4.2.2.3 beschriebene Zusammensetzung von Werteknoten).

Das Einfügewerkzeug für Vererbungspfeile im Vererbungsdiagramm ist in der Klasse *ClassInsertInheritArrows* definiert. Die die Auswahl der Klassenknoten betreffenden System-

[22] Benachrichtigungen werden näher in Abschnitt 4.1.3.2 erklärt.

und Benutzeraktivitäten sind analog zu denen für das Einfügen von Referenzkanten im Referenzdiagramm (vgl. Abschnitt 4.2.4.5). Vor dem Einfügen berechnen Methoden der Werkzeugklasse (*ClassInsertInheritArrows*) aus den Start- und Zielpositionen *Länge* und *Neigungswinkel* des neuen Vererbungspfeils, der dann erzeugt wird, wobei die Länge, der Winkel und die Startposition als Eingabeparameter verwendet werden (vgl. Abschnitt 4.2.2.4). Danach wird er in das Vererbungspfeildiagramm eingefügt.

4.2.4.6 Löschwerkzeuge

Die verschiedenen Löschwerkzeuge für Klassenknoten, Referenzkanten und Attributwerte arbeiten alle nach demselben Verfahren, welches hier anhand des Beispiels des Löschens von Klassenknoten skizziert wird.

- **Initialisierung:** Das Löschwerkzeug wird auf dem Bildschirm nach vorn gelegt und wartet auf ein Mausereignis.
- **Auswahl des zu löschenden Basissymbols:** Nach dem Mausereignis wird das Klassenknotendiagramm vorübergehend nach vorn auf den Bildschirm gelegt. Es wird der Klassenknoten ermittelt, der dort auf der durch das Mausereignis spezifizierten Position liegt. Der Klassenknoten wird gelöscht und das Diagramm wird neu angezeigt.
- **Anzeige** des geänderten Diagramms und Nachvornlegen des Löschwerkzeugs auf der Bildschirmfläche.

4.2.4.7 Zusammenfassung der Methodik von Werkzeugen

In diesem Abschnitt wird die gemeinsame Methodik der vorher beschriebenen Werkzeuge zur integrativen Bearbeitung verschiedener Teildiagramme vorgestellt. Zusammenhänge zwischen Objekten einzelner Teildiagramme werden über ihre (auf Rasterpunkte gerundeten) *Positionen* hergestellt. Beispielsweise wird beim Einfügen von Werteattributen wird zunächst im *Klassenknotendiagramm* durch die interaktive Auswahl ein Klassenknoten spezifiziert, dessen Position erfragt wird. Dann wird im *Werteknotendiagramm* unter derselben Position ein vorhandener Werteknoten gesucht und angezeigt. Die Art der Werkzeugrealisierung durch Flächen bietet folgende Vorteile.

- Das Aktivieren einzelner Diagrammwerkzeuge ist leicht steuerbar. Das gerade gewünschte Werkzeug wird durch *einen* Methodenaufruf – der in *ClassCanvas* definierten Methode *totop* – nach oben auf den Bildschirm gelegt und ist damit als einziges aktiv.
- Die Anzeige eines Diagramms wird bei der Bearbeitung durch ein Werkzeug nicht verdeckt oder ungewollt geändert.

4.3 Erweiterung der Tycoon Bibliothek newsenv

Im vorigen Abschnitt wurden neue NeWS Klassen vorgestellt, die die Funktionalität zur interaktiven Erzeugung und Manipulation von OM1 Diagrammen auf dem Bildschirm bereitstellen. Um diese Funktionalität in die *Tycoon* Umgebung einzubinden, ist eine Erweiterung der Bibliothek *newsenv* erforderlich. In diesem Abschnitt ist die Realisierung der die Erweiterung realisierenden Module Gegenstand der Betrachtung, wobei es um folgende Aspekte im Vordergrund stehen.

- **Benennung der Funktionalität**, die vom Graphikeditor aufzurufen ist, und Auflistung der sie realisierenden Schnittstellenmodule;

- **Darstellung des Implementationsaufwands der Modulfunktionen** durch Kodebeispiele, die belegen, daß der Implementationsaufwand durch Nutzung von *Basisdiensten* des *newsenv* reduziert wird;

- **Beschreibung von Funktionsaufrufen und der dahinterstehenden Dienste** mit dem Ziel, die einfache Handhabung komplexer Graphikfunktionalität in TL zu demonstrieren sowie das Zusammenspiel von Tycoon und NeWS Server während der Ausführung von TL Funktionen zu darzustellen.

Die in diesem Abschnitt eingeführten Module und deren Funktionen werden zur eigentlichen Realisierung des Editors (vgl. Abschnitt 4.4) verwendet.

Im folgenden Abschnitt 4.3.1 werden die bibliothekserweiternden Module und die durch sie realisierte Funktionalität vorgestellt. Im Abschnitt 4.3.2 wird exemplarisch der Implementationsaufwand bezüglich der Module gezeigt. Der Abschnitt 4.3.3 zeigt anhand eines Anwendungsbeispiels das Zusammenspiel der aufgerufenen Dienste.

4.3.1 Die Module

Die in Abschnitt 3.3.2 ausführlich beschriebene Editorfunktionalität beinhaltet Operationen, wie Laden, Anzeige, Manipulation, Ausdrucken und Sichern von Diagrammen. Die Realisierung dieser Operationen geschieht mit Hilfe von Methodenaufrufen der NeWS Klassen, die in Abschnitt 4.2 eingeführt wurden. Zu folgenden NeWS Klassen der *TNT Erweiterungen* gibt es unter Tycoon jeweils ein korrespondierendes Modul.

- Materialklassen, die die Diagramme (und Teildiagramme) definieren;

- Werkzeugklassen, die die Diagramme manipulieren;

- Zwei weitere Klassen, die die Rastergößen definieren und das Rollen (*scrolling*) von Diagrammausschnitten in Fenstern ermöglichen.

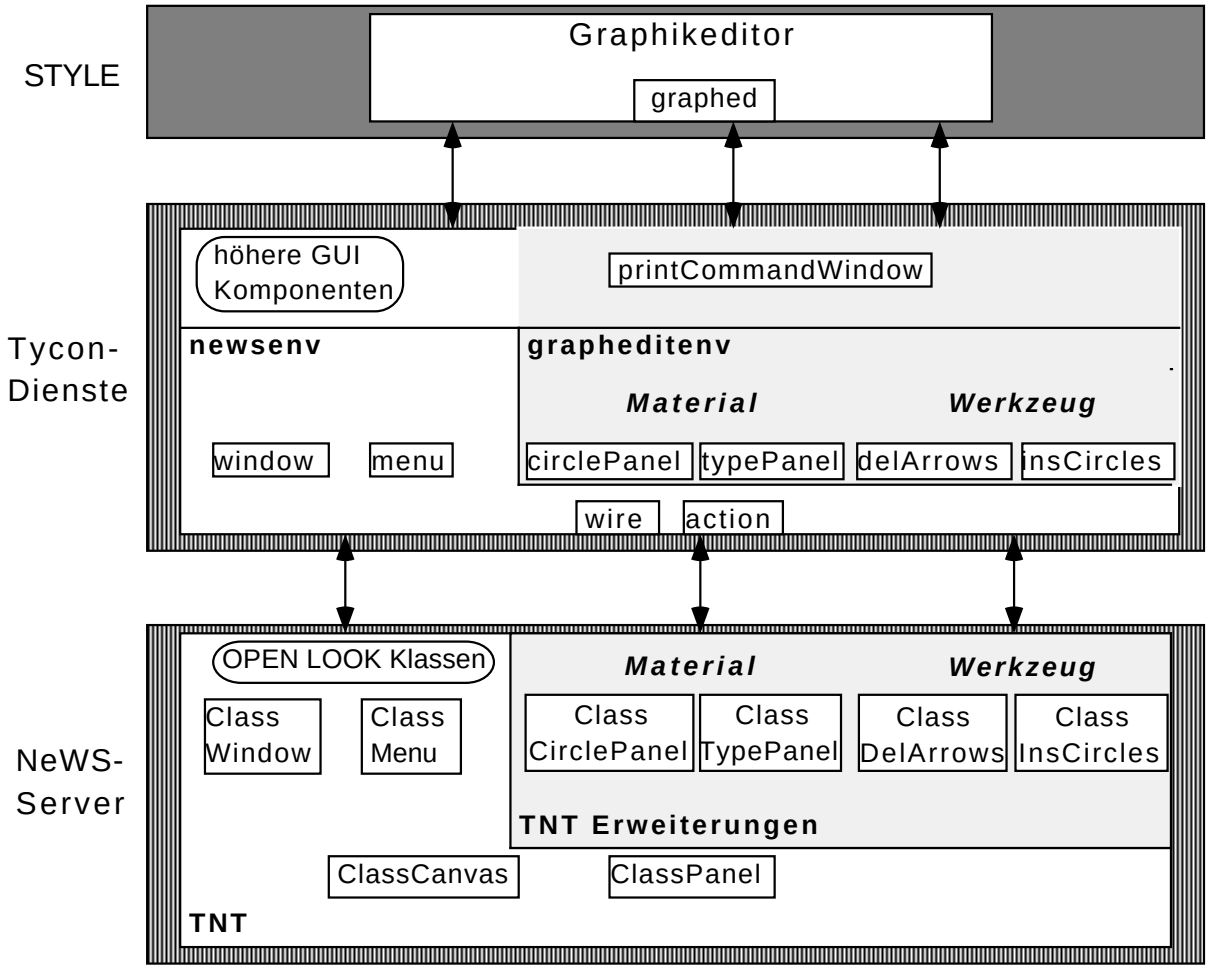


Abbildung 4.19: Erweiterungen der Dienste zur Einbindung von Graphikfunktionalität in TL

Jedes Modul trägt den Namen der entsprechenden NeWS Klasse, jedoch ohne das Präfix *Class*. Die Erweiterung der Bibliothek *newsenv* und die Anbindung von Klassen des erweiterten Werkzeuges TNT ist in Abbildung 4.19 ausschnitthaft dargestellt.[23] In jedem Modul sind jeweils die zentralen Methoden einer Klasse durch TL Funktionen implementiert.
Diejenigen Module, die die *Materialklassen* in Tycoon einbinden, beinhalten neben Erzeugungsfunktionen jeweils Funktionen zur Anzeige von Teildiagrammen sowie zur Extraktion von Diagrammdaten. Die Funktionen werden beispielhaft am Modul *circlePanel* erläutert. Die Funktion

createCalculated(parent :Canvas) :CirclePanel.T

dient zur Erzeugung einer neuen Klassenknotenfläche des Typs *CirclePanel.T*. Der Namensteil *Calculated* weist darauf hin, daß es sich um eine Subklasse von *ClassPanel* handelt, die weitere, „kalkuliert“ angeordnete Teilflächen (die Klassenknoten), enthält. Die drei Funktionen

getLabels(pan :CirclePanel.T) :Iter.T(String)
getXValues(pan :CirclePanel.T) :Iter.T(Int)
getYValues(pan :CirclePanel.T) :Iter.T(Int)

[23] Die hellgrau unterlegten Regionen der Abbildung kennzeichnen die Diensterweiterungen. Der Name *grapheditenv* benennt die Erweiterung der Bibliothek *newsenv*.

dienen zur *Datenextraktion* aus dem Diagramm, die für die Diagrammsicherung und Datentransformation (vgl. Abschnitt 5.1) benötigt wird. Die Rückgabewerte der Funktionen sind Iterationen über die Namen und die x/y-Koordinaten aller Klassenknoten des Diagramms.

Die zu den *Werkzeugklassen* korrespondierenden Module beinhalten Funktionen zur Erzeugung der *Werkzeugflächen*. Diejenigen Module, die Einfügewerkzeuge implementieren, enthalten Einfügefunktionen, die für den Diagrammaufbau beim Starten des Editors verwendet werden. Es werden Funktionen des Moduls *insertCircles*, zum Einfügen von Klassenknoten, vorgestellt.

createCalculated(parent :Canvas) :InsertCircles.T
insert(insCirc :InsCircles.T x,y :Int label, labPos :String) **:Ok**

Die erste Funktion erzeugt ein neues Einfügewerkzeug für Klassenknoten. Die zweite wird verwendet, um einen neuen Klassenknoten, dessen Name und Position durch die Parameter *label* und *x,y* spezifiziert sind, zu erzeugen und ihn über das im Parameter *insCirc* instanziierte Einfügewerkzeug in eine Klassenknotenfläche einzufügen.

4.3.2 Implementationsbeispiele von Funktionen

Es folgt ein Kodebeispiel für die Implementation der Funktion *insertCircles.insert*.

```
let insert(insCirc :T  x,y :Int  label, labPos :String) :Ok =
  begin
    wire.writeInt(x)
    wire.writeInt(y)
    wire.writeF("{%s}" labPos)
    wire.writeF("{%s}" label)
    object.send(insCirc "/insert")
  end
```

Unter Nutzung von Basisdiensten des Moduls *wire* wird eine Art „Standleitung“ zur Kommunikation zwischen Tycoon und NeWS Server hergestellt. Im Beispiel werden zunächst über getypte Schreibfunktionen die Eingabeparameter für die Methode *insert* an den Server verschickt.[24] Anschließend wird über den Aufruf der im *newsenv* Modul *object* implementierten Funktion *object.send* bewirkt, daß auf Serverseite an das NeWS Objekt *insCirc* (das ist der Name einer Instanz der Klasse *InsertCircles*) der Aufruf der Methode *insert* gesendet wird. Der Funktionsaufruf

insertCircles.insert(insCirc 190 390 "Country" "East")

erzeugt auf der Serverseite den Methodenaufruf

190 390 East Country /insert gui20 **send**

Es wird die Methode *insert* an die Instanz *gui20*[25] gesendet, die dann die ersten vier Werte als Eingabeparameter erhält.

Durch Nutzung der Basisdienste *wire* und *action* (vgl. Abschnitt 4.1.4.1 und [Mü94b]) ist der Programmkode der TL Funktionen, die NeWS Methoden aufrufen, einfach zu implementieren. Von Kommunikationsprotokollen kann weitgehend abstrahiert werden.

[24] Beispielsweise schickt *wire.writeInt* eine ganze Zahl und *wire.writeF* eine Zeichenkette über die Leitung.

[25] Dies ist eine Zeichenkette, die bei Initialisierung eines Bildschirmobjekts als eindeutiger Identifikator erzeugt wird.

4.3.3 Ein Ausführungsbeispiel

In dem folgen Szenario wird das Zusammenspiel zwischen Tycoon und dem NeWS Server exemplarisch verdeutlicht. Der Aufruf der TL Funktion *insertCircles.insert* bewirkt die folgenden Ausführungsschritte.

Auf der Tycoonseite werden Funktionen des Moduls *wire* zum Senden der Eingabeparameter und des Methodenaufrufs an das *Einfügewerkzeug insCirc* aufgerufen, die über die Leitung an den NeWS Server geschickt werden.

Auf der Serverseite steht jetzt auf dem Ausführungskellerspeicher (*operand stack*) die Kodesequenz

190 390 East Country /insert gui20 ***send***

Zunächst wird die aktuelle Klassenumgebung[26] der Instanz *gui20* in den Wörterbuchkellerspeicher (*dictionary stack*) geladen, dann wird dort die Methode *insert* der definierenden Klasse *ClassInsertCircles* gesucht und aufgerufen. Bei deren Ausführung werden weitere Methodenaufrufe an folgende Instanzen und Klassen gesendet.

- An die Klassenknotenfläche (die Instanz *CirclePanel*): Boolesche Abfragen über die Korrektheit der Koordinaten (z.B. innerhalb?/ nicht besetzt?);
- an die Klasse *ClassCircleItem*: Initialisieren einer neuen Klassenknoten-Instanz durch die Methode */new ClassCircle send*, die die Koordinaten, das Label und die Labelposition als Eingabeparameter benutzen;
- an die Klassenknotenfläche: Einfügen der neuen Klassenknoten-Instanz in die Klassenknotenfläche und Anzeige der Klassenknotenfläche mit dem neuen Klassenknoten.

Die obige Erläuterung verdeutlicht die Komplexität des Ablaufs im NeWS Server. Von den komplexen Methodenaufrufen der *NeWS* Klassen seitens des Servers wird auf der Tycoonseite abstrahiert. Die Syntax der Aufrufe von TL Funktionen aus Modulen der erweiterten Bibliothek *newsenv* ist entsprechend kurz und einfach.

4.4 Editorrealisierung

Das Thema dieses Abschnitt ist die Implementation des Graphikeditors durch Benutzung der in den vorhergehenden Abschnitten beschriebenen Dienste. Es wird erläutert, wie Bildschirmobjekte des Graphikeditors erzeugt werden und wie Funktionalität, die interaktiv aufrufbar ist, an bestimmte Bedienelemente des Graphikeditors gebunden wird.

Im ersten Abschnitt wird beschrieben, wie die OPEN LOOK Komponenten des Graphikeditors durch Benutzung der Tycoon Bibliothek *newsenv* implementiert werdenv. Im zweiten Abschnitt wird gezeigt, wie durch Benutzung der Erweiterungen des *newsenv* die Diagramme als *anwendungsspezifische* Komponenten und Funktionalität zur Bearbeitung der Diagramme in die OPEN LOOK Komponenten integriert werden. Es werden jeweils Kodebeispiele vorgestellt, die zeigen, daß durch wenige Aufrufe von TL Funktionen der in Abschnitt 4.1.4 und 4.3 Module die Graphikfunktionalität der im Server geladenen NeWS Klassen genutzt wird. Entsprechende Abbildungen illustrieren die Interaktionen zwischen TL Funktionen und NeWS Methoden über eine Verbindungsleitung (*wire*) (vgl. Abschnitt 4.1.4.1), sowie die durch sie bewirkte Anordnung und Anzeige von Bildschirmobjekten.

[26] Das ist die definierende Klasse einer Instanz, in diesem Falle *ClassInsertCircles*, sowie deren Superklassen.

4.4.1 Implementation von OPEN LOOK Komponenten

In diesem Abschnitt wird gezeugt, aus welchen verschiedenen OPEN LOOK Komponenten der Graphikeditor zusammengesetzt ist und wie diese – unter Nutzung der Tycoon Bibliothek *newsenv* (vgl. Abschnitt 4.1.4.2 und [Mü94b]) und darauf aufbauender höherer Komponenentn der Benutzerschnittstelle[27] (*higher gui components*) – erzeugt werden.

Wie in den Abschnitten 3.1 und 3.3 bereits ausgeführt, lassen sich Komponenten der Benutzeroberfläche in drei Arten unterteilen.

- Editorfenster, die den äußeren Rahmen eines Editors vorgeben;
- Eigenschafts- und Kommandofenster, mit denen Eigenschaften von bestimmten Objekten (z.B. von druckbaren Dokumenten) editiert werden können und die Knöpfe zum Ausführen von spezifischen Operationen (z.B. Drucken von Dokumenten) besitzen;
- Kontrollelemente (*controls*), wie Menüs oder Knöpfe, zur Auswahl und Steuerung der Funktionalität in den Fenstern.

Zunächst wird die Realisierung der Editorfenster beschrieben, dann die der Kommandofenster und im letzten Abschnitt die der Kontrollelemente.

4.4.1.1 Editorfenster

Der Graphikeditor besteht aus zwei *Editorfenstern*, die jeweils ein Diagramm als *Applikationsfläche* (vgl. Abschnitt 3.3.1) enthalten. Das *Hauptfenster*[28] beinhaltet das *Referenzdiagramm*, das *Vererbungsfenster* beinhaltet das *Vererbungsdiagramm* als Applikationsfläche. Die beiden Fenster werden, jeweils mit Erläuterungen zu ihrer Funktionalität und Funktionen des *newsenv* zu ihrer Implementation, im Folgenden vorgestellt. Aus didaktischen Gründen wird die Erzeugung und Anbindung der Diagrammflächen erst in Abschnitt 4.4.2 behandelt.

Das Hauptfenster des Graphikeditors ist ein OPEN LOOK Basisfenster (*base window*). Die optische Gestaltung ist in Abschnitt 3.3.2 illustriert. Beim Starten des Graphikeditors erscheint es zuerst auf dem Bildschirm. Als Applikationsfläche besitzt es das *Referenzdiagramm* eines Schemas, welches editiert oder in verschiedenen Sichten angezeigt werden kann. Über zwei Rollbalken (*scrollbars*) kann das Referenzdiagramm im Hauptfenster verschoben werden. Über oben im Fenster befindliche Menüknöpfe können verschiedene Bearbeitungsmodi und Operationen gewählt werden. Deren Implementation wird in Abschnitt 4.4.1.3 beschrieben. Ferner kann über ein Menü das *Vererbungsfenster* geöffnet werden.

Zur *Realisierung* werden die Module *gui*[29], *window* und *borderBag* des *newsenv* benutzt. Es folgen Beispiele für Funktionsaufrufe dieser Module.

- **let** *mainWin = gui.newBase(refPan winLab quit em)*
  Es wird ein neues OPEN LOOK Basisfenster (*base window*) erzeugt und sein Identifikator der Variablen *mainWin* zugewiesen. Der Parameter *refPan* enthält den Identifikator des

[27] Im folgenden werden Komponenten der Benutzerschnittstelle auch als *GUI Komponenten* (GUI bedeutet: *Graphic User Interface*) bezeichnet.

[28] Es wird so benannt, weil es beim Starten des Graphikeditors zuerst angezeigt wird, und von ihm die Hauptfunktionalität ausgeht. Es könnte es auch *Referenzfenster* genannt nennen, weil es das Referenzdiagramm als Applikationsfläche beinhaltet.

[29] Abkürzung für *Generic User Interface*: Die Funktionen erzeugen generische OPEN LOOK Komponenten.

Referenzdiagramms[30]. Er instanziiert die Applikationsfläche des Basisfensters. *winLab* ist die Beschriftung des Fensters, *em* ist der Ereignisverwalter (*event manager*) und *quit* beschreibt die Beendigungsaktion beim Schließen des Fensters.

- *borderBag.addClient(mainWin borderBag.south hscrollBar)*
  Es wird ein vorher erzeugter (horizentaler) Schiebebalken (*Scrollbar*) unten in das Hauptfenster eingefügt.

- *window.map(mainWin)*
  Das Hauptfenster wird auf dem Bildschirm angezeigt.

Das Vererbungsfenster des Graphikeditors ist ein OPEN LOOK Aufklappfenster (*popup window*)[31]. Es wird vom *Hauptfenster* des Graphikeditors aus geöffnet und besitzt als Applikationsfläche das Vererbungsdiagramm, welches angezeigt werden kann. Der sonstige Aufbau entspricht dem des *Hauptfensters*.

Zur *Realisierung* des Vererbungsfensters wird das Modul *gui* verwendet. Die Funktion

gui.newSubwindow(inhPan winLab dismiss unpin mainWin)

erzeugt das Fenster. Der Parameter *inhPan* enthält den Identifikator des Vererbungsdiagramms für die Instanziierung der Applikationsfläche des Aufklappfensters. Der Parameter *mainWin* enthält den Identifikator des oben beschriebenen Hauptfensters als „Elternfenster“ dieses Aufklappfensters.

4.4.1.2 Kommando- und Eigenschaftsfenster

In OPEN LOOK sind Eigenschaftsfenster (*property windows*) und Kommandofenster (*command windows*) spezifiziert, die Ähnlichkeit mit Formularen haben und spezielle Arten von Aufklappfenstern darstellen (vgl. Abschnitt 3.1.3). Nach einer Benutzereingabe in der Anwendung, die sich auf ein bestimmtes Objekt bezieht (z.B. der Menüpunkt *Diagramm drucken*) klappt das Fenster auf, und es können bestimmte *Eigenschaften* (*properties*) des Objekts in den Zeilen des Fensters editiert werden. In der Fußzeile befinden sich ein *Apply*- und ein *Reset*-Knopf. Beim Drücken des *Apply*-Knopfs werden die Eigenschaften des Objekts gesetzt und eventuell weitere Befehle ausgeführt.[32] Bei Betätigung des *Reset*-Knopfs werden die Attribute auf Standardwerte zurückgesetzt und es wird kein Befehl ausgeführt (siehe dazu [SM90b, S. 108 ff.]). Es folgen einige Beispiele für im Graphikeditor verwendete Kommandofenster mit Erläuterungen zu ihrer Implementation.

Ein generisches Fenster zum Formatieren und Drucken von Diagrammen: Beim Aufruf des Fensters wird das zu bearbeitende Diagramm als Parameter angegeben. Interaktiv wird das Format des Ausdrucks (Din A4, Originalgröße oder individuell skaliert), die Ausrichtung (hoch oder quer) sowie das Ausgabemedium (Drucker oder Datei) festgelegt. Durch Drücken des *Apply*-Knopfs wird das Diagramm mit den vorher spezifizierten Eigenschaften auf dem gewählten Ausgabemedium erzeugt. In Abb. 3.7 des Abschnitts 3.3.2 wird das Fenster dargestellt.

---

[30] Die Erzeugung des Referenzdiagramms wird aus didaktischen Gründen in Abschnitt 4.4.2 behandelt.

[31] Aufklappfenster sind immer als Subfenster *Subwindow* eines „Elternfensters“ definiert

[32] Diese Spezifikation weicht etwas von der OPEN LOOK Vorgabe ab, weil dort die Ausführung von Befehlen nur den Befehlsfenstern (*command windows*) vorbehalten ist. Dder formularartige Fensteraufbau ist aber derselbe

Realisiert ist das Aufklappfenster im Modul *printPopup*, einer höheren GUI Komponente. Die Funktion

create(win :Base.T panel :GraphPanel.T schemaName, path :String) :Popup.T

erzeugt ein neues Aufklappfenster. Die Parameter *win*, *schemaName* und *path* spezifizieren das Elternfenster des Aufklappfensters sowie den Schema- und Pfadnamen für ein zu erzeugendes Dokument. Der wichtigste Parameter ist *panel*. Er kann durch Identifikatoren für *beliebige Diagrammflächen* des Bildschirms instantiiert werden. Damit ist das Fenster für die *Formatierung und den Ausdruck verschiedener Diagramme* generisch nutzbar. Es folgen Beispielaufrufe zur Erzeugung von Aufklappfenstern für *verschiedene Diagramme*.

- *printPopup.create(mainWin refPanel schemaName path)* Beim Aufruf wird durch den aktuellen Parameter *refPanel* ein Identifikator des *Referenzdiagramms* als zu bearbeitendes Diagramm angegeben.
- *printPopup.create(inheritPopup inheritPanel inhSchemaName path)* Es wird durch *inheritPanel* ein Identifikator des *Vererbungsdiagramms* als zu bearbeitendes Diagramm angegeben.

Durch Benutzerinteraktionen ausgelöste Operationen werden innerhalb des Moduls *printPopup* durch Aufrufe von TL Funktionen zweier Module realisiert, die kurz vorgestellt und erläutert werden.

```
let bBox   = graphPanel.getClientsBbox(pan)
let header = genPS.getHeader(sName  bBox  a4  upright  scale)
graphPanel.printToPS(pan  pathFile  header)
```

Über das Modul (des erweiterten *newsenv*) *graphPanel* wird durch den Funktionsaufruf *getClientsBbox* die Lage und Größe (*bounding box* – vgl. [AS91]) eines auf dem Bildschirm sichtbaren Diagramms erfragt. Die Funktion *getHeader* des Moduls *genPS* erhält als Eingabeparameter den Schemanamen, die (zuvor erfragte) Diagrammgröße sowie Format- und Skalierungsangaben. Sie gibt einen Tupelwert zurück, der einen *Dateikopf* (*header*) im EPSF Format[33] spezifiziert (siehe dazu [AS91]). Dieser wird neben dem Diagrammidentifikator als aktueller Parameter im Funktionsaufruf *graphPanel.printToPS(...)* angegeben. Auf der Serverseite wird durch diesen Aufruf die druckbare PostScript Datei im EPSF Format erzeugt (vgl. Abschnitt 4.2.3.3).

Ein Aufklappfenster zum Editieren und Einfügen von Werteattributen: Der Name und Typ eines Werteattributs werden interaktiv in diesem Fenster editiert. Durch Betätigung des *Apply*-Knopfs wird das Werteattribut einem zuvor im Referenzdiagramm selektierten Klassenknoten hinzugefügt.

Realisiert ist dieses Fenster durch die Funktion *create* des Moduls *valuePopup*[34]. Es folgt ein Kodebeispiel eines Funktionsaufrufs.

```
let popupWin = valuePopup.create(mainWin  insValPanel)
```

Im Beispiel wird *popupWin* ein Identifikator für das Aufklappfenster zugewiesen. Der Parameter *mainWin* givt den Identifikator des „Elternfensters“ und *insVaPanel* den des *Einfügewerkzeugs*

[33] *Extended PostScript Format*
[34] Dies ist eine höhere GUI Komponente.

für Werteattribute an. Zum Ausführen einer Einfügefunktion wird in *valuePop* eine Einfügemethode für Werteattribute über das Modul (des erweiterten *newsenv*) *insTypes* aufgerufen. Die die Funktionalität realisierenden Methodenaufrufe auf der Seite des NeWS Servers werden in Abschnitt 3.3.2.4 beispielhaft beschrieben .

4.4.1.3 Kontrollelemente

In diesem Abschnitt wird beschrieben, wie *Menüs* und *Knöpfe* durch Aufrufe von Funktionen des *newsenv* erzeugt und in Fenster eingehängt werden. Weitere Kontrollelemente – wie Auswahlen (*settings*), Textfelder (*text fields*) und numerische Felder (*numeric fields*), die beispielsweise zum Aufbau der Eigenschafts- und Kommandofenster (siehe oben) verwendet werden – werden durch ähnliche Funktionsaufrufe wie den unten beschriebenen erzeugt. Sie sind hier nicht Gegenstand der Erörterung. In einem Kodebeispiel wird die Syntax der Funktionsaufrufe gezeigt und anschließend erläutert.

```
let mEditButs= menuButtons.create(object.framebuffer)
let menuClass = menu.create(object.framebuffer)
let menuRef = menu.create(object.framebuffer)

menuButtons.setItemList(mEditButs
  iter.enum of
    tuple "Classes" menuClass end
    tuple "References" menuRef end
  end)

borderBag.addClient(mainWin  borderBag.north  mEditButs)
```

- *menuButtons.create(...)*
  Es wird ein Objekt (*mEditButs*) für eine Liste von Menüknöpfen erzeugt.
- *menu.create(...)*
  Durch zweimaligen Aufruf werden zwei Menü-Objekte (*menuClass* und *menuRef*) erzeugt.
- *menuButtons.setItemList(mEditButs ...)*
  Es werden beschriftete Knöpfe (*Classes* und *References*) in die Menüknopfliste *mEditButs* gehängt und die beiden vorher erzeugten Menüs den Knöpfen zugeordnet.
- *borderBag.addClient(... mEditButs)*
  Die Liste der Menüknöpfe wird oben im Hauptfenster *mainWin* positioniert.

Wie den Menüs Menüpunkte mit jeweiligen *Aktionen* zugeordnet werden, wird im folgenden Abschnitt behandelt.

4.4.2 Integration von Graphikfunktionalität

Dieser Abschnitt befaßt sich mit der Erzeugung von Komponenten des Graphikeditors und das Binden von Funkioonalität an die Menüs. Im ersten Teilabschnitt geht es um die Realisierung zur Erzeugung des Hauptfensters und des Vererbungsfensters sowie zur Anzeige von Diagrammen in den Applikationsflächen. Der zweite Teilabschnitt beschreibt die Realisierung der Erzeugung von spezifischen Werkzeugen zur Diagrammbearbeitung und Implementation von Aufrufen der Werkzeuge über Menüs in den oben erwähnten Fenstern.

4.4.2.1 Erzeugung und Einbindung der Diagrammflächen

Erzeugt werden die Diagrammflächen über *create*-Funktionen in Modulen der erweiterten Tycoon Bibliothek *newsenv* (vgl. Abschnitt 4.3.1). Eingebunden in das jeweilige Editorfenster werden sie bei dessen Erzeugung. Es folgt ein Kodebeispiel zum Erzeugen des Referenzdiagramsm und des es beinhaltenden Hauptfensters.

```
let circlePan = circlePanel.createCalculated(object.framebuffer)
let arrowPan  = arrowPanel.createCalculated(object.framebuffer)
let valuePan  = valuePanel.createCalculated(object.framebuffer)
let refPan = graphPanel.createCalculated(object.framebuffer)

panel.addCalculatedClient(refPan "circPan" circlePan
   "[/SouthWest/SouthWest PARENT POSITION]")
panel.addCalculatedClient(refPan "arrowPan" arrowPan
   "[/SouthWest/SouthWest PARENT POSITION]")
panel.addCalculatedClient(refPan "valuePan" valuePan ...)
  ...
let mainWin = gui.newBase(refPan winLab quit em)
window.map(mainWin)
```

In den ersten drei Zeilen werden zunächst Bildschirmobjekte für die drei Diagrammteilflächen[35] durch Funktionen des erweiterten *newsenv* erzeugt. In der vierten Zeile wird ein Objekt für die *Behälterfläche* erzeugt, die die Teilflächen als Klienten aufnehmen soll.

Die drei Funktionsaufrufe *panel.addCalculatedClient(refPan ...)* fügen der Behälterfläche (*refPan*) jeweils eine Teildiagrammfläche als Komponente hinzu. Damit liegen alle Teildiagramme deckungsgleich voreinander, wodurch ein *vollständiges* Diagramm definiert ist. Das Instanziieren von *Diagramminhalten* aus Graphikmetadaten wird hier aus Gründen der Übersichtlichkeit nicht beschrieben.

Der Aufruf *gui.newBase(...)* erzeugt ein neues Basisfenster (*base window*) mit der Behälterfläche *refPan* (und den davor liegenden Diagrammflächen) als Applikationsfläche. Der Aufruf *window.map(mainWin)* zeigt das Fenster mit dem Diagramm an. Ein graphisches Beispiel ist in Abb. 3.5 in Abschnitt 3.3.1 zu finden.

4.4.2.2 Anbindung von Graphikfunktionalität an Menüs

Wie in Abschnitt 4.2.4 beschrieben, beinhalten Instanzen von *Werkzeugklassen* in der TNT Erweiterung die Funktionalität zur Bearbeitung von Diagrammen. Über die Schnittstellen des erweiterten *newsenv* (vgl. Abschnitt 4.3.1) sind diese erzeugbar. In den folgenden Schritten wird anhand von Kodebeispielen die Initialisierung eines *Einfügewerkzeugs für Klassenknoten des Referenzdiagramms* im Hauptfenster sowie dessen Aufruf über Menüs gezeigt. Das erste Kodebeispiel zeigt die Erzeugung einer Werkzeugfläche, die über die Variable *insCircles* identifiziert wird. Sie wird in die Behälterfläche *refPan* (die im vorherigen Kodebeispiel erzeugt wurde) als Komponente eingehängt.

```
let insCircles = InsCirclePanel.createCalculated(object.framebuffer)
panel.addCalculatedClient(refPan  "insCircles"  InsCircles
  "[/SouthWest/SouthWest PARENT POSITION]")
```

[35] Das sind Klassenknoten- Referenzkanten- und Werteknotenfläche.

Das nächste Kodebeispiel zeigt die Erzeugung eines Menüs mit Menüpunkten (*items*) zum Einfügen (*Insert*) und Löschen (*Delete*) von Klassenknoten sowie die Anbindung der Werkzeugaktivierung an den Menüpunkt *Insert*:

```
let menuClass = menu.create(object.framebuffer)
menu.setLabel(menuClass "Class")

menu.setActionItemList(menuClass  iter.enum of
  tuple  "Insert"
    fun() begin
       drawable.unmap(valuePan)
       canvas.toTop(circlePan)
       drawable.map(insCircles)
       canvas.toTop(insCircles)
       ...
     end
   end
  tuple "Delete" ...
  end)
```

Durch die beiden Funktionsaufrufe *menu.create* und *menu.setLabel* wird ein neues Menü (*menuClass*) erzeugt und seine Beschriftung wird gesetzt. Funktionsaufrufe zur Positionierung des Menüs im Hauptfenster sind im Beispiel weggelassen. Der Aufruf *menu.setActionItemList(menuClass ...)* führt zwei Aufgaben aus.

- **Erzeugung von Menüpunkten**:
  Die Menüpunkte werden durch die Aufzählung (*iter.enum*) von Tupeln erzeugt. Ihre Beschriftung steht in der jeweils ersten Tupelkomponente.

- **Anbindung von Aktionen an die Menüpunkte**:
  In der zweiten Komponente jedes Tupels ist eine Funktion als *Aktion* des jeweiligen Menüpunkts implementiert. Damit wird ein *call back*-Mechanismus installiert (vgl. 4.1.3.2 und 4.1.4.1), der bei jedem Selektieren eines Menüpunkts durch den Benutzer die zugehörige Funktion als *Aktion* ausführt. Der Beispielkode definiert folgenden konkreten Ablauf. Immer, wenn der Benutzer den *Insert*-Knopf des *Class*-Menüs wählt, werden die folgenden, in der anonymen Funktion *fun* definierten, Funktionsaufrufe ausgeführt.

  - *drawable.unmap(valuePan)*
    Die Werteknoten des Diagramms werden nicht angezeigt.
  - *canvas.toTop(circlePan)*
    Die Klassenknotenfläche wird zuoberst auf den Bildschirm gelegt.
  - *drawable.map(insCircles)*
    Die (unsichtbare) Werkzeugfläche wird angezeigt.
  - *canvas.toTop(insCircles)*: Die Werkzeugfläche zum Einfügen von Klassenknoten liegt jetzt zuoberst auf dem Bildschirm, d.h. sie ist für Benutzereingaben *aktiviert* und die Methoden der sie definierenden Klasse übernehmen die Kontrolle für weitere Eingaben (vgl. dazu die Abschnitte 4.2.4 und 4.2.4.3).

Kapitel 5

Integration in STYLE

In diesem Kapitel werden zwei verschiedene Themenkomplexe behandelt, die sich mit der Integration des Graphikeditors in die STYLE Entwicklungsumgebung befassen.

Der erste im folgenden Abschnitt erörterte Komplex beinhaltet die *Integration der Modellierung*. Es wird die Transformation von Daten der *graphischen Modellierung* in eine für die *textuelle Modellierung* geeignete Form beschrieben. Schwerpunkt der Erörterung ist die Vorstellung geeigneter Datenrepräsentationen für verschiedene Anwendungen sowie der darauf aufsetzenden Dienste für Transformationen.

Der zweite Abschnitt befaßt sich mit dem Komplex *Interaktion zwischen den STYLE Softwarewerkzeugen*. Es werden Schnittstellen zur Interaktion der Werkzeuge vorgestellt, die im Zusammenhang mit dem Graphikeditor stehen. Die Schnittstelle des Klasseneditors wird für dessen Aufruf vom Graphikeditor aus benutzt. Ferner wird der Aufruf des Graphikeditors in Menüpunkte des *StyleTop Editor* der STYLE Umgebung eingebunden. Am Ende des zweiten Abschnitts werden exemplarisch die für den konsistenten Wechsel von graphischer in textuelle Modellierung anzuwendenden TL Funktionsaufrufe benannt. Es soll das Zusammenspiel zwischen Wechsel der internen Repräsentation eines OM1 Schemas und Werkzeugwechsel veranschaulicht werden in der Entwicklungsumgebung verdeutlicht werden.

5.1 Graphikrepräsentationen und Textgenerierung

In diesem Abschnitt werden unterschiedliche Repräsentationen von modellierten Daten vorgestellt, die verschiedenen Anwendungszwecken dienen. Ferner werden diejenigen Dienste beschrieben, die Transformationen zwischen den Datenrepräsentationen durchführen. Es wird vermittelt, welche Schritte notwendig sind, um aus angezeigten OM1 Diagrammen textuelle OM1 Schemadefinitionen zu erzeugen (siehe Abb. 5.1 aus [Wet94]).

Der Abschnitt 5.1.1 befaßt sich mit der Gewinnung von Daten aus den Diagrammen der Graphikmodellierung. Im Abschnitt 5.1.2 werden die Graphikmetadaten und ihre Generierung vorgestellt. Der Abschnitt 5.1.3 stellt OM1 Syntaxbäume vor, die zur internen Repräsentaton von OM1 Spezifikationen dienen. Im Abschnitt 5.1.4 werden die Generierungsfunktionen zur Erzeugung von OM1-Syntaxbäumen aus Graphikmedaten beschrieben.

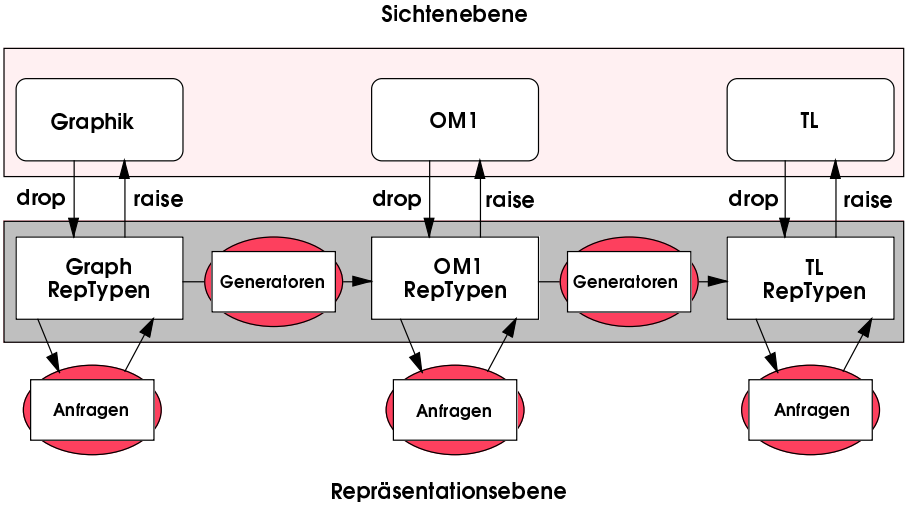


Abbildung 5.1: Generierungsfunktionen auf TL Repräsentation

5.1.1 Gewinnung von Graphikdaten

Die angezeigten Diagramme setzen sich aus Bildschirmobjekten zusammen, die durch NeWS Klasseninstanzen erzeugt werden. Die Diagrammdaten werden in Attributen der Klasseninstanzen gehalten. Für jede dieser Instanzen sind die Attribute und Zugriffsmethoden in der sie erzeugenden NeWS Klasse definiert (vgl. Abschnitt 4.1.3). Ziel der beiden folgenden Abschnitte ist, zu zeigen, wo und wie die Daten der Bildschirmobjekte auf Serverseite gehalten werden und wie sie über TL Funktionen zugreifbar gemacht werden können.

5.1.1.1 Datenstrukturen und Methoden diagrammspezifischer NeWS Klassen

In diesem Abschnitt werden beispielhaft einige diagrammspezifische NeWS Klassen vorgestellt.[1] Dabei werden jeweils die für die Datengewinnung relevanten Attributdefinitionen sowie die sie lesenden *get*-Methoden aufgeführt. Der darauf aufbauende Abschnitt 5.1.1.2 beschreibt TL Funktionen der Module (des erweiterten *newsenv*), die die Daten aus den Klasseninstanzen durch interne Aufrufe der NeWS Klassenmethoden für die TL Anwendung zugänglich machen.

Es folgt ein Kodebeispiel der TNT Klasse *ClassCircleItem*, die Bildschirmobjekte für *Klassenknoten* definiert. Es werden zunächst die schemarelevanten Attribute der Klasse, ihre Initialisierung sowie *get*-Methoden zum lesenden Zugriff auf Attributwerte vorgestellt.

```
/ClassCircleItem  ClassCircleShape []
 classbegin
 ...
  /X null def
  /Y null def
  /LabelText () def
  /LabelPosKind (East) def

  /NewInit  {
```

[1] Diese werden ausführlich in Abschnitt 4.2 als neue Subklassen der TNT Klassenhierarchie beschrieben.

```
    /NewInit super send
    /LabelText exch promote
    /PosKind exch promote
    /Y exch promote
    /X exch promote
     ...
  } def

  /getX {X} def
  /getY {Y} def
  /getLabelText {LabelText} def
  /getLabelPosKind {PosKind} def
 ...
classend def
```

Die *Attributdefinitionen* stehen in den ersten vier Zeilen nach **classbegin**. Sie beschreiben die x/y-Koordinaten (die mit *Null*-Werten initialisiert werden), den Klassennamen der Klassenknotenbeschriftung (*LabelText*) sowie die relative Position der Beschriftung zum Kreissymbol des Klassenknotens (hier mit *East* – also rechts davon – initialisiert).

Die *Initialisierung der individuellen Attributwerte* erfolgt beim Erzeugen einer Instanz eines Klassenknotens durch die Methode *NewInit*: Es werden die vier oben beschriebenen Attribute, die als Eingabeparamter auf einem Kellerspeicher liegen, durch **promote** Prozeduren gesetzt.[2] Die vier *get-Methoden* am Ende des Beispiels dienen zum Auslesen der Attributwerte.

Um die relevanten Attribute *aller Klassenknoteninstanzen eines Diagramms* auslesen zu können, sind in der TNT Klasse *ClassCirclePanel* weitere *get*-Methoden definiert, die jeweils ein Feld (*array*) von Werten *desselben Attributs* aller Klassenknoten zurückgeben.

```
/getClientLabels {
  /clientlist self send
  dup length /setItemCount self send
   {/getLabelText exch send}forall
  getItemCount array astore
} def

/getClientLabelPosKinds { ... } def
/getClientXValues ...
/getClientYValues ...
```

Am Beispiel der Methode *getClientLabels*, die alle Klassenknotennamen eines Diagramms ausliest, werden die inneren Aufrufe exemplarisch erläutert.

- *clientlist*
  ist eine aus *ClassPanel* geerbte Methode. Sie gibt ein Feld (*array*) aller *Klassenknoteninstanzen* der Diagrammfläche zurück.

- *length setItemCount*
  speichert die Anzahl der Feldelemente.

[2] Genauer gesagt, wird durch den Aufruf **promote** jeweils eine Instanzvariable erzeugt und gesetzt.

- {*getLabelText* ***exch send***} ***forall***
  bewirkt, daß an jede Klassenknoteninstanz des Feldes die Methode *getLabelText* gesendet wird. Nach der Ausführung stehen jetzt alle Klassenknotennamen auf dem Kellerspeicher des NeWS Servers.

- *getItemCount* ***array astore***
  bewirkt das Lesen der (vorher gespeicherten) Elementzahl und die dynamisch Erzeugung ein neuen Feldes, welches alle Klassenknotennamen enthält.

Beurteilung der Datenstrukturen und ihres Zugriffs: Es ist festzustellen, daß als *Datenstruktur* für Mengen von Daten auf der Serverseite nur das in PostScript definierte Feld zur Verfügung steht, welches nur atomare Werte eines Typs enthalten kann. Vorteilhaft ist, daß Felder, unter Angabe ihrer Länge, dynamisch erzeugt werden können.

5.1.1.2 TL Funktionen zum Auslesen von Diagrammdaten

Im erweiterten *newsenv* werden die im vorigen Abschnitt vorgestellten Methoden in TL Funktionen des Moduls *circlePanel* aufgerufen.

getLabels(circPan :CirclePanel.T) :Iter.T(String)
getXValues(circPan :T) :Iter.T(Int)
getYValues(circPan :T) :Iter.T(Int)
getLabPosKinds(circPan :T) :Iter.T(String)

Die Funktion *getLabels(...)* gibt eine Iteration aller Klassenknotennamen der im Parameter *circPan* spezifizierten Instanz einer Klassenknotenfläche zurück. Die drei weiteren Funktionen dienen zum Lesen der Koordinaten und der Labelpositionen aller Klassenknoten eines Schemas.

Die hier am *Klassenknotenbeispiel* vorgestellte Verfahrensweise für Attributdefinitionen und Methoden sowie für Funktionen zum Lesen von Attributwerten ist übertragbar auf *Referenzkanten*, *Werteattribute* und *Vererbungspfeile*. Im Modul *arrowPanel* gibt es unter anderem folgende *get*-Methoden für alle Referenzkanten eines Diagramms.

getXSources(arrowPan :T) :Iter.T(Int)
getYSources(arrowPan :T) :Iter.T(Int)
getXTargets(arrowPan :T) :Iter.T(Int)
getYTargets(arrowPan :T) :Iter.T(Int)
getRefKinds(arrowPan :T) :Iter.T(String)
getKeyKinds(arrowPan :T) :Iter.T(Bool)
getLabels(arrowPan :T) :Iter.T(String)

Die ersten vier Funktionen liefern die x/y-Koordinaten der Start- und Endkoordinaten aller Referenzkanten. Die Funktion *getRefKinds(...)* gibt eine Iteration über die *Kardinalitäten* der Referenzkanten zurück.[3] *getKeyKinds(...)* liefert eine Iteration von booleschen Werten zurück. Jeder Wert besagt, ob die betreffende Referenzkante ein Schlüsselattribut repräsentiert oder nicht. Die Funktion *getLabels(..)* gibt die Namen aller Referenzkanten (die die Attributnamen repräsentieren) zurück.

Für *Werteattribute* gibt es entsprechende *get*-Methoden, die Iterationen für jeweils folgende Attribute einer Werteattributinstanz im Diagramm liefert.

[3] Der Wert "singleCase" gibt die Einwertigkeit, der Wert "setCase" die Mengenwertigkeit der Referenz an.

- Name des Werteattributs;
- Name der Domäne des Werteattributs (z.B. *Integer*, *Real* etc.);
- x/y-Koordinate der Position des Klassenknotens, dem das Werteattribut im Diagramm zugeordnet ist;
- Boolesches Attribut, das über die Schlüsseleigenschaft des Werteattributs Auskunft gibt.

Für *Vererbungspfeile* gibt es lediglich *get*-Funktionen, die die Start- und Endkoordinaten aller Vererbungspfeile des Vererbungsdiagramms zurückgeben.

Beurteilung der Funktionalität: Unter Nutzung der Module bewirken Aufrufe von TL Funktionen[4] intern die Aufrufe der entsprechenden TNT Klassenmethoden auf der Serverseite. Die Funktionalität dieser Funktionen abstrahiert von der Tatsache der Verschiedenheit zwischen Client (auf Tycoonseite) und dem NeWS Server bezüglich der Strukturierbarkeit von Daten und Zugriffsmöglichkeiten auf Massendaten. Während des Aufrufs wird ein beliebig langes Feld (*PostScript array*) in einen Iterator vom (in der Tycoon Bibliothek *stdenv* definierten) Typ *Iter.T* transformiert, wohingegen die Leitung (*wire*) jedoch nur einen elementweisen Zugriff auf Rückgabewerte gestattet. Jede dieser *get*-Funktionen enthält weitere Aufrufe, die die folgende Funktionalität erfüllen.

- Aufbau einer synchronisierten Kommunikation zwischen Client und Server;
- Senden des Methodenaufrufs an den NeWS Server, der das Feld von Werten erzeugt;
- Senden einer *Auspackprozedur* an den Server, die das Feld entpackt und seine atomaren Werte im *PostScript* Typformat einzeln auf Serverseite ausgibt sowie die Anzahl der atomaren Werte zurückgibt;
- Lesen jedes atomaren Wertes über die Verbindung bei gleichzeitiger Typkonvertierung in einen TL Datentyp;
- Erzeugung einer Iteration über die gelesenen Werte.

Die Implementation dieser Funktionalität ist durch Nutzung der Module *wire* und *iter* nicht aufwendig. Sie umfaßt in der Regel ca. zehn Programmzeilen je *get*-Funktion. Die Datengewinnung aus Diagrammen durch Verwendung von TL Funktionen hat die folgenden Vorteile.

- Abstraktion von Kommunikationsprotokollen und syntaktisch andersartigen Methodenaufrufen der Serverseite;
- Typsicherheit durch 1:1-Typkonvertierung auf atomarer Ebene;
- vereinfachter Zugriff auf Massendaten als Rückgabewerte von TL Funktionen durch Iterationsabstraktion.

Die Benutzung der in diesem Abschnitt vorgestellten *get-Funktionen* zur Erzeugung von die Graphik repräsentierenden *Metadaten* wird im nächsten Abschnitt 5.1.2 behandelt.

[4] Die Parameter dieser Funktionen spezifizieren diejenigen Klasseninstanzen, die die jeweiligen die Teildiagramme anzeigenden Bildschirmobjekte definieren.

5.1.2 Generierung von Graphikmetadaten

Die Abschnitte 5.1.2.1 bis 5.1.2.3 befassen sich mit der Einführung von Graphikmetadaten. Es werden zunächst Anforderungen an Graphikmetadaten hinsichtlich verschiedener Repräsentationsaufgaben formuliert. Dann folgt die Vorstellung von TL Datentypen zur Repräsentation von Metadaten. Diese werden anschließend anhand der Anforderungen beurteilt. Im Abschnitt5.1.2.4 werden TL Funktionen zur Transformaton von Graphikdaten, die aus auf dem Bildschirm angezeigten Diagrammen gewonnen werden (vgl. Abschnitt 5.1.1.2), in Graphikdaten vorgestellt.

5.1.2.1 Anforderungen an Graphikmetadaten

Die Daten zur Graphikrepräsentation in Tycoon – im folgenden als (*Graphik-*)*Metadaten* bezeichnet – stellen eine Schnittstelle für zwei Anwendungskategorien dar. Die erste ist das *Laden und Sichern von Diagrammen*, die mit dem Graphikeditor interaktiv erstellt werden; die zweite Kategorie ist die *Generierung von OM1-Syntaxbäumen* (siehe Abschnitt 5.1.4) aus Graphikmetadaten, aus denen textuelle Spezifikationen in der Modellierungssprache OM1 erzeugt werden können (vgl. Abschnitt 2.3). Aus den beiden Anwendungskategorien ergeben sich jeweils *verschiedene Anforderungen* an die Metadaten hinsichtlich der zu repräsentierenden Informationen, wie in Tabelle 5.1 gezeigt wird. Die linke Spalte beschreibt die aus der graphischen Darstellung zu extrahierenden unterschiedlichen Informationen, die aus dem Informationsbedarf der beiden Anwendungskategorien resultieren. Der Informationsbedarf ist in den beiden folgenden Spalten, kategorieweise geordnet, aufgeführt.

Die Tabelle 5.1 zeigt, daß es gemeinsam verwendbare Information (vgl. die ersten fünf Zeilen der Tabelle) gibt. Sie betrifft schemarelevante Eigenschaften, die sowohl graphisch als auch in OM1 textuell darstellbar sind. Unterschiedlich in den Anforderungen ist, daß für die Diagrammdarstellung die *topologische Anordnung der Basisobjekte* aufgrund von Rasterpositionen wichtig ist (vgl. die Punkte unter den Einträgen *„Rasterpositionen“* und *„Referenzkantenausrichtung“* in der Tabelle), während für die textuelle Schemaspezifikation, die durch Generierung von textuellen Schemabeschreibungen in OM1-Syntax erzeugt wird, die *Beziehungen zwischen Objekten* eine große Rolle spielen (vgl. die unteren drei Tabellenabschnitte).

Zusammenfassend gesagt, beschreiben die Einträge der linken Spalte genau die aus den beiden Anwendungskategorien resultierenden Anforderungen an die zu repräsentierende Information in den Graphikmetadaten. In Abschnitt 5.1.2.3 wird darauf eingegangen, inwieweit eine gewählte Datenrepräsentation für die Graphikmetadaten die in den Spalteneinträgen aufgeführten Anforderungen erfüllt.

Geforderte Information aus der Graphik	*Anwendungskategorien* **Diagrammdarstellung**	**Schemaspezifikation**
Klassenknotenname	Beschriftung des jeweiligen Symbols	Klassenname
Werteattributname		Attributname
Wertetypbezeichner		Attributtyp
Referenzattributname		Attributname
Attributeigenschaften	Wagenradsymbol und Labelunterstreichungen	Attributarten (*kinds*)
Rasterpositionen von: Klassenknoten Referenzkanten Werteknoten Vererbungspfeilen Klassenlabeln	Positionierung der Diagrammkomponenten im Referenz- und Vererbungsdiagramm	—
Referenzkantenausrichtung	Alternative Ausrichtung einer Referenzkante	—
Beziehungen zwischen Klassenknoten über Referenzkanten	—	Alle referenzwertigen Attribute einer Klasse
Beziehungen zwischen Klassenknoten über Vererbungspfeile	—	Alle Superklassen einer Klasse
Werteknoten mit Werteattributen eines Klassenknotens	—	Alle Werteattribute einer Klasse

Tabelle 5.1: Anforderungen an Graphikmetadaten

5.1.2.2 TL Datentypen für Graphikmetadaten

In diesem Abschnitt werden beispielhaft einige in TL definierte Datentypen vorgestellt, die als Datenstruktur für die Graphikmetadaten dienen.

Klassenknoten: In TL wird der Tupeltyp *ClassNode* definiert, dessen Komponenten im folgenden kurz beschrieben werden. Die Struktur und zusätzliche Typen sind in dem nachfolgenden Kodebeispiel veranschaulicht.

```
Let Position = Tuple x, y :Int end
Let LabPosKind = Tuple  case  left, right, upper, lower end

Let ClassNode =
  Tuple
    var name :String
    var refPosition :Position
    var inhPosition :Position
    var refLabPosKind :LabPosKind
    var inhLabPosKind :LabPosKind
  end
```

Es folgt die Erläuterung der Tupelkomponenten. Die Komponente *name* vom TL Basistyp *String* enthält den Namen eines Klassenknotens, der in der Graphik angezeigt wird. *refPosition* und *inhPosition* spezifizieren Positionen des Klassenknotens im Referenzdiagramm und im Vererbungsdiagramm. *refLabPosKind* und *inhLabPosKind* enthalten die (zum Klassenknoten relativen) Labelpositionen im Referenzdiagramm und im Vererbungsdiagramm. Mögliche Werte sind im Optionstyp *LabPosKind* definiert.

Werteknoten: Werteknoten werden durch den TL Tupeltyp *ValueNode* repräsentiert.

```
Let Kind = Tuple case key, set end

Let ValueComp =
   Tuple
      labelName :String
      typeName  :String
      var kinds :list.T(Kind)
   end

Let ValueNode =
   Tuple
      var position   :Position
      components     :varList.T(ValueComp)
   end
```

Die Tupelkomponenten *position* und *components* spezifizieren neben der Diagrammposition die Werteknotenkomponenten, welche als Typ eine variable Liste des Tupeltyps *ValueComp* haben. Der Typ *ValueComp* wiederum repräsentiert ein *Werteattribut* durch Komponenten, die den Namen, den Typ und Schlüssel- sowie Kardinalitätseigenschaften eines Werteattributs beinhalten.

Vererbungspfeil: Der TL Typ *Inheritance* enthält die beiden Komponenten *sourceNode* und *destNode*. Sie referenzieren den erbenden und den vererbenden Klassenknoten einer Vererbungsbeziehung. Die Start- und Zielpositon des Vererbungspfeils sind in den entsprechenden Komponenten der referenzierten Klassenknoten (s.o.) spezifiziert. Es folgt das Kodebeispiel für die Datenstruktur.

```
Let Inheritance =
   Tuple
      sourceNode  :ClassNode
      destNode    :ClassNode
   end
```

Schema: Es wird durch den Datentyp *Scheme* repräsentiert (siehe dazu das Kodebeispiel unten). Die vier Tupelkomponenten haben als Wertebereiche Listen von Typen für Klassenknoten, Referenzkanten, Werteknoten und Vererbungspfeile. Diese Typen wurden, bis auf TL-Datentypen für Referenzkanten, auf deren Einführung hier verzichtet wird, bereits oben erläutert.

```
Let Scheme =
   Tuple
```

```
    classNodes :varList.T(ClassNode)
    references :varList.T(Reference)
    valueNodes :varList.T(ValueNode)
    inheritances :varList.T(Inheritance)
end
```

5.1.2.3 Beurteilung der Graphikmetadaten hinsichtlich der Anforderungen

Die in dem vorigen Abschnitt eingeführten TL Daten erfüllen die in Abschnitt 5.1.2.1 tabellarisch aufgelisteten Anforderungen weitgehend. In der Tabelle 5.2 sind die Anforderungen wiederholt aufgelistet (siehe linke Spalte). In der rechten Spalte sind die korrespondierenden Repräsentationstypen bzw. deren Tupelkomponenten aufgeführt, die diese Anforderungen erfüllen. Um Informationen über *Beziehungen* zwischen Diagrammelementen zu erhalten(siehe die letzten drei Abschnitte der linken Spalte), sind Anfragefunktionen definiert, die aus den Metadaten die Beziehungen über direkte Tupelreferenzen sowie über Vergleiche von Rasterpositionen ableiten. Dieses Vorgehen wird beispielhaft erläutert.

Beispiel einer Anfragefunktion: Die in dem Modul *queries* definierte Funktion *getClassNodeByInhPos* gibt beispielsweise zu einer angefragten Position ein Tupel des Typs *ClassNode* zurück, das einen Klassenknoten repräsentiert. So können aus der Start- und Zielposition eines im Diagramm angezeigten Vererbungspfeils die entsprechenden Identifikatoren von Klasseknoten-Tupeln angefragt und als Komponentenwerte (*sourceNode*, *desNode*) einer Vererbungsbeziehung (*Inheritance*) gesetzt werden. Mithilfe dieser Identifikatoren können beispielsweise alle direkten Vererbungsbeziehungen einer Klasse angefragt werden, wie das folgende Kodebeispiel zeigt.

```
let getInheritances(cN :ClassNode  inhs :varList.T(Inheritance))  :Iter.T(Inheritance) =
    iter.select(varList.elements(inhs)  fun (inh :Inheritance) inh.sourceNode == cN)
```

Eingabeparameter sind ein Klassenknoten *cN* und eine variable Liste aller Vererbungsbeziehungen einer Schemarepräsentation. Durch Anwendung der Funktion *iter.select* aus dem Tycoon Bibliotheksmodul *iter* werden aus der Iteration über alle Listenelemente der Vererbungsbeziehungen (*varList.elements(inhs)*) diejenigen zurückgegeben, deren Startklassenknoten (*inh.sourceNode*) identisch mit dem Klassenknoten *cN* ist.

In ähnlicher Weise wird bei Anfragen der Namen der von einem Klassenknoten ausgehenden Referenzkanten vorgegangen.

Die Werteattribute eines Klassenknotens werden wie folgt angefragt. Durch einen Positionsvergleich zwischen dem Klassenknoten und sämtlichen Werteknoten wird derjenige Werteknoten identifiziert (mit identischem Positionswert), der die dem Klassenknoten zuzuordnenden Werteattribute im Diagramm enthält. Weitere mögliche Anfragefunktionen zur Integritätsüberwachung von graphisch modellierten OM1 Schemata dienen der Überprüfung von unzulässigen Mehrfachverwendungen von Klassennamen (in einem Schema) und Attributnamen einer Klasse (in den entsprechenden Werteattributen und Referenzkanten eines Klassenknotens).

Geforderte Information aus der Graphik	**Repräsentationen in den Metadaten**
Klassenknotenname	*ClassNode.name*
Werteattributname	*ValueComp.labelName*
Wertetypbezeichner	*ValueComp.typeName*
Referenzattributname	*Reference.label*
Attributeigenschaften	*list.T(Kind)*
Positionen von:	Vom Typ *Position*:
Klassenknoten	*ClassNode.refPosition* *ClassNode.inhPosition*
Referenzkanten	*Reference.sourceNode.refPosition* *Reference.desNode.refPosition*
Werteknoten	*ValueNode.position*
Vererbungspfeilen	*Inheritance.sourceNode.inhPosition* *Inheritance.destNode.inhPosition*
Klassenlabeln	*ClassNode.refLabPosKind* *ClassNode.inhLabPosKind*
Referenzkantenausrichtung	*RefPosKind*
Beziehungen zwischen Klassenknoten über Referenzkante	Anfragefunktionen
Beziehungen zwischen Klassenknoten über Vererbungspfeile	
Werteknoten mit Werteattributen eines Klassenknotens	

Tabelle 5.2: Anforderungen an Graphikmetadaten und die sie realisierenden TL Repräsentationen

5.1.2.4 Generierungsfunktionen

Mit Hilfe der in Abschnitt 5.1.1.2 vorgestellten *get*-Funktionen, die aus auf dem Bildschirm angezeigten Diagrammen Daten extrahieren, lassen sich Graphikmetadaten als Werte von Repräsentationstypen erzeugen. Dies wird an einem Kodebeispiel, welches Graphikmetadaten aus den Klassenknoten eines Diagramms erzeugt, gezeigt. Der Datentyp *Scheme* des Formalparameters *schema* wurde bereits in Abschnitt 5.1.2.2 eingeführt.

```
let setGRTClasses(panel :CirclePanel.T schema :Scheme) :Ok =
  begin
    let iLabels = circlePanel.getLabels(panel)
    let iXVals = circlePanel.getXValues(panel)
    let iYVals = circlePanel.getYValues(panel)
     ...
    for j = 1 upto n do
      let cn :GraphRep.ClassNode =
```

```
        tuple
          let  var label = iter.nth(iLabels j)
          let  var refPosition =
            tuple
              let  x = iter.nth(iXVals j)
              let  y = iter.nth(iYVals j) end
                ...
            end
        end
      graphRep.insertClass(schema.classNodes cn)
    end
  end
```

Der Paramater *panel* identifiziert die Instanz, die die *Klassenknotenfläche* eines Referenzdiagramms anzeigt. In der Funktion *setGRTClasses* wird folgendes realisiert. Durch *get*-Funktionen (*circlePanel.get...*) werden alle Klassennamen (*labels*) sowie alle x/y-Koordinaten des Klassenknotenteildiagramms gelesen. Sie sind jeweils in *Iteratoren* geordnet. Aus den Iteratoren wird (in der *for*-Schleife) jeweils ein Wert *cn* vom Typ *ClassNode* erzeugt und in die Klassenknotenliste (unter Verwendung der Funktion *graphRep.insert*) in das Schema *schema*, welches die Metadaten enthält, eingefügt. Die durch "..." angedeuteten ausgelassen Kodefragmente enthalten Iteratoren und Tupelkomponentenkonstruktoren für Vererbungsdiagrammpositionen (*inhPosition*) und Labelpositionen (*refLabPosKind* und *inhLabPosKind*), für die analog verfahren wird.

Ähnliche Funktionen stehen zur Erzeugung von Metadaten für Referenzkanten, Werteknoten und Vererbungspfeile zur Verfügung.

5.1.3 OM1 Syntaxbäume

In diesem Abschnitt wird zunächst auf die Bedeutung von OM1 Syntaxbäumen im Rahmen der OM1 Umgebung eingegangen. Es folgt eine Beschreibung des Aufbaus, wobei kurz auf die Datenstrukturen eingegangen wird, die die Syntaxbäume als *Repräsentationstypen* in TL implementieren. Eine Kenntnis des Aufbaus der OM1 Syntaxbäume und der korrespondierenden Repräsentationstypen ist notwendig, um die in Abschnitt 5.1.4 beschriebene Generierung von Werten von OM1 Syntaxbäumen aus Graphikmetadaten verstehen zu können.

5.1.3.1 Bedeutung

OM1 ist eine Spezifikationssprache, die durch eine *Sprachsyntax* formal beschrieben wird [Wet94]. In [Mü94a] und [Wet94] werden die Sprachkonzepte und die Grammatik von OM1 durch eine abstrakte *abstrakte OM1 Syntax*, bestehend aus abstrakten *OM1 Syntaxkategorien* und *Produktionen* darauf, beschrieben. Die Produktionen werden in *EBNF*[5] Notation dargestellt (vgl. [Wet94, S.114]).
Für die abstrakten OM1 Syntaxkategorien werden in TL Repräsentationstypen (Syntaxbäume) zur Verfügung gestellt, auf denen Werte als Repräsentationen von OM1 Schemata durch Transformationsdienste erzeugt werden. Auf den erzeugten Werten arbeiten wiederum Generierungsfunktionen, die Werte von TL Syntaxbäumen – als interne Repräsentation von Datenbankschnittstellen und Modulen in TL – erzeugen. Für eine detaillierte Darstellung der Generierungsfunktionen wird auf [Mü94a, Kapitel 5] verwiesen.

[5] Extended Backus Naur Formalism

OM1 Syntaxbäume definieren somit eine interne, uniforme Datenstruktur zur Integration unterschiedliche Entwurfsobjekte und darauf arbeitender Dienste der OM1 Entwicklungsumgebung. Im nächsten Abschnitt wird der Aufbau der OM1 Syntaxbäume in Ausschnitten anhand der korrespondierenden TL Repräsentationstypen erläutert.

5.1.3.2 Aufbau

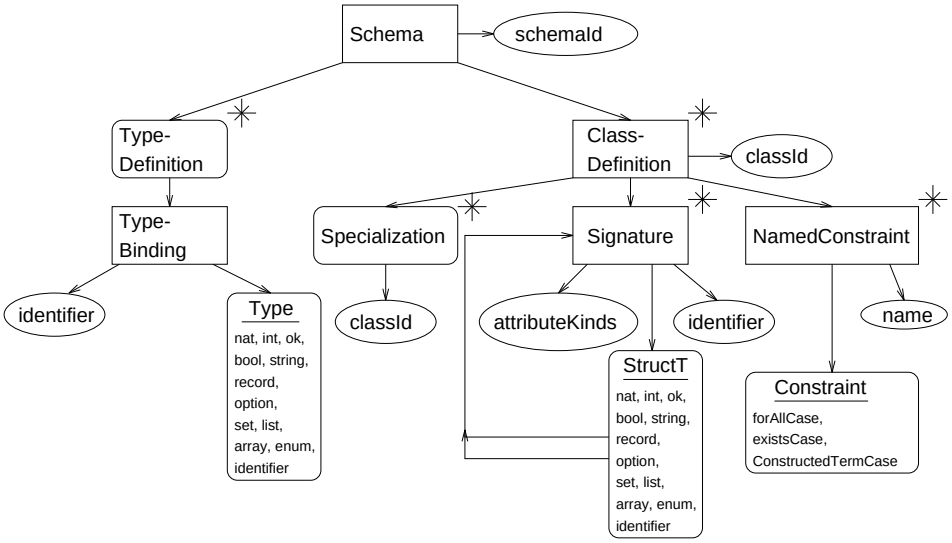


Abbildung 5.2: Abstrakter OM1 Syntaxbaum

Für die in Abschnitt 2.3.1) vorgestellten Konzepte Schema, Typ, Klasse und deren Komponenten gibt es korrespondierende Syntaxbäume. Die Abbildung 5.2 (aus [Gei94]) gibt einen Überblick über die Struktur von Syntaxbäumen und der sie realisierenden Typstrukturen. TL Tupeltypen sind als Rechtecke, Optionstypen als Rechtecke mit gerundeten Ecken und skalare Tupel-/Optionsfelder als Ellipsen dargestellt. Ein Stern an der rechten oberen Ecke eines Symbols kennzeichnet, daß es sich um eine Liste von Elementen des jeweiligen Typs handelt. Pfeile sind als *„besteht aus“* zu lesen. Der rekursive Aufbau des Syntaxbaumes zeigt sich daran, daß *Signaturen* (Attribute) als Wertebereich wiederum Listen von Signaturen (*Record*s oder *Options*) besitzen können.

Es wird exemplarisch für die abstrakte Syntaxkategorie *Class*, die eine OM1 Klasse repräsentiert, der korrespondierende Repräsentatonstyp als TL Tupeltyp vorgestellt.

```
Let Class =
  Tuple
    classId :TLIde.T
    specialization :Specializations
    structure :Structure
  end
```

Die Tupelkomponenten verweisen auf Typen, die einen Bezeichner (*TLIde.T*), eine Superklassenliste (*Specializations*) und die Attributliste (*Structure*) einer OM1 Klasse spezifizieren.[6]

[6] Die Komponente *NamedConstraint* des in Abb. 5.2 gezeigten abstrakten Syntaxbaums wird in der derzeitigen Implementation des Graphikeditors nicht visualisiert.

Nähere Erläuterungen der abstrakten Syntax und Beispiel weiterer TL Repräsentationstypen für die Syntaxkategorien sind in [Mü94a] zu finden.

5.1.4 Generierung von OM1 Syntaxbäumen

Die Generierungsfunktionen leisten einen essentiellen Beitrag, um in der STYLE Umgebung von der graphischen Modellierung zur textuellen Modellierung einer Anwendung zu gelangen. Im folgenden wird die Generierung von Werten der OM1 Syntaxbäume aus Graphikmetadaten in Ausschnitten beschrieben. Schwerpunkt ist die Beschreibung der Methodik der für die Generierungsfunktionalität verwendeten TL Funktionen.

5.1.4.1 Generierung einer OM1 Klassenrepräsentation

In diesem Abschnitt werden exemplarisch Signaturen von TL Funktionen und Teilfunktionen zur schrittweisen Generierung von Tupelkomponenten des TL Typs *Class* (s.o.) vorgestellt. Es folgen jeweils Erläuterungen und teilweise Kodefragmente zur genaueren Illustration. Am Ende dieses Abschnitts wird in einer Zusammenfassung die Generierungsfunktion für die Repräsentation einer OM1 Klasse vorgestellt, die die Teilfunktionen verwendet.

Zur Unterscheidung von Typbezeichnern der Graphikmetadaten und der OM1 Syntaxbäume in TL werden die Präfixe der definierenden TL Schnittstellen vorangestellt. Der Präfix *GraphRep* bezeichnet Typen der Schnittstelle für Graphikmetadaten. Präfixe mit dem Anfangsbuchstaben "0" (z.B. *OClass* und *OStructure*) bezeichnen Typen der Schnittstellen für OM1 Syntaxbäume.

Die Superklassenliste vom Typ (*OClass.Specializations*) wird durch die Funktion *genOSpecs* generiert. Deren Signatur wird hier aufgeführt.

```
genOSpecs(classNode :GraphRep.ClassNode
  inhList :varList.T(GraphRep.Inheritance)): OClass.Specializations
```

Als Parameter werden ein Klassenknoten und die Liste der Vererbungsbeziehungen aus den *Graphikmetadaten* eines Schemas angegeben. Der Rückgabewert ist die Superklassenliste einer OM1 Klasse als Wert eins OM1 Syntaxbaums. Die Generierungsfunktionalität dieser Funktion wird durch ein Kodebeispiel verdeutlicht.

```
let classInherits = iter.select(varList.elements(inhList)
    fun(inh :GraphRep.Inheritance) inh.sourceNode == classNode)

let genOSpec(inherit :GraphRep.Inheritance) :OClass.Specialization =
  tuple case subClassCase of OClass.Specialization with
      ...
    let classId = genId.refIdCase(inherit.destNode.label)
  end

list.create(iter.map(classInherits genOSpec))
```

Die Generierung wird durch Selektions- und Transformationsfunktionalität von Funktionen des TL Moduls *iter* unterstützt. Zunächst werden durch die Funktion *iter.select* alle erreichbaren Superklassen aus der Liste der Vererbungsbeziehungen selektiert, die den spezifizierenden Klassenknoten als *erbenden Knoten* besitzen. Dann wird lokal die Funktion *genOSpec* definiert, die

aus einem Klassenknoten der Graphikmetadaten eine Superklasse als Wert des OM1 Syntaxbaums erzeugt. Ein nachfolgender Aufruf der generischen Funktion *iter.map* mit der Funktion *genOSpec* als aktuellem Parameter bewirkt deren Anwendung auf jeden Klassenknoten der anfangs selektierten Iteration. Schließlich wird durch die Funktion *list.create* aus der (transformierten) Iteration eine Superklassenliste als Rückgabewert erzeugt. An dem Beispiel wird deutlich, daß durch die Anwendung generischer Bibliotheksfunktionen in TL die Transformation zwischen unterschiedlich strukturierten Repräsentationen von Schemainformationen effizient zu kodieren ist.

Die Struktur der Klasse wird durch eine Liste von (Attribut-)Signaturen im OM1 Syntaxbaum dargestellt und ist vom Typ (*OClass.Structure*). Es folgt die Signatur der sie generierenden Funktion.

```
genOStructure(classNode :GraphRep.ClassNode
              refList :varList.T(GraphRep.Reference)
              valList :varList.T(GraphRep.ValueNode) :OClass.Structure
```

Als Parameter werden neben dem Klassenknoten die Listen der Referenzkanten und Werteattribute aus den *Graphikmetadaten* eines Schemas angegeben. Der Rückgabewert ist die Klassenstruktur als Wert eines OM1 Syntaxbaums. Die Generierungsfunktionalität ist analog zu der im Beispiel der Superklassenliste (s.o.) kodiert und wird nur kurz erläutert. Aus den in den Graphikmetadaten *getrennt* gehaltenen Informationen für Werteattribute und Referenzattribute werden wiederum die mit dem spezifischen Klassenknoten in Zusammenhang stehenden über referentielle Vergleiche *selektiert*. Die Funktionen *iter.select*, *iter.map* und *list.create* werden entsprechend angewendet. Von dem teilweisen Zugriff auf tiefer geschachtelte Datenstrukturen wird hier abstrahiert. Dieser wird durch entsprechende Schachtelungen von Funktionen und Iteratoren getätigt.

Zusammenfassung: Um eine Repräsentation einer OM1 Klasse zu generieren, werden die bereits eingeführten Funktionen *genOSpecs* und *genOStructure* verwendet. Es folgt der Programmkode der TL Funktion zur Erzeugung der Repräsentation.

```
let genOClass(s :GraphRep.Schema) (cn :GraphRep.ClassNode) :OClass.Class =
  tuple
    ...
    let classId = genId.defIdCase(cn.label)
    let specialization = genOSpecs(cn s.inheritances)
    let structure = genOStructure(cn s.references s.valueNodes)
    ...
  end
```

Im Beispiel sind nur die für die Generierung aus den Graphikmetadaten relevanten Tupelkomponenten der OM1 Klassenrepräsentation aufgeführt.[7] Die Funktion *genId.defIdCase* erzeugt aus dem Klassenknotennamen einen definierenden TL Bezeichner.

Für die Generierung eines *Schemas* aus *allen Klassenknoten* der Graphikmetadaten werden wiederum die generischen Funktionen *iter.map* und *list.create* angewendet. Es folgt ein relevantes Kodefragment einer Generierungsfunktion. Der Aufruf

[7] Für die OM1 Klassenkomponenten *parameters*, *instantiation* und *constraints* werden nur *Nullwerte* generiert, da es keine graphische Modellierung für diese Komponenten gibt.

iter.map(classNodeIter genOClass(gSchema))

bewirkt die Anwendung der klassengenerierenden Funktion *genOClass* auf alle Klassen eines in Graphikmetadaten repräsentierten Schemas.

5.1.4.2 Metadatenanbindung bei Generierung

Aufgaben der Metadatenverwaltung sind, OM1 Syntaxbäume von Schemata strukturierbar, einfach zugreifbar und änderbar sowie persistent zu halten. Das TL Modul *dictionary*[8] wird als Basisdienst benutzt, um polymorphe Wörterbücher (*dictionaries*) für OM1 Schemata, OM1 Typen und OM1 Klassen zu instanziieren und hierarchisch zu gliedern sowie Einträge zu verwalten (einfügen, löschen, abfragen, überschreiben). Weitere, darauf aufsetzende Dienste abstrahieren von diesen Wörterbüchern und liefern spezifische Funktionalität für die Verwaltung von OM1 Syntaxbäumen als Metadaten.

Von Bedeutung für die klassenspezifischen Querreferenzen in den verschiedenen Repräsentationen ist der gemeinsame Bezug, der über den *Namen* der spezifizierten OM1 Klasse hergestellt wird. In der Graphik ist es der Name des Klassenknotens, der in der Klasseninstanz der NeWS Klasse *ClassCircleItem* repräsentiert ist. In den Graphikmetadaten sowie in den Repräsentationen der OM1 Syntaxbäume sind es Tupelkomponenten der TL Typen (*GraphRep.ClassNode.name* und *OClass.Class.classId*), die in der Metadatenverwaltung als Zugriffsschlüssel dienen.

5.2 Einbindung des Graphikeditors in die STYLE Umgebung

Dieser Abschnitt befaßt sich mit der Integration des Graphikeditors als Modellierungswerkzeug in die STYLE Umgebung. Die STYLE Werkzeuge werden im Abschnitt 1.2 beschrieben. Dort werden auch die verschiedenen Schritte und Alternativen der interaktiven Werkzeugauswahl spezifiziert.

In den folgenden beiden Abschnitten werden im Wesentlichen zwei Schnittstellen vorgestellt. Die erste wird für den Aufruf des Graphikeditors vom *StyleTop* Editor aus, die zweite wird für den Aufruf des Klasseneditors vom Graphikeditor aus verwendet. Anhand der Schnittstellen soll gezeigt werden, wie einfach durch deren Verwendung die Integration von Werkzeugen der STYLE Umgebung zu realisieren ist.

Am Ende, in Abschnitt 5.3, folgt ein Beispiel eines Wechsels der graphischen zur textuellen Modellierung in der OM1 Entwicklungsumgebung. Die nötigen Funktionsaufrufe zur Generierung von textueller aus graphischer Modellierung und zum Werkzeugwechsel werden aufgelistet. Die Funktionalität wird graphisch illustriert.

5.2.1 Interaktion zwischen StyleTop- und Graphikeditor

Vor der Beschreibung der Schnittstelle zum Aufruf des Graphikeditors werden zunächst die Benutzerinteraktionen, die zum Öffnen des Graphikeditors führen, genannt. Nach der Schemaauswahl durch den Benutzer im *StyleTop Editor* wird über den weiteren Menüpunkt *open graphics* das Hauptfenster des Graphikeditors geöffnet (vgl. Abschnitt 3.3). Die TL Funktion

[8] aus der Tycoon Bibliothek *bulkenv*

openSchema(schemaName, pathName :String) :**Ok**

ist im Modul *graphEditor* definiert. Sie wird für den Aufruf des Graphikeditors verwendet. Die Parameter *schemaName* und *pathName* spezifizieren den Namen des zu editierenden Schemas sowie den Pfadnamen, welcher angibt, in welchem Verzeichnis die Datei mit den Graphikmetadaten des Schemas zu finden ist. Der Aufruf dieser Funktionen ist an den entsprechenden Menüpunkt des *StyleTop* Editors gebunden. Die Anbindung von Funktionen an Menüpunkte wird in Abschnitt 4.4.2 behandelt.

5.2.2 Interaktion zwischen Graphik- und Klasseneditor

In diesem Abschnitt werden die Benutzeraktionen für den Aufruf des Klasseneditors vom Graphikeditor aus aufgeführt. Dann folgt eine Beschreibung der Schnittstelle, die den Aufruf eines Klasseneditors realisiert. Am Ende werden die Transformationen und die sie realisierenden Dienste beschrieben, die erforderlich sind, um von der graphischen Modellierung einer OM1 Klasse in die textuelle Modellierung zu weseln – unter Wahrung der Konsistenz der Sichten beider Modellierungen auf die Metadaten.

Vom Graphikeditor aus gibt es die Möglichkeit, zu einem interaktiv selektierten Klassenknoten im OM1 Referenzdiagramm einen Klasseneditor aufzurufen, der die aus der Graphik generierte textuelle Beschreibung der korrespondierenden Klasse in OM1 Textsyntax enthält. Dafür ist in dem dem Klassenknoten zugeordneten Menü der Menüpunkt *Open Classeditor* auszuwählen, wonach ein Klasseneditor mit der korrespondierenden OM1 Klassendefinition als Inhalt geöffnet wird (vgl. Abschnitt 3.3). Als Schnittstelle für den Aufruf des Klasseneditors dient die im Modul *classEditor* definierte Funktion

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

(vgl. [Kas94]). Der Parameter *path* wird nicht näher erläutert.[9] Die Parameter *desiredClass* und *schemaName* geben den Namen der Klasse und des Schemas an. Sie dienen zum Zugriff der Klasse in den OM1 Metadaten. Es sei betont, daß die OM1 Metadaten, in denen u.a. die OM1 Klassendefinitionen durch Syntaxbäume persistent gespeichert sind, eine *gemeinsame Datenbasis* für Graphik- und Klasseneditor darstellen und beim Wechsel vom Graphikeditor in den Klasseneditor konsistent zu halten sind (vgl. auch Abschnitt 5.1.3). Ein in der Graphik angezeigter Klassenknoten muß, bevor er mit dem Klasseneditor editiert werden kann, in den OM1 Metadaten vorhanden sein.

5.3 Beispiel eines Werkzeugwechsels

In diesem Abschnitt werden die für einen konsistenten Übergang von graphischer zu textueller Modellierung erforderlichen Transformationen und die sie realisierenden TL Funktionen sowie die für den Werkzeugwechsel erforderlichen TL Funktionsaufrufe Softwarewerkzeuge in Zusammenhang gebracht. Damit wird eine Zusammenfassung der Generierungsfunktionen (vgl. Abschnitt 5.1.4) und der auf verschiedenen Repräsentationen aufsetzenden Werkzeugschnittstellen (vgl. vorigen Abschnitt) gegeben.

Für die Beschreibung des Übergangs von graphischer in textuelle Modellierung wird beispielhaft folgendes Szenario angenommen. Durch den Graphikeditor wird im Referenzdiagramm des

[9] Der darin spezifizierte Pfad wird nur verwendet, wenn eine Klasse **nicht** aus den Metadaten geladen wird oder nicht in Metadaten gesichert werden soll.

Schemas TRACY ein neuer Klassenknoten mit dem Namen *CarService* erzeugt, der eine OM1 Klasse graphisch repräsentiert. Es besteht der Wunsch, die OM1 Klasse textuell mit einem Klasseneditor weiter zu bearbeiten.

Über vom Benutzer betätigte Menüpunkte des Graphikeditors werden intern Transformationen ausgelöst, die gewährleisten, daß für den neu editierten Klassenknoten genau eine OM1 Klassenrepräsentation in den OM1 Metadaten erzeugt wird. Die folgende Auflistung beschreibt dazu die Abfolge der Transformationsschritte unter Angabe der Signaturen der sie realisierenden TL Funktionen.

Sichern der Graphik in Graphikmetadaten: Es wird unter anderem der neu erzeugte Klassenknoten (mit eventuell vorhandenen Beziehungen und Werteattributen) in den Graphikmetadaten gesichert. Die Signatur der verwendeten TL Funktion zur Sicherung aller Klassenknoten ist *setGRTClasses* (vgl. Abschnitt 5.1.2.4).

Abfrage des Klassennamens: Der Name des selektierten Klassenknotens (*CarService*) wird über den Aufruf der TL Funktion *getCurrClassNodeLabel(circPan)* des Schnittstellenmoduls *circlePanel*. Intern wird eine Methode einer NeWS Klasseninstanz ausgeführt (vgl. Abschnitt 4.3.1 und 4.3.3).

Aufsuchen der Repräsentation des Klassenknotens: Die auf Graphikmetadaten arbeitende Anfragefunktion *getClassNodeByLabel* gibt über den Klassennamen (*CarService*) die Repräsentation eines Klassenknotens aus den Graphikmetadaten des Schemas *Tracy* zurück (vgl. Abschnitt 5.1.2).

Generierung einer OM1 Syntaxrepräsentation der Klasse: Aus der vorher erhaltenen Graphikmetadatenrepräsentation des Klassenknotens wird – unter Verwendung weiterer Schemainformationen[10] eine Repräsentation der OM1 Klasse als Wert eines OM1 Syntaxbaums erzeugt. Dies erfolgt durch den Aufruf der Funktion *genOClass* (vgl. Abschnitt 5.1.4.1), die den Klassenknoten *CarService* und das Schema *Tracy* aus den Graphikmetadaten als aktuelle Parameter hat.

Einfügen der Generierung in die Metadaten: Schließlich wird die vorher erzeugte Repräsentation der OM1 Klasse in die Metadaten eingefügt. Dies geschieht durch den Aufruf der im Modul *omTransform* der Bibliothek *om1metaenv* definierten Funktionen.

Aufruf des Klasseneditors: Nach der Durchführung der Transformationsschritte kann der Klasseneditor durch den Funktionsaufruf *classEditor.filled(... "CarService" "Tracy")* mit der gerade neu erzeugten OM1 Klasse ("CarService") geöffnet werden (vgl. Abschnitt 5.2.2).

[10] Dies sind auf den Klassenknoten bezogene referentielle und Vererbungsbeziehungen sowie Werteattribute

Kapitel 6

Zusammenfassung und Ausblick

Dieses Kapitel faßt die Ergebnisse dieser Arbeit zusammen und wertet sie aus. Das Ziel der Arbeit ist, ein systematisches Vorgehen für die Realisierung eines objektorientierten Graphikeditors in Tycoon zu beschreiben. In den Kapiteln 1 bis 3 werden Anforderungen an die Realisierung bezüglich der Unterstützung des exemplarisch gewählten OM1 Graphikmodells, der Gestaltung der graphischen Benutzerschnittstelle und der Integration des Editors in die OM1 Entwicklungsumgebung formuliert. Die im Rahmen dieser Arbeit erfolgte Realisierung und Integration des Graphikeditors werden in den Kapiteln 4 und 5 behandelt.

Die Abschnitte 6.1 und 6.2 fassen das Vorgehen zur Realisierung und zur Integration des Editors in die STYLE Umgebung zusammen. Die Ergebnisse der Arbeit werden bezüglich der in Kapitel 1 formulierten Anforderungen bewertet. Grenzen und Schwierigkeiten bei der Benutzung von Werkzeugen und Diensten in Tycoon werden diskutiert. Der Abschnitt 6.3 gibt einen Ausblick auf mögliche Erweiterungen des Graphikeditors bezüglich der Konzepte der OM1 Graphik, der Integration mit anderen Werkzeugen in STYLE sowie der Unterstützung weiterer Datenmodelle.

6.1 Realisierung des Graphikeditors

Die zu realisierende übersichtliche Diagrammgestaltung in der graphischen Modellierung basiert auf dem in Abschnitt 2.3.2 spezifizierten OM1 Graphikmodell, welches die zu erstellenden Diagrammarten beschreibt. Die angewendeten Visualisierungstechniken in Form von visuell gewichteten Symbolen, gerastertem Layout, verschiedenen Sichten und Separierung von Information in verschiedene Diagrammarten berücksichtigen Gesetze der menschlichen Wahrnehmung von graphischer Information. Das OM1 Graphikmodell führt zu übersichtlichen und semantisch gewichteten Diagrammen. Sie stellen eine adäquate Repräsentation von Konzepten des objektorientierten Datenmodells OM1 dar. Die Modellierung großer Anwendungen stößt in der graphischen Darstellbarkeit in den Diagrammen auf Grenzen. Eine Möglichkeit, diese Grenzen aufzuheben, ist die Modularisierung von Schemata [LM92] [Tha91b].

Das Design des Graphikeditors (vgl. Kapitel 3) befolgt die Stilrichtlinien (*style guides*) des OPEN LOOK Standards für das Design und das interaktive Verhalten von Komponenten der Benutzerschnittstelle. Die Dialoggestaltung ist so gewählt, daß die vom Graphikmodell vorgegebene Art der graphischen Modellierung unterstützt wird. Wahlmöglichkeiten zwischen menügesteuerter und direkter Manipulation beim Editieren von Diagrammen tragen zu einer flexiblen Handhabung des Editors bei. Die Einhaltung des Graphikstandards und eine Dialogführung, die Alternativen zwischen verschiedenen Dialogtechniken anbietet, erhöhen die Benutzbarkeit

des Graphikeditors.
Bei der Realisierung des Graphikeditors (vgl. Kapitel 4) wird der Schwerpunkt auf systematische Erweiterung und typsichere Einbindung von externen Bildschirmdiensten gelegt. Zu verfolgende Ziele sind Effizienz in der Verwaltung und Manipulation von komplexen Diagrammobjekten sowie Wiederverwendbarkeit und Erweiterbarkeit von Bausteinen.

In den zur Verfügung stehenden Diensten sind der Entwicklung neuer Bildschirmsymbole Grenzen gesetzt. Die Funktionalität der Tycoon Bibliothek *newsenv* reicht nicht aus, um in TL von OPEN LOOK abweichende, anwendungsspezifische Bildschirmsymbole zu erzeugen. Da auch außerhalb der Tycoon Umgebung keine Werkzeuge zur generischen Erzeugung von beliebigen Diagrammarten existieren, war ein investiver Aufwand für die Realisierung von OM1 Diagrammen erforderlich. Es mußten neue NeWS Klassen im Werkzeugkasten TNT implementiert werden. Das Vorgehen bei der Entwicklung neuer Klassenkonzepte wurde durch die Anwendung von effizienten Verfahrensweisen systematisiert. Die Verfahrensweisen sind Benutzung von Vererbungsmechanismen, Modularisierung von Konzepten nach dem Leitbild der Werkzeug-Material Metapher und Dekomposition komplexer Graphikobjekte. Die Wiederverwendbarkeit generischer Bausteine zur Unterstützung der graphischen Repräsentation weiterer Datenmodelle wird im Abschnitt 6.3.3 beispielhaft belegt.

Die Einbindung der definierten Graphikfunktionalität in TL erfolgt durch die Erweiterung der Tycoon Bibliothek *newsenv*. Damit wird ein Beitrag zur typsicheren, von Kommunikationsdiensten abstrahierenden, Verwaltung von OM1 Diagrammen geleistet. Die Bibliothekserweiterung kann durch vorhandene Basisdienste des *newsenv* effizient realisiert werden.

Eine noch ausstehende Aufgabe ist die Evaluierung des Graphikeditors bezüglich der Benutzerfreundlichkeit (vgl. Kapitel 3). Mit Hilfe von Evaluationsverfahren (vgl. [Obe92]) kann eine Qualitätsprüfung des Prototypen durchgeführt werden. Ein mögliches Evaluationsverfahren besteht in der Beobachtung von Benutzern bei der Arbeit am Prototypen des Graphikeditors und in der Auswertung der gewonnenen Ergebnisse anhand verschiedener Kriterien der Benutzerfreundlichkeit. Die Aufgabe der Evaluierung liegt außerhalb des Rahmens dieser Arbeit.

6.2 Integration in die STYLE Umgebung

Die Integration betrifft den Wechsel von graphischer in textuelle Modellierung und die Werkzeugeinbindung in die graphische Benutzerschnittstelle der OM1 Entwicklungsumgebung. Die Realisierung der Integration bezüglich graphischer und textueller Modellierung wird durch Transformationsfunktionen auf dafür definierten TL Repräsentationen erreicht. Eine Voraussetzung für die Integration ist die Möglichkeit der Datengewinnung aus angezeigten Graphiken und deren Umformung in effizient verwaltbare TL Repräsentationen. Die TL Funktionen der erweiterten Bibliothek *newsenv* gestatten einen effizienten Datenzugriff auf Bildschirmgraphik. Weitere Tycoon Bibliotheken stellen generische Massendatentypen, wie Listen, Mengen und Wörterbücher, und Iterationsabstraktionen bereit. Sie unterstützen die Verwaltung von komplexen Entwurfsobjekten in TL Repräsentationen und die darauf operierenden Anfrage- und Generierungsdienste.

Die gewählte Methodik, von werkzeugspezifischen Datenstrukturen zu abstrahieren, indem diese typsicher in Repräsentationen des uniformen TL Sprachraums transformiert werden, erleichtert die Integration von graphischer und textueller Modellierung und die damit verbundene Imple-

mentierung von Generierungsfunktionalität.

Die Integration des Graphikeditors in die Benutzeroberfläche der OM1 Entwicklungsumgebung wird durch Funktions- und Modulabstraktion in Tycoon unterstützt. Die Interaktion von Graphik- und Texteditor erfolgt durch Aufruf entsprechender Funktionen.

6.3 Ausblick

Dieser Abschnitt befaßt sich mit weiteren Anwendungen von Graphikmodellierung in datenintensiven Entwicklungsumgebungen. Die weiteren Anwendungen werden in folgende Themengebiete unterteilt.

- Die Visualisierung weiterer semantischer Konzepte innerhalb des vorgegebenen Datenmodells;
- die Integration von graphischer und textueller Modellierung;
- die Werkzeugflexibilität hinsichtlich Realisierung graphischer Modellierung in anderen Datenmodellen und datenmodellübergreifende Anwendungsmöglichkeiten von Visualisierungstechniken.

Die Themengebiete werden in den Abschnitten 6.3.1 bis 6.3.3 behandelt. In Abschnitt 6.3.4 werden Fragen der Portabilität des Graphikeditors erörtert.

6.3.1 Visualisierungen von OM1 Konzepten

Wie in Abschnitt 2.3.2 erwähnt, existiert nicht für jedes sprachliche Konzept von OM1 eine Visualisierung in der OM1 Graphik. Dafür gibt es folgende Gründe.

1. Visualisierungen sind spezifiziert, aber in der derzeitigen, prototypischen Version des Graphikeditors nicht implementiert.
2. Bestimmte OM1 Konzepte sind nur schwer oder gar nicht visualisierbar.
3. Für einige OM1 Konzepte sind Visualisierungen prinzipiell denkbar, aber noch nicht spezifiziert.

Es folgen Beispiele und Erklärungen zu noch nicht realisierten Visualisierungen. Bereits spezifizierte OM1 Visualisierungen sind Darstellungen von geschachtelten Referenzen bis zu einer bestimmten Schachtelungstiefe sowie geschachtelte Werteattribute (vgl. [LM92]). Die Beispiele beziehen sich auf eine Universitätsanwendung [Tha91a].

Im Beispiel der Abb. 6.1 wird graphisch modelliert, daß jeder Student in einer Menge von Vorlesungen eingeschrieben ist und einen Professor als Betreuer hat. Für jede Einschreibung ist ein Ergebnis vorhanden, der Betreuung ist ein Anfangsdatum zugeordnet. Abb. 6.2 zeigt die Visualierung von in Tupeln geschachtelten Wertetypen.

Benutzerdefinierte Integritätsbedingungen und Klassenmethoden lassen sich wegen ihrer komplexen Syntax im allgemeinen nicht visualisieren. Hinzu kommt, daß Vor- und Nachbedingungen von Methoden sowie die in ihnen definierten Zustandsänderungen eine *dynamische* Semantik besitzen, deren Modellierung den Rahmen eines die *Datenstruktur* modellierenden

Diagramms überschreitet. Eine denkbare Form der Methodenvisualisierung sind Zustandsübergangsdiagramme.

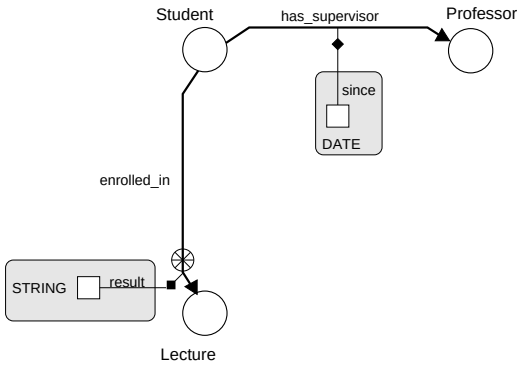


Abbildung 6.1: Graphische Repräsentation von geschachtelten Referenzen

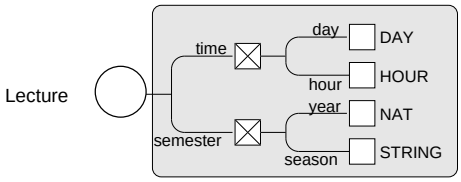


Abbildung 6.2: Graphische Repräsentation von geschachtelten Werteattributen

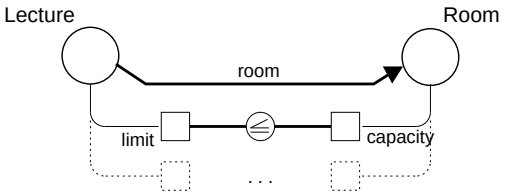


Abbildung 6.3: Visualisierung von Integritätsbedingungen zwischen klassenübergreifenden Attributwerten

Ein Ansatz, eine bestimmte Klasse von Integritätsbedingungen zu visualisieren, ist in Abb. 6.3 dargestellt. Dort werden Integritätsbedingungen, die über Ordnungsrelationen zwischen klassenübergreifenden Attributwerten definiert sind, visualisiert.

Für bestimmte OM1 Schlüsselwörter, die Attributeigenschaften näher spezifizieren, z.B. ***constant*** und ***derived*** (vgl. Abschnitt 2.3), fehlen bislang Symbole zu deren Visualisierung. Sie sind mit geringem Aufwand, wie das bereits vorhandene Mengensymbol, in der graphischen Repräsentation realisierbar.

6.3.2 Integration von graphischer und textueller Modellierung

In der gegenwärtigen Version ist die Generierung von Graphik aus textueller OM1 Modellierung nicht verwirklicht. Die Aufgabe, für ein ganzes Schema automatisch eine OM1 Graphik zu erzeugen, stößt algorithmisch an Grenzen. Die Schwierigkeit der Aufgabe liegt darin, eine Graphik mit übersichtlich und überschneidungsfrei angeordneten Diagrammobjekten zu generieren. Weniger komplex ist die Erzeugung von Schemaausschnitten. Beispielsweise könnte zu einer textuellen OM1 Klassendefinition in Form eines Referenzdiagramms eine Visualisierung der Klasse mit direkten referentiellen Beziehungen zu weiteren Klassen erzeugt und angezeigt werden. TL Repräsentationstypen für OM1 Text- und Graphikdaten sind vorhanden. Die Generierungsfunktionalität ist durch Transformationsfunktionen, die eine TL Repräsentation in die andere umformen, ohne großen Aufwand zu realisieren.

6.3.3 Flexible Anbindung weiterer Datenmodelle

Die in Abschnitt 4.2 eingeführte Erweiterung der Klassenhierarchie des TNT Werkzeugkastens ist durch Anwendung der Vererbung als Methode leicht um neue Bausteine ergänzt werden, die Diagrammarten mit anderen Symbolen enthalten. Es existieren bereits generische Klassen zur Verwaltung von beliebigen Diagrammen (vgl. Abschnitt 4.2.3.), die in Subklassen mit neuen Komponentenarten instantiiert werden können. Die typsichere Anbindung der neu erzeugten Klassen und Methoden des TNT Dienstes in die Tycoon Umgebung, unter Verwendung und Erweiterung der Bibliothek *newsenv*, ermöglicht es, Diagrammformen explizit über TL Funktionen zu setzen. In einer flexiblen Entwicklungsumgebung könnten beispielsweise über Menüs die Diagrammformen vom Benutzer wählbar sein.

Es folgt ein Kodebeispiel, an dem der geringe Aufwand zur Definition neuer Symbolformen und Diagrammarten verdeutlicht wird. Es sollen statt kreisförmiger Knoten quadratische Knoten in einem gerasterten Diagramm verwendet werden.

```
/ClassRectangle ClassCircle
  classbegin
    /path
       rectpath
     def
    /StrokeCanvas
       StrokeCanvas ClassCanvas send
     def
  classend def

/ClassRectanglePanel ClassCirclePanel[]
  classbegin
    /ItemClass ClassRectangle def
  classend def
```

In der neuen TNT Klasse *ClassRectangle*, die Subklasse der kreisdefinierenden Klasse *ClassCircle* ist, werden die geerbten Methoden */path* und */StrokeCanvas*, die die Umrandung und die Zeichenprozedur definieren, überschrieben. Statt eines Kreises wird eine rechteckige Form definiert. Die Klasse *ClassRectanglePanel* erbt alle Eigenschaften des Kreisdiagramms (z.B. Selektierbarkeit von Knoten) und instantiiert über den Namen */ItemClass* die Rechteck-

klasse *ClassRectangle* als die in ihr verwaltete Klientenart.

Ferner besteht die Möglichkeit, von Tycoon aus über Funktionsaufrufe des *newsenv* Eigenschaften zu setzen und Methoden zu überschreiben. Im folgenden Kodebeispiel wird der Name der kreisdefinierenden Klasse *ClassCircle* an einen Bezeichner in Tycoon gebunden und die Methoden *path* und *StrokeCanvas* in der Klasse überschrieben.

```
let circle = object.bind("ClassCircle")
object.installMethod(circle "path" "rectpath")
object.installMethod(circle "StrokeCanvas" "StrokeCanvas ClassCanvas send")
```

Das Resultat beider Kodebeispiele – der neu erzeugten Subklassen in TNT oder des alternativen Überschreibens von Klassenmethoden über TL Funktionen – ist in Abbildung 6.4 beispielhaft veranschaulicht. Statt Kreisen werden Rechtecke im Diagramm dargestelt. Dasselbe Vorgehen ist auch zur Definition neuer Kantenarten verwendbar, um beispielsweise Beziehungskanten für ER Diagramme zu erzeugen.

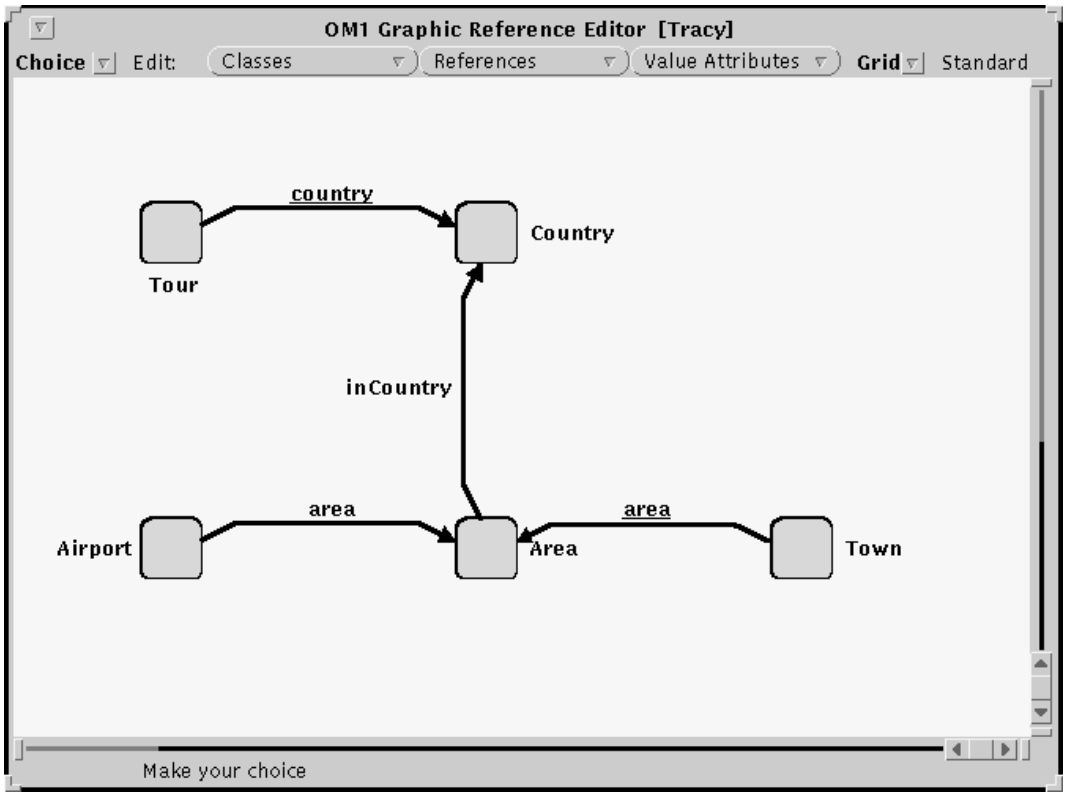


Abbildung 6.4: Graphische Modellierung mit geänderten Klassensymbolen

Es ist denkbar, die für die OM1 Graphik verwendeten Visualisierungstechniken, wie Rasterung, Anzeige von Diagrammsichten mit verschiedener Granularität der Details und Separierung von Graphik in verschiedenen Diagrammen (vgl. Abschnitt 2.3.2), auch für die übersichtliche Darstellung anderer Diagrammarten zu verwenden. In ER Diagrammen könnten z.B. die Attribute ein- und ausblendbar sein. Im HERM würde eine Separierung verschiedener Beziehungsarten (Klassenreferenzen, Beziehungen höherer Ordnung und Subklassenbeziehungen) in unterschiedliche Diagramme ein semantisches Überladen reduzieren (vgl. Abschnitt 2.1).

6.3.4 Portabilität

Das NeWS Fenstersystem und der darauf aufsetzende Werkzeugkasten TNT sind unabhängig von Hardwareplattformen und Betriebssystemen und deshalb grundsätzlich portabel. Da NeWS aber außer von Sun Microsystems von keiner weiteren Firma unterstützt wird, ist eine Portabilität nicht gegeben. Während der Realisierungsphase der vorliegenden Arbeit war nicht abzusehen, daß NeWS und TNT beim Erscheinen des neuen Betriebssystems Solaris 2.3 nicht mehr von der Firma Sun Microsystems unterstützt werden. NeWS, welches durch seine Erweiterbarkeit konzeptuelle Vorteile gegenüber dem X11 Fensterprotokoll hat, war in der Entwicklungsphase der Bibliothek *newsenv* [Mü94b], die die Grundlage für die Anbindung von externen Bildschirmdiensten für diese Arbeit darstellt, die geeignetere Wahl.

Das StarView Toolkit [BKJ93] bildet einen systemübergreifenden Werkzeugkasten, der, ähnlich dem TNT, eine erweiterbare Klassenhierarchie für die Erzeugung von graphischen Benutzerschnittstellen anbietet. Dienste dieses externen Werkzeugs werden zur Zeit im Rahmen einer Diplomarbeit [Gei94] durch eine Bibliothek in Tycoon angebunden. Inwieweit StarView geeignet ist, anwendungsspezifische Symbole mit speziellen Formen zu definieren, ist Gegenstand der derzeitigen Untersuchung. Es ist geplant, Tycoon auf andere Plattformen zu portieren. Eine zukünftige Verwendung von StarView im Rahmen eines portablen Tycoon Entwicklungssystems würde die Portabilität von graphischen Benutzerschnittstellen wesentlich erhöhen.

Anhang A

Modellierung

A.1 OM1 Grammatik

Der hier dargestellte Ausschnitt der OM1 Grammatik liegt der Entwicklung des OM1 Datenbankprototypen im Rahmen des STYLE Projekts zugrunde.

schemaDef ::= **Schema** *schemaId*
[*typeDefList*] *classDefList*
End *schemaId*

typeDefList ::= typeDef {*typeDef*}

classDefList ::= classDef {*classDef*}
schemaId ::= *String*

A.1.1 Typen

typeDef ::= **Type** *typeBinding*
typeBinding ::= *typeBind* | **rec** *typeBind* {**and** *typeBind*}

typeBind ::= *typeId* "=" *typeExpr*
typeExpr ::= *typeId* | *typeConsAppl* | *baseType*

baseType ::= **Bool** | **Int** | **String** | **Ok**

typeConsAppl ::= **Record** *compList* **end** |
SetOf "(" *typeExpr* ")" |
ListOf "(" *typeExpr* ")" |
Array *typeExpr* **of** *typeExpr* **end**
Option *optList* **end**

compList ::= [*component* {"," *component*}]

optList ::= [*component* {"|" *component*}]

component ::= *label* ":" *typeExpr*

label ::= *String*

typeId ::= *String*

A.1.2 Klassen

classDef ::= **Class** *classId*
[**Specialization** *specList*]
[**Structure** *structure*]
[**Constraints** *constraints*]
End

A.1.2.1 Spezialisierung

specList ::= *specialization* {“,” *specialization*}
specialization ::= *specConsOp classId*

classId ::= *String*

specConsOp ::= **isA**

A.1.2.2 Struktur

structure ::= **Attributes** *attrList*

attrList ::= *attribute* {“,” *attribute* }

attribute ::= {*attrKind*} *label* “:” *structureExpr*

attrKind ::= **key** | **constant**

structureExpr ::= **ref** *classId* | *typeId* | *structureConsAppl* | *baseType*

structureConsAppl ::= **Record** *attrList* **end** |
SetOf “(” *structureExpr* “)” |
ListOf “(” *structureExpr* “)” |
Array *structureExpr* **of** *structureExpr* **end** |
Option *optStructList* **end**

optStructList ::= [*optComponent* {“|” *optComponent*}]

optComponent ::= *label* “:” *structureExpr*

A.1.2.3 Integritätsbedingungen

constraints ::= [**static** *consList*]

consList ::= *consName*":" *constraint* {"," *consName*":" *constraint*}

constraint ::= **forAll** *boundVarList* "|" *constraint* |
exists *boundVarList* "|" *constraint* |
constructedTerm

boundVarList ::= *varList* ":" *ident*

varList ::= *String* {"," String}

constructedTerm ::= *term* [*String term*]

term ::= *atomicTerm* | "(" *constraint* ")" | *ident* ["(" *constructedTermList*")"]

constructedTermList ::= [*constructedTerm* {"," constructedTerm}]

atomicTerm ::= **true** | **false** | **ok** |
record *bindingList* **end** |
setOf "(" *construction* ")" |
listOf "(" *construction*")" |
String | *Int*

construction ::= **each** *boundVarList* **where** *constraint*"|" *constructedTerm* | [*varList*]

bindingList ::= {*String* "=" *constructedTerm*}

ident ::= *String* [*selector*]

selector ::= "." *ident* | "[" *constructedTerm*"]"

A.2 Graphische Modellierung: TRACY Beispiel

Im folgenden wird die graphische Modellierung der Anwendung TRACY dargestellt. Das Referenzdiagramm wird in zwei Sichten präsentiert, gefolgt von einer Abbildung des Vererbungsdiagramms.

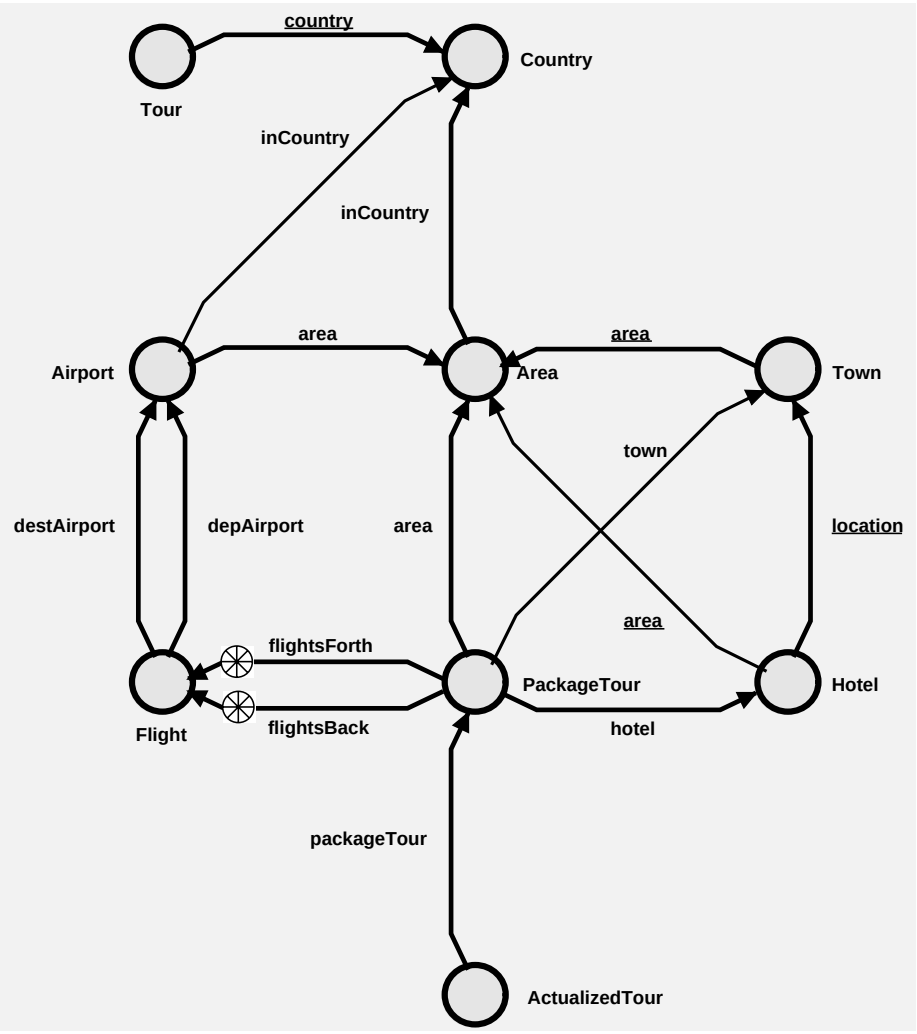


Abbildung A.1: Visualisierung der TRACY Anwendung in der Ebene 1 (Standardsicht)

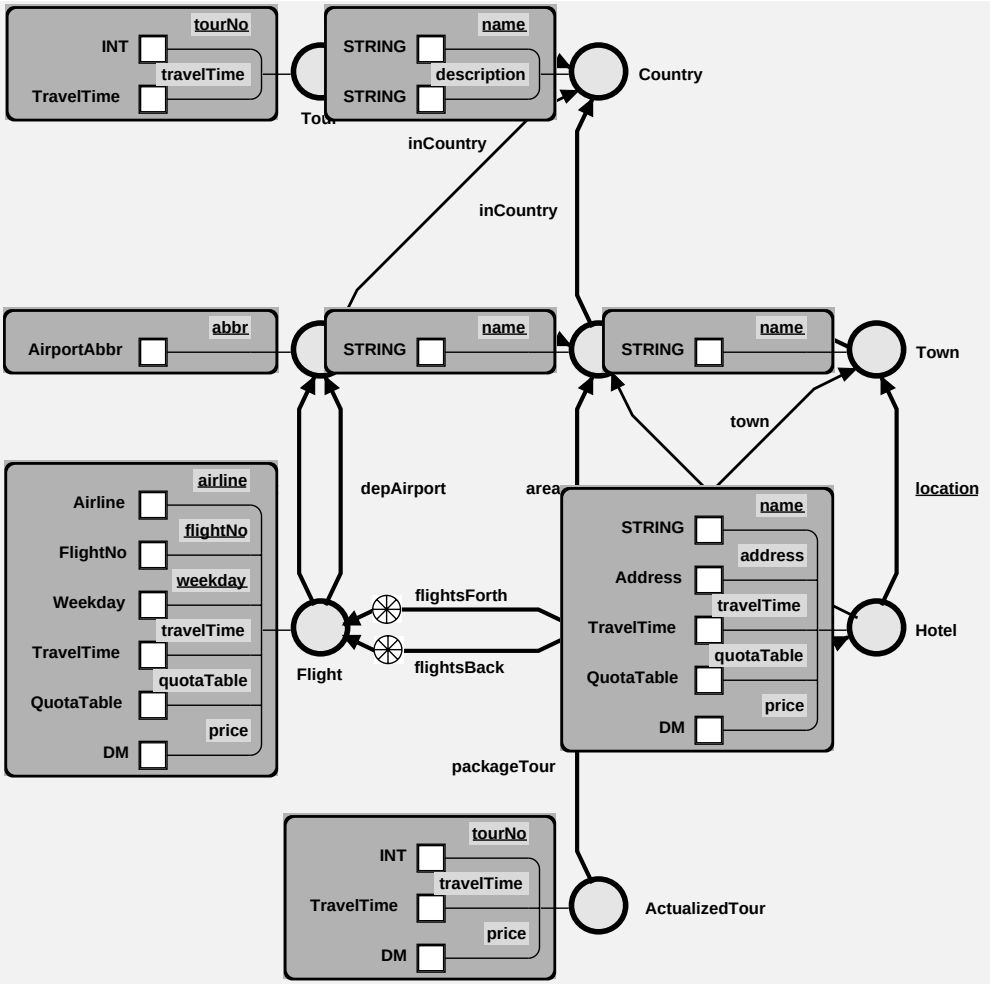


Abbildung A.2: Visualisierung des TRACY Anwendung in Ebene 3

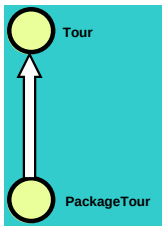


Abbildung A.3: Visualisierung der Vererbungsbeziehungen der TRACY Anwendung

Es folgt die Darstellung der Modellierung einer erweiterten TRACY Anwendung (vgl. [Koe94] als Referenz- und Vererbungsdiagramm.

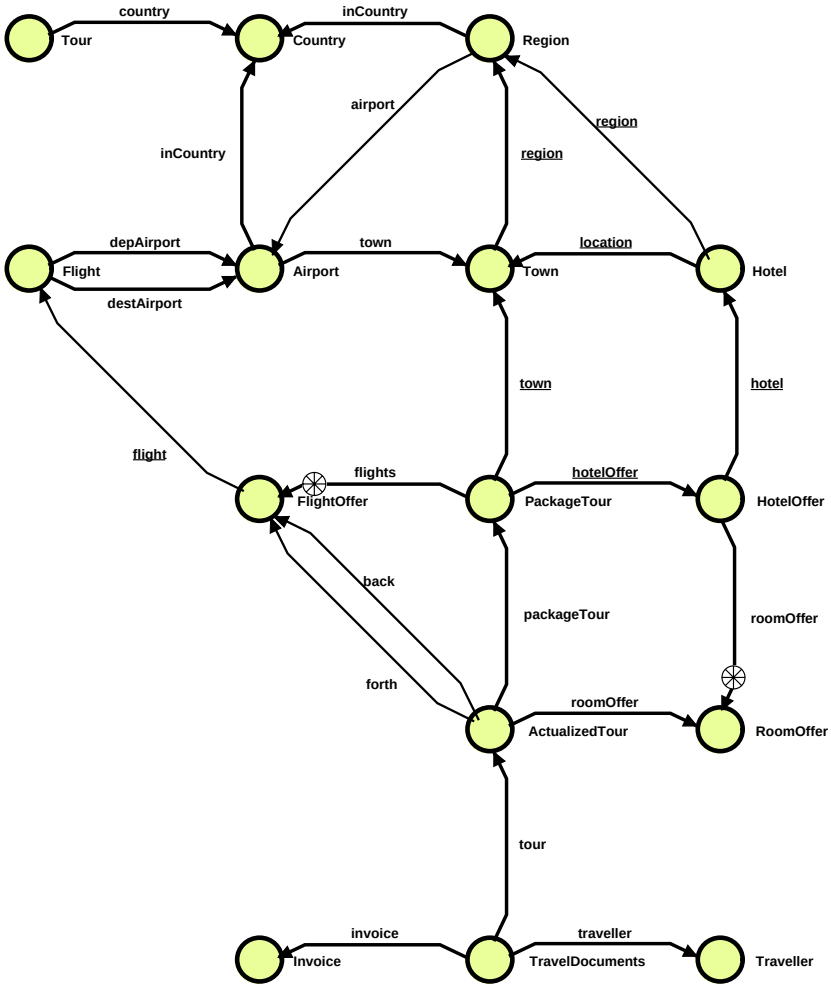


Abbildung A.4: Visualisierung der erweiterten TRACY Anwendung in der Ebene 1 (Standardsicht)

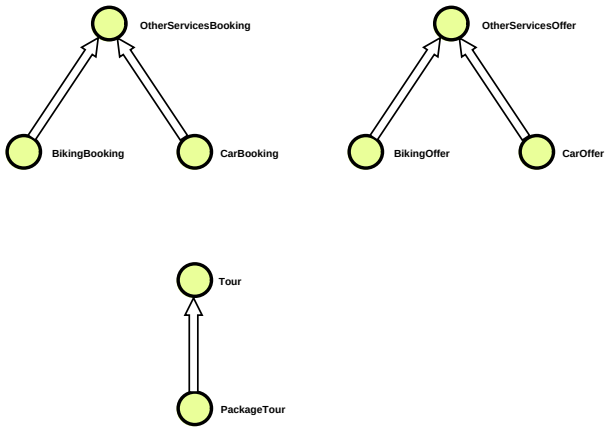


Abbildung A.5: Visualisierung der Vererbungsbeziehungen der erweiterten TRACY Anwendung

A.3 Textuelle Modellierung: TRACY Beispiel

Im folgenden wird die Anwendung TRACY textuell in OM1 Syntax spezifiziert. Die textuelle Modellierung der Anwendung enthält zusätzlich zur graphischen Modellierung benutzerdefinierte Integritätsbedingungen.

Schema TRACY

Type Address = **Record**
street :String,
zipCode :String,
city :String,
tel :String
end

Type Airline = String

Type AirportAbbr = String

Type Date = **Record** day :DayOfMonth, month :Month, year :Year **end**

Type DM = Int

Type DayOfMonth = Int

Type DayOfWeek = String

Type FlightNo = Int

Type Month = Int

Type Name = **Record** firstName :String, sirName :String **end**

Type ReservationRec = **Record** booked :Int, vacant :Int **end**

Type TravelTime = **Record** from :Date, till :Date **end**

Type Year = Int

A.3.0.4 Country

Class Country

Structure

Attributes
`key` name :String,
description :String

End

A.3.0.5 Area

Class Area

Structure

Attributes
`key` name :String,
inCountry :**ref** Country,
airport :**ref** Airport

Constraints

static
AreaCountry:`this`.airport.inCountry = `this`.inCountry
(* The airport must belong to the same country as the area *)

End

A.3.0.6 Town

Class Town

Structure

Attributes
`key` name :String,
`key` area :**ref** Area

End

A.3.0.7 Airport

Class Airport

Structure

Attributes
`key` abbr :AirportAbbr,
inCountry :**ref** Country,
area :**ref** Area

Constraints

static
AreaAirport:`this`.area.airport = `this`
(* the airport must be the same one as the area's *),
CountryArea:`this`.inCountry = `this`.area.inCountry
(* the airport's country must be the same on as the area's *)

End

A.3.0.8 Flight

Class Flight

Structure

Attributes
`key` airline :Airline,
`key` flightNo :FlightNo,
`key` weekday :DayOfWeek,
travelTime :TravelTime,
route :**Record**
depAirport :**ref** Airport,
destAirport :**ref** Airport
end,
quotaTable :**Array** Date **of** ReservationRec **end**
price :DM

End

A.3.0.9 Hotel

Class Hotel

Structure

Attributes
`key` name :String,
`key` area :**ref** Area,
`key` location :**ref** Town,
address :Address,
travelTime :TravelTime,
quotaTable :**Array** Date **of** ReservationRec **end**,
price :DM

End

A.3.0.10 Tour

Class Tour

Structure

Attributes
`key` tourNo :Int,
`constant key` country :**ref** Country,
travelTime :TravelTime

End

A.3.0.11 PackageTour

Class PackageTour

Specialization

isA Tour

Structure

Attributes

`key` area :**ref** Area,
`key` town :**ref** Town,
hotel :**ref** Hotel,
flights :**Record**
forth :**ref** Flight,
back :**ref** Flight
end

Constraints

static

HotelFlight:
(`this`.hotel.area = `this`.flights.forth.destAirport.area) ∧
(`this`.hotel.area = `this`.flights.back.depAirport.area)
(* the forth flight's destinationairport and the return flight's departure airport must belong to the same area as the hotel *),
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))),
(* the traveltime must be within the hotel's and flights' traveltime *)

End

A.3.0.12 ActualizedTour

Class ActualizedTour

Structure

Attributes

`key` tourNo :Int,
packageTour :**ref** PackageTour,
travelTime :TravelTime,
price :DM

Constraints

static

PriceOk:`this`.price =
(`this`.packageTour.hotel.price +
(`this`.packageTour.flights.forth.price +
`this`.packageTour.flights.back.price))
(* the price must match the sum of the hotel's and the flights' prices *)

End

A.4 OM1 Entwicklungsumgebung

Der Bildschirmausdruck in Abbildung A.6 zeigt den StyleTop Editor, den Schemabrowser sowie Typ- und Klasseneditoren mit geöffneten Fenstern für Standardanfragen.

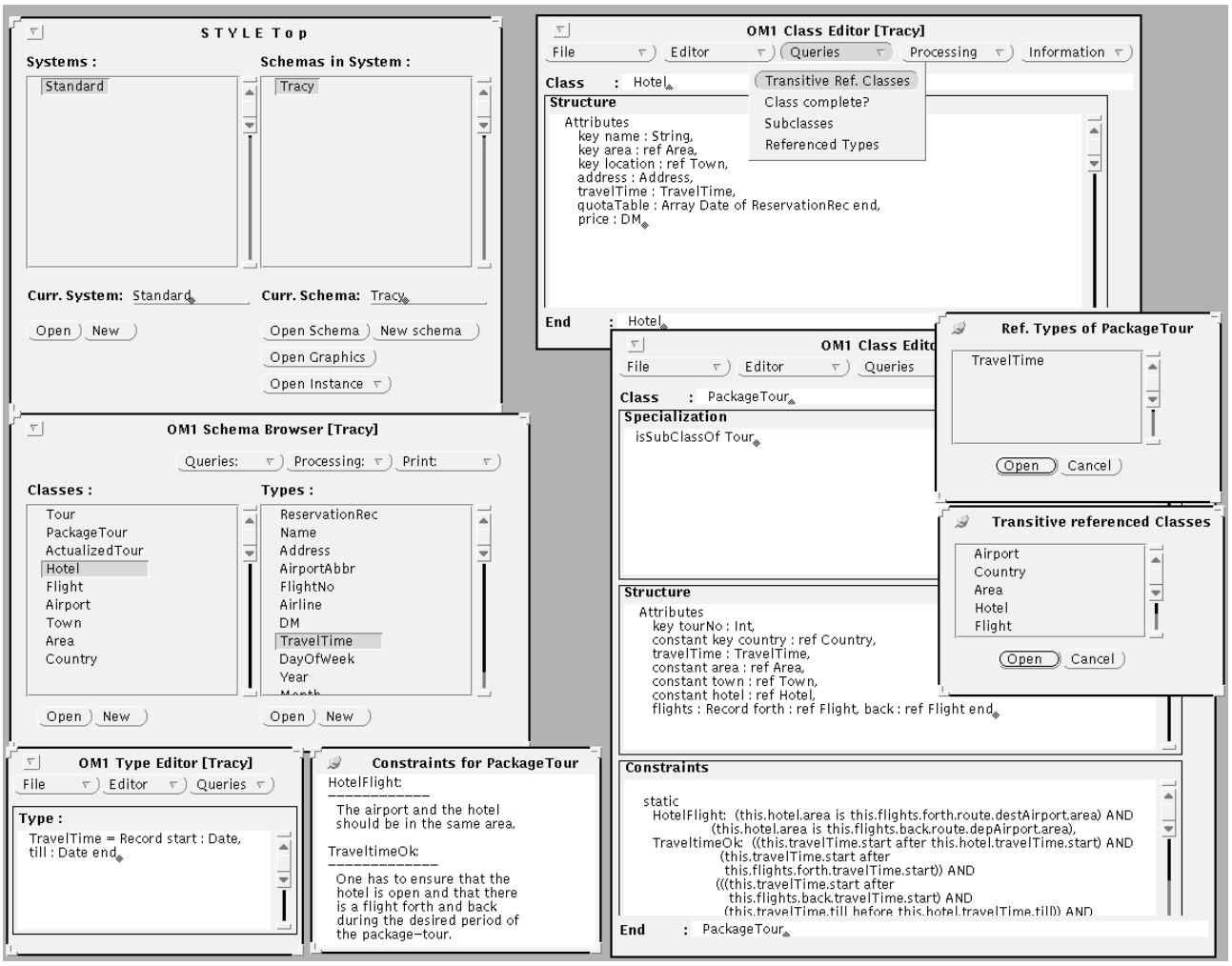


Abbildung A.6: Textuelle Modellierung in OM1

Der Bildschirmausdruck in Abbildung A.7 stellt ausschnittsweise generierte Schnittstellen und Module, den Instanzbrowser sowie Datenbankeditoren dar.

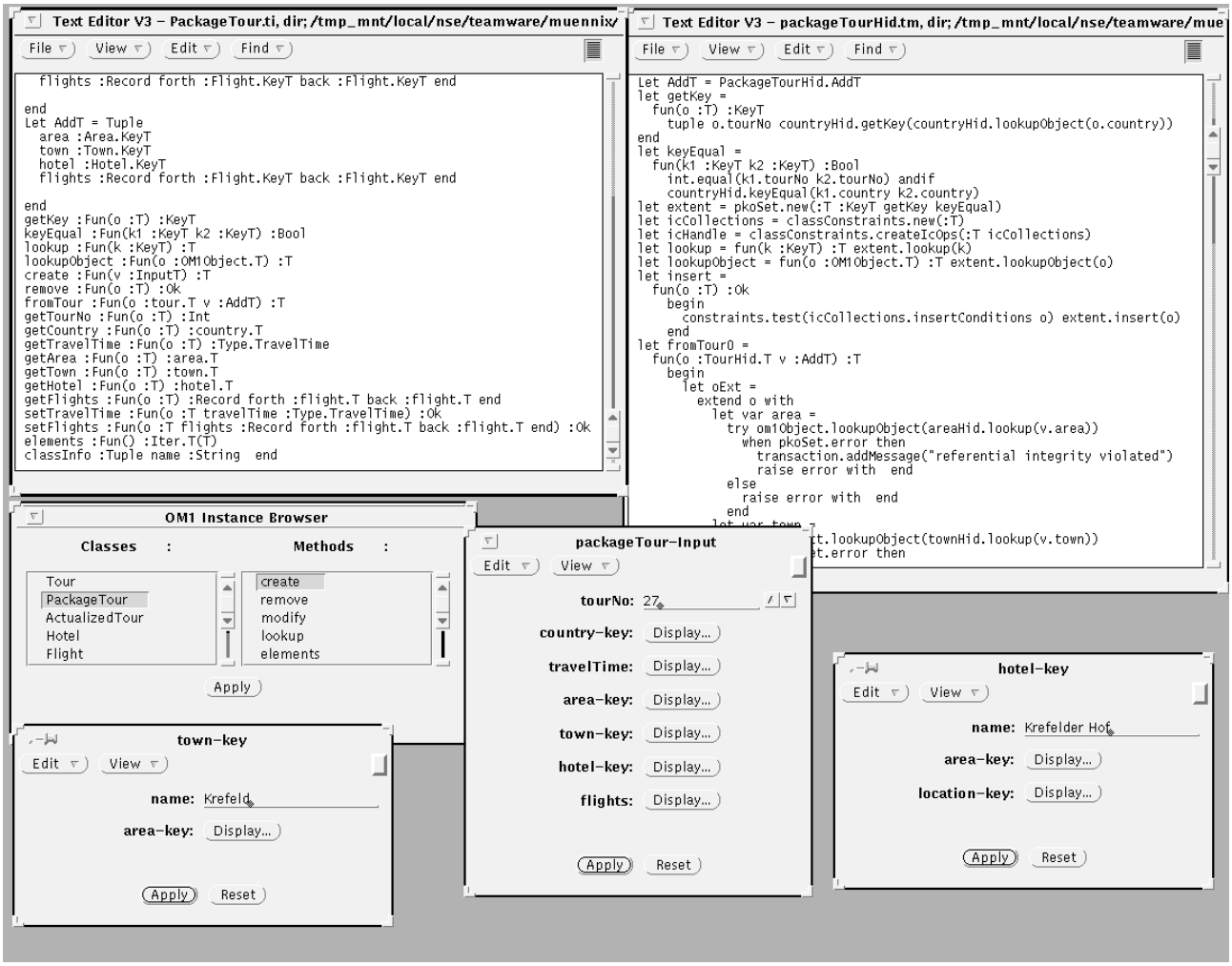


Abbildung A.7: Generierungsunterstützung: TL Schnittstellen, Module und Datenbankeditoren

Der Bildschirmausdruck in Abbildung A.8 veranschaulicht eine integrierte Sicht der Entwicklungsumgebung mit folgenden Komponenten.

- Geöffneter Graphik- und Texteditor auf der Modellierungsebene,
- generierte TL Schnittstelle und Modul zur Anwendungsprogrammierung,
- geöffneter Instanzbrowser sowie Datenbankeditor zur Objekterzeugung.

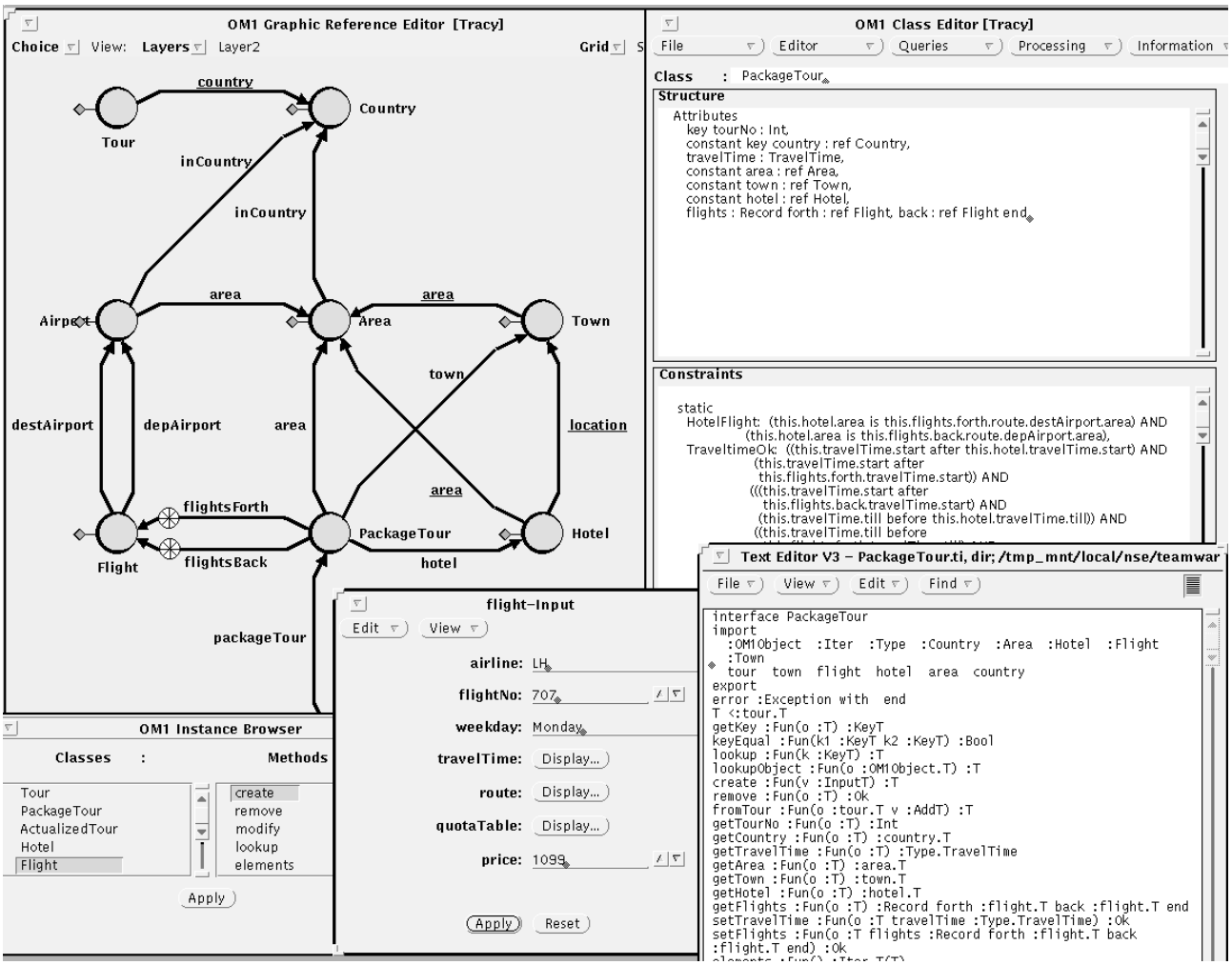


Abbildung A.8: Integrierte Sicht der OM1 Entwicklungsumgebung

Anhang B

Weitere Anwendungen

B.1 OM1 Diagramme mit erweiterter Rasterung

Es folgt ein Beispiel einer Gerichtsanwendung (aus [Tha91b]). Durch eine engere Anordnung werden semantisch zusammengehörige Klassen und Beziehungen als Gruppen visualisiert.

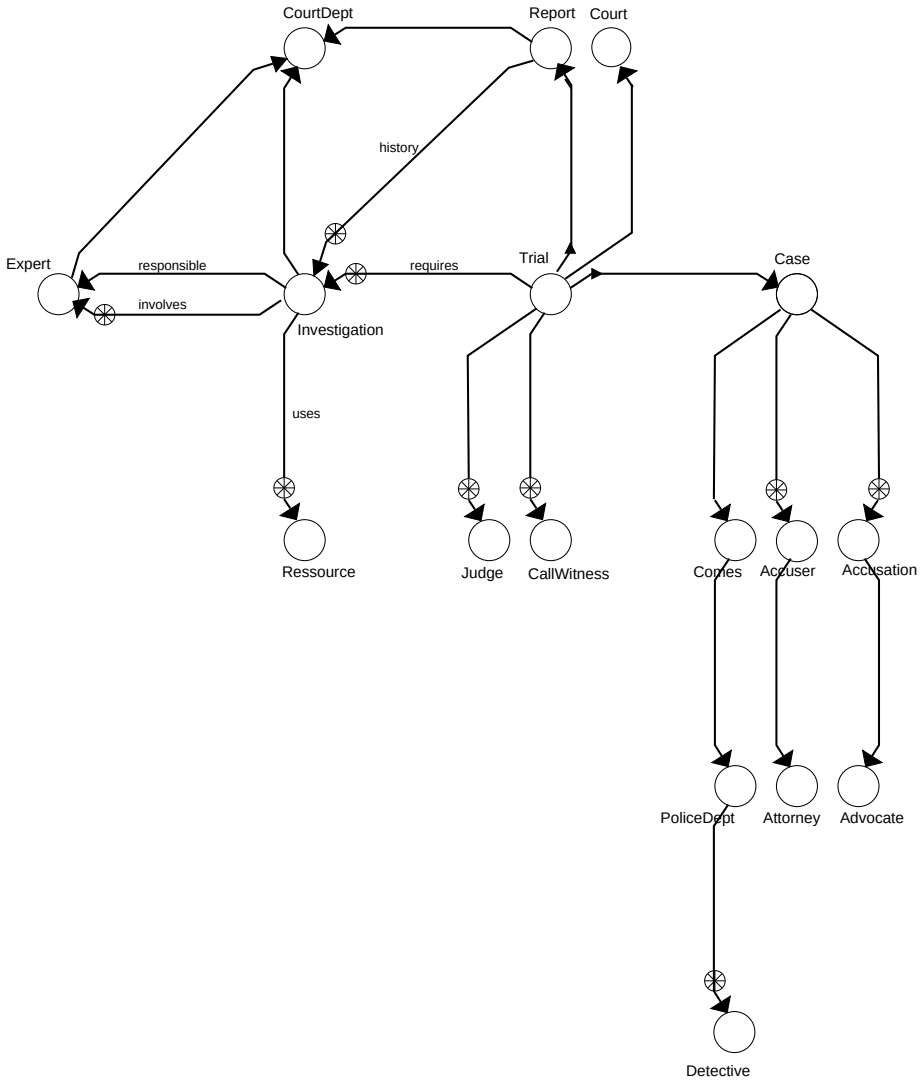


Abbildung B.1: Erweiterte Rasterung: Modellierung einer GerichtsanwendungWerteattribute

B.2 Overlay Techniken zur Aufhebung getrennter Diagrammdarstellungen

Die in den beiden folgenden Abbildungen modellierte Anwendung einer Universitätsanwendung ist aus [Tha91b] entnommen. Die erste Abbildung zeigt getrennte Referenz- und Vererbungsdiagramme. In der zweiten Darstellung wird das Referenzdiagramm durch folienartiges Auflegen von Vererbungsbeziehungen überlagert.

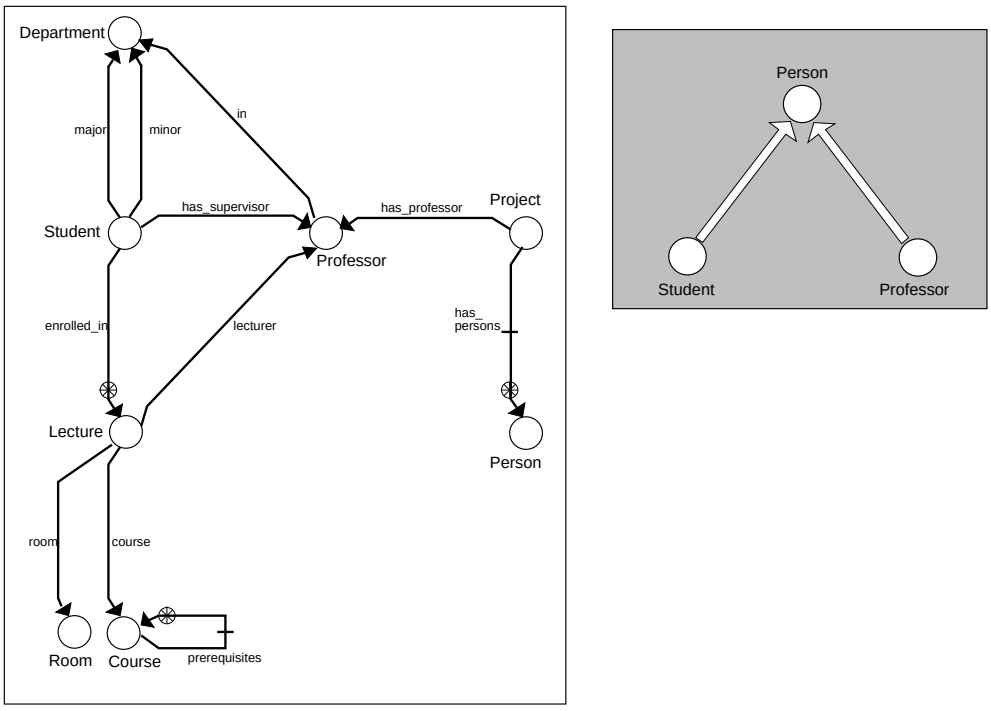


Abbildung B.2: Anzeige beider Diagramme auf einer Oberfläche

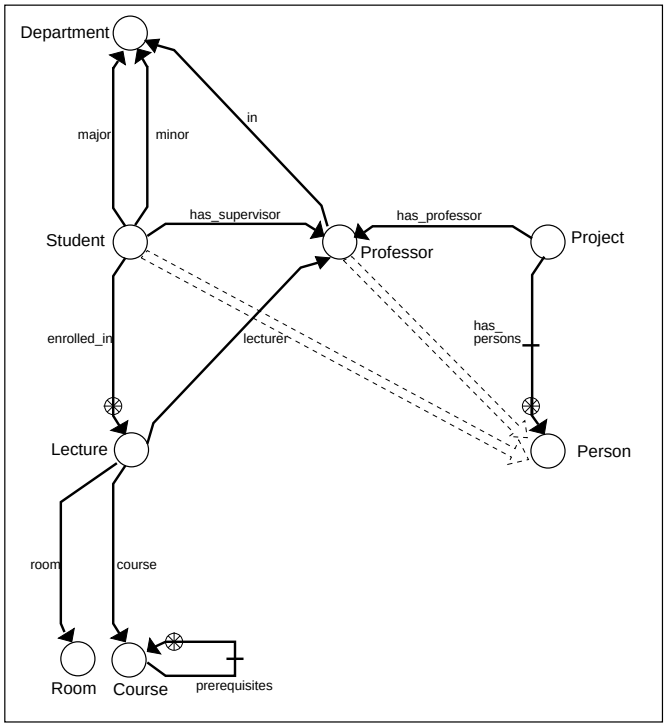


Abbildung B.3: Überlagerung von Referenz- und Hierarchischem Diagramm

B.3 Modularisierung von Diagrammen

Im folgenden werden Beispiele für Modularisierung von Diagrammen gegeben. Das Anwendungsbeispiel eines Krankenhauses ist aus [Tha91b] entnommen. Die ersten beiden der folgenden vier Abbildungen zeigen zwei Sichten einer vollständig modellierten Anwendung. Die dritte und vierte Abbildung stellen Möglichkeiten der Modularisierung der Anwendung dar.

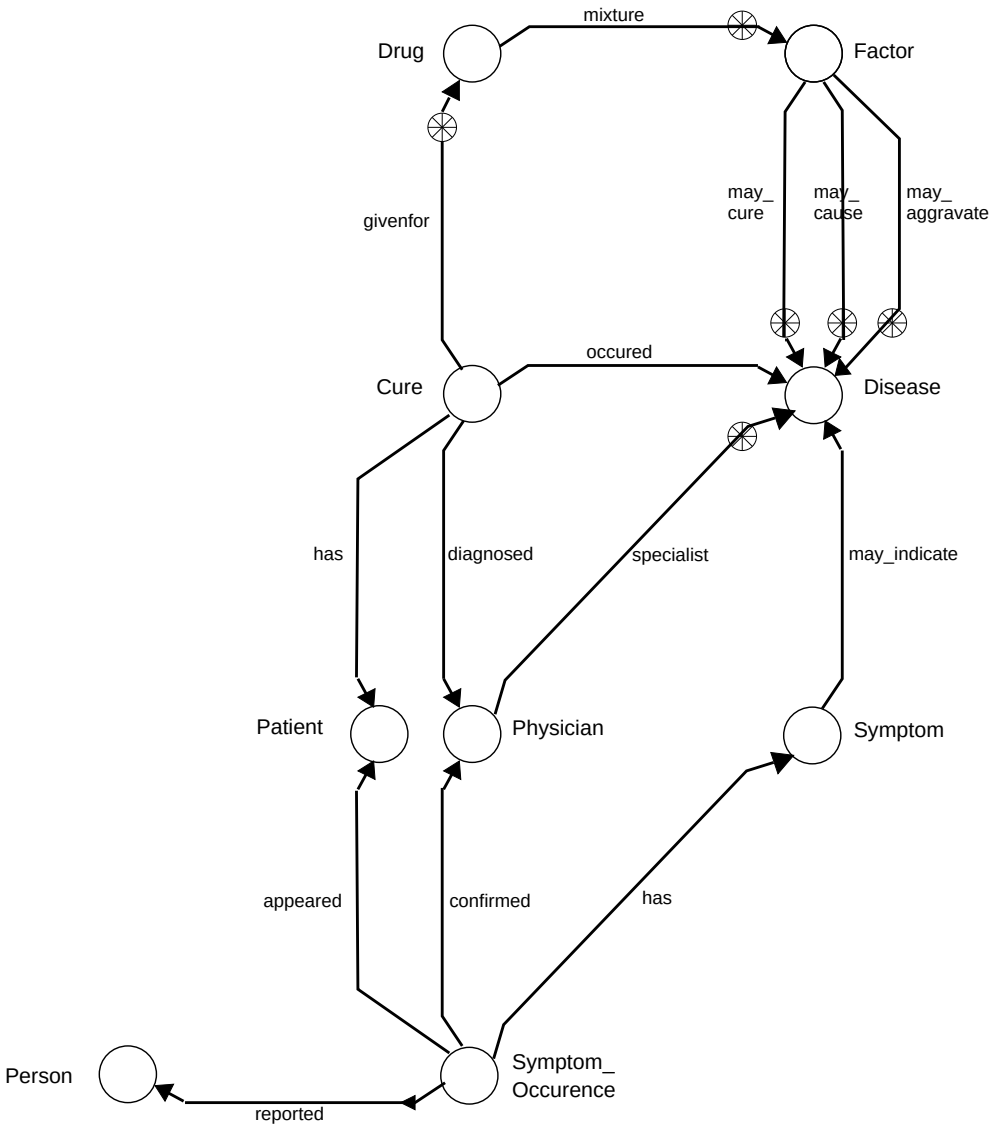


Abbildung B.4: Modellierung einer Krankenhausanwendung in OM1: Vollständige Schemaanzeige

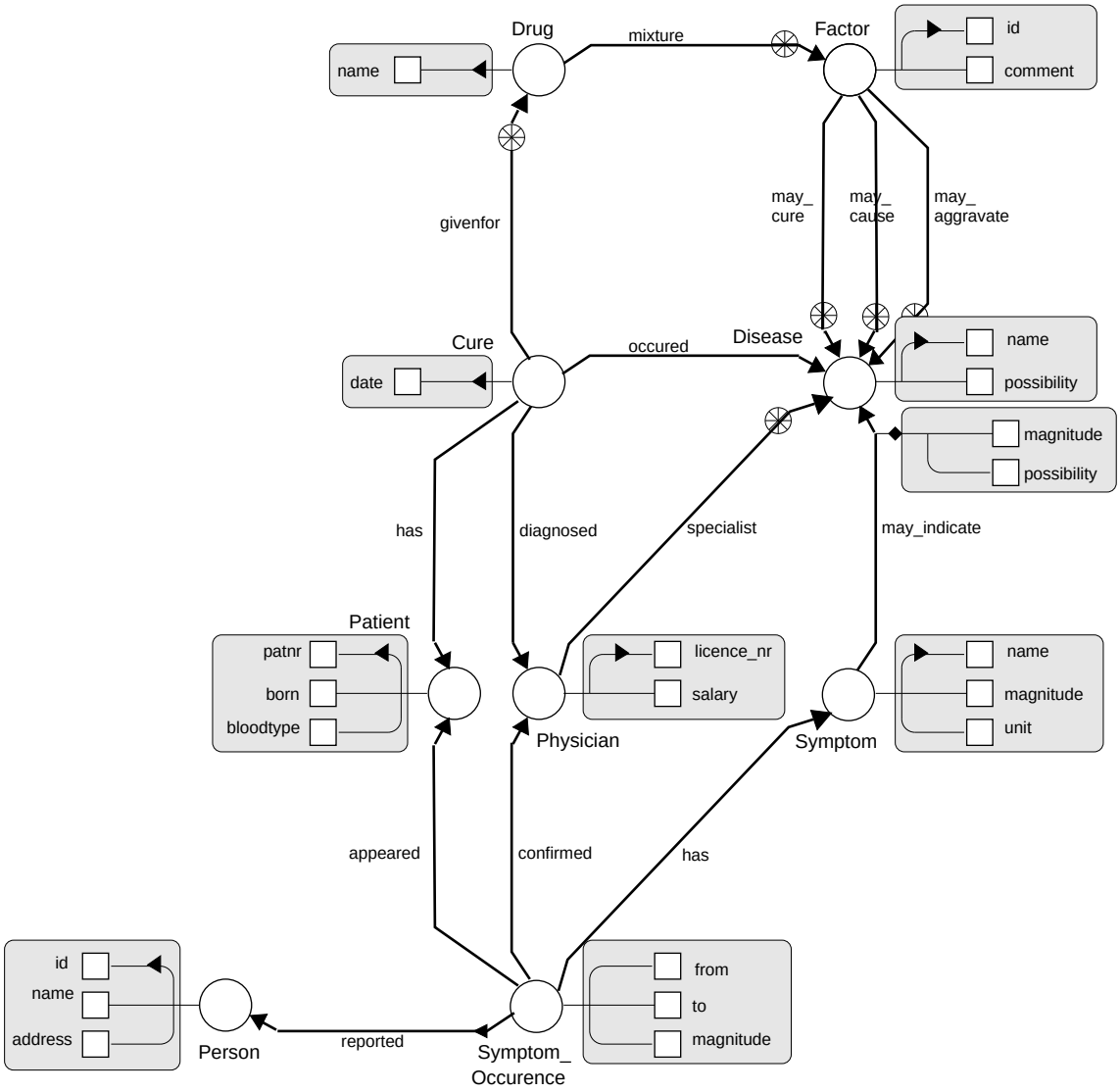


Abbildung B.5: Vollständige Modellierung, mit angezeigten Werteattributen

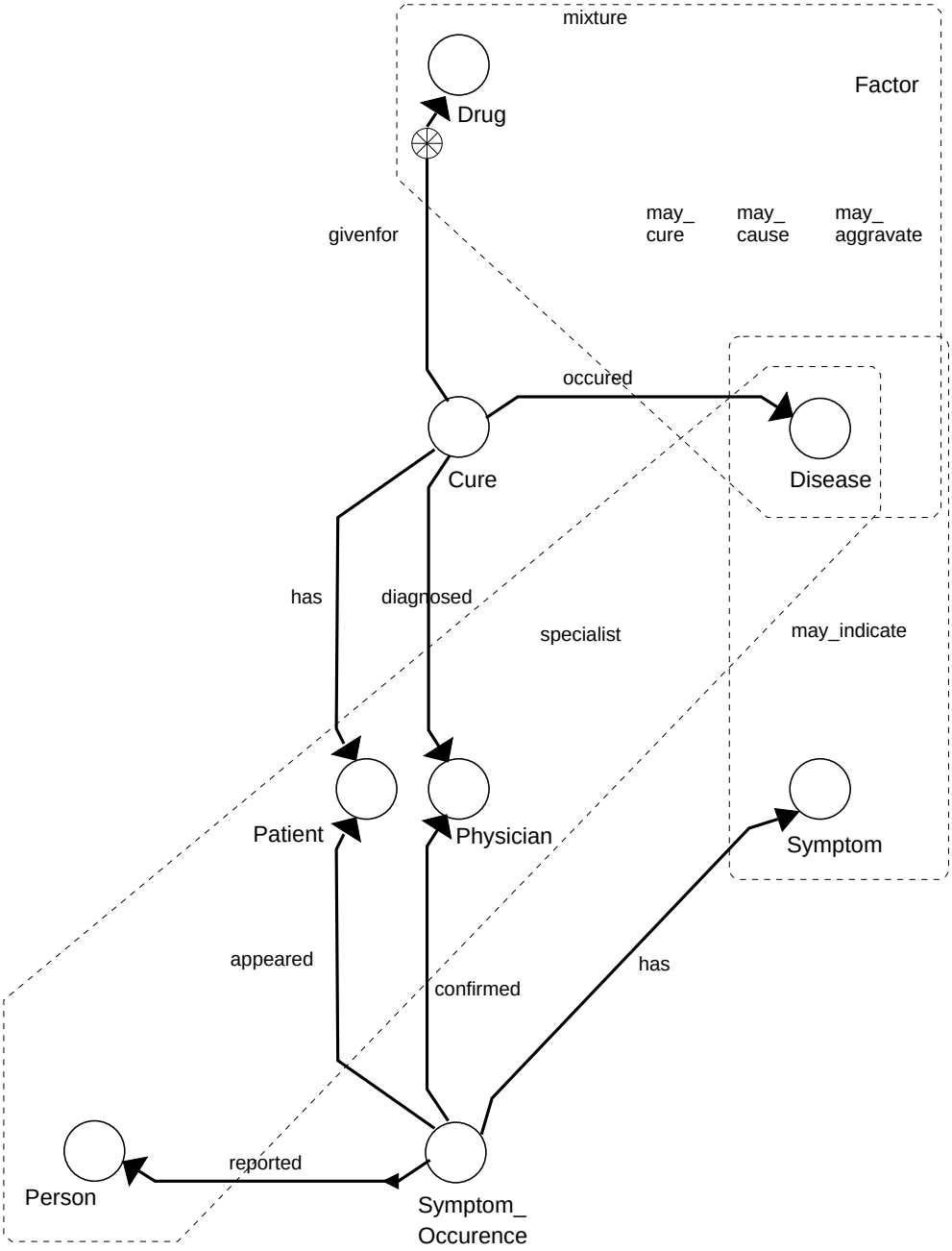


Abbildung B.6: Aufteilung einer graphischen Spezifikation in Module

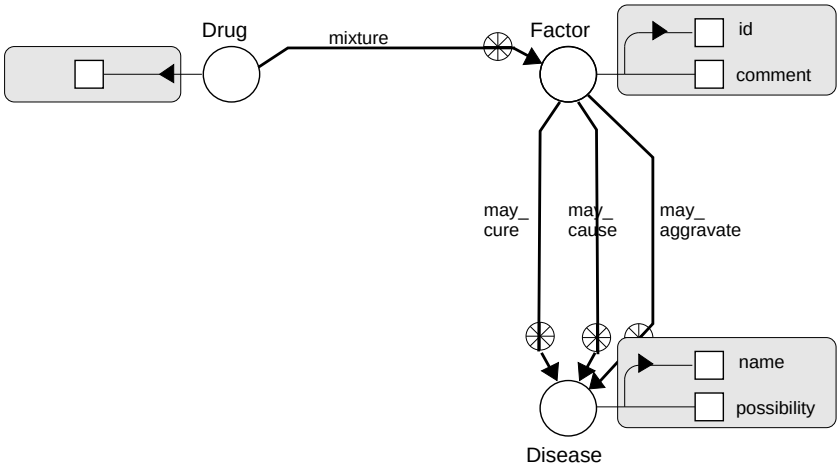


Abbildung B.7: Der Modul DRUG im Referenzdiagramm

Literaturverzeichnis

[A+91] J.R. Abrial et al. B-Tool: User Manual, Tutorial, Reference Manual. Technischer Bericht, Edinburgh Portable Compilers, Edinburgh, 1991.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, und R. Orsini. An Object Data Model with Roles. FIDE Technical Report Series FIDE/93/65, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1993.

[AH87] S. Abiteboul und R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, 12(4), 1987.

[AS90] Inc. Adobe Systems, Hrsg. *PostScript Language Reference Manual.* Addison-Wesley Publishing Company, Reading et al., 1990.

[AS91] Inc. Adobe Systems, Hrsg. *PostScript Language Tutorial and Cookbook.* Addison-Wesley Publishing Company, Reading et al., 1991.

[Bal88] H. Balzert. *Einführung in die Software-Ergonomie.* deGruyter, Berlin, New York, 1988.

[BDRZ83] R.P. Brägger, A. Dudler, J. Rebsamen, und C.A. Zehnder. Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions. In C.A. Zehnder, Hrsg., *Database Techniques for Professional Worskstations*, Seite 65–96. ETH Zürich, September 1983.

[Bee92a] C. Beeri. New Data Models and Languages – the Challenge. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1992.

[Bee92b] C. Beeri. Some thoughts on the future evolution of object-oriented database concepts. In W. Stucky und A. Oberweis, Hrsg., *Datenbanksysteme in Büro, Technik und Wissenschaft*, Seite 18–32. Springer-Verlag, 1992.

[BEM92] A.W. Brown, A.N. Earl, und J.A. McDermid. *Software Engineering Environments. Automated Support for Software Engineering.* McGraw-Hill Book Company, London, 1992.

[Ber88] J. Bertin. *Graphische Semiologie: Diagramme, Netze, Karten.* deGruyter, Berlin, New York, 1988.

[BKJ93] A. Busch, Th. Kuehnel, und A. Jahnke. *StarView C++ Klassenbibliothek.* Star Division, Hamburg, 1993.

[BKKZ92] R. Budde, K. Kautz, K. Kuhlenkamp, und H. Züllighoven. *Prototyping – an Approach to Evolutionary System Development.* Springer-Verlag, 1992.

[BMS93] A. Borgida, J. Mylopoulos, und J.W Schmidt. The TaxisDL Software Description Language. In M. Jarke, Hrsg., *Database Application Engineering with DAIDA*, Research Reports ESPRIT. Springer-Verlag, 1993.

[Bro84] M.L. Brodie. On the Development of Data Models. In M.L. Brodie, J. Mylopoulos, und J.W. Schmidt, Hrsg., *On Conceptual Modelling*, Topics in Information Systems. Springer-Verlag, 1984.

[BW94] G. Bremer und J. Wahlen. Strukturierte Generatoren zur Codeerzeugung. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, 1994.

[Che76] P.P.S. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, März 1976.

[Che80] P.P.S. Chen. *Entity-relationship Approach to Systems Analysis and Design.* North Holland, 1980.

[Den91] E. Denert. *Software Engineering. Methodische Projektabwicklung.* Springer-Verlag, 1991.

[Flo94] C. Floyd. Software-Engineering – und dann? *Informatik-Spektrum*, 17(1):29–37, 1994.

[Gal89] Galitz. *Handbook of Screen Format Design.* Wellesley, MA, 1989.

[Gei94] A. Geisler. Objektorientierte Datenmodellierung: Eine Funktionsbibliothek zur Entwurfsunterstützung. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Januar 1994.

[Gog94] J.A. Goguen. *En Extended Entity-Relationship Model.* Springer-Verlag, 1994.

[Heu92] A. Heuer. *Objektorientierte Datenbanken. Konzepte, Modelle, Systeme.* Addison-Wesley Publishing Company, 1992.

[HK87] R. Hull und R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3), 1987.

[HM81] M. Hammer und D. McLeod. Database Description with SDM: A Semantic Data Model. *ACM Transactions on Database Systems*, 6(3):351–386, 1981.

[Hug91] J.G. Hughes. *Object-Oriented Databases.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[Kas94] T. Kass. Objektorientierte Datenmodellierung: Ein Klasseneditor zur Entwurfsunterstützung. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1994.

[KGZ93] K. Kilberth, G. Gryczan, und H. Züllighoven. *Objektorientierte Anwendungsentwicklung.* Vieweg, 1993.

[Koe94] H. Koehler. Datenbankprogrammierung mit STYLE: Ein Anwendungsbeispiel. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1994.

[LM92] R. Löst und K. Möller. Entwurf einer Graphischen Schnittstelle für objektorientierte Datenbank-Spezifikationen. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Januar 1992.

[LS87] P.C. Lockemann und J.W. Schmidt, Hrsg. *Datenbank-Handbuch.* Springer-Verlag, 1987.

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

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German.).

[MM93] F. Matthes und S. Müßig. The Tycoon Language TL: An Introduction. DBIS Tycoon Report 112-93, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1993.

[MS91] F. Matthes und J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Database Programming Languages: Bulk Types and Persistent Data*, Nafplion, Greece, September 1991. Morgan Kaufmann Publishers.

[MS92] F. Matthes und J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[Mü94a] P. Münnix. Objektorientierte Datenmodellierung: Generierung statisch typisierter Repräsentationen. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.

[Mü94b] S. Müßig. Beiträge zur typsicheren generischen Datenvisualisierung. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.

[NS86] W. Newmann und R. Sproul. *Grundzüge der Interaktiven Computergraphik.* McGraw-Hill, Hamburg, 1986.

[Obe92] H. Oberquelle. *Vorlesungsskript Software-Ergonomie I + II.* Fachbereich Informatik, Universität Hamburg, Germany, 1992.

[Rau89] M. Rauterberg. *Die transparenz von desktop-orientierten im Vergleich zu konventionellen menü-gesteuerten Benutzungsoberflächen.* Forschungsbericht, ETH Zürich, 1989.

[Shn92] B. Shneiderman. *Designing the User Interface, Strategies for effective Human-Computer Interaction.* Addison Wesley, 1992.

[Sin92] A. Sinha. Client/Server Computing. *Comm. of the ACM*, 35(7):77–98, Juli 1992.

[SM90a] Inc. Sun Microsystems, Hrsg. *Network Interfaces Programmer's Guide.* Addison-Wesley Publishing Company, 1990.

[SM90b] Inc. Sun Microsystems, Hrsg. *OPEN LOOK Graphical User Interface Application Style Guidelines.* Addison-Wesley Publishing Company, 1990.

[SM90c] Inc. Sun Microsystems, Hrsg. *OPEN LOOK Graphical User Interface Functional Specification.* Addison-Wesley Publishing Company, 1990.

[SM92a] Inc. Sun Microsystems, Hrsg. *NeWS 3.1 Programmer's Guide.* Sun Microsystems, Inc., Mountain View, 1992.

[SM92b] Inc. Sun Microsystems, Hrsg. *The NeWS 3.1 Toolkit 3.1 Reference Manual.* Sun Microsystems, Inc., Mountain View, 1992.

[Sta87] B. Staufer. *Piktogramme für Computer: Kognitive Verarbeitung, Methoden zur Produktion und Evaluation.* deGruyter, Berlin, 1987.

[Tha91a] B. Thalheim. *Dependencies in relational databases.* Teubner-Verlag, Stuttgart, 1991.

[Tha91b] B. Thalheim. The Higher-Order Entity-Relationship Model. In *Proceedings of the Kiev East/West Workshop on Next Generation Database Technology*, 1991.

[TYF86] T.J. Teorey, D. Yang, und J.P. Fry. A logical design methodology for relational databases using the extended entity relationship model. *ACM Computing Surveys*, 18(2):197–222, Juni 1986.

[Wet94] I. Wetzel. *Programmieren mit STYLE: Über die systematische Entwicklung von Programmierumgebungen.* Dissertation, Fachbereich Informatik, Universität Hamburg, Germany, Juli 1994.