Erweiterung objektorientierter Methoden für den konzeptuellen Datenbankentwurf

Diplomarbeit
der philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Benno Burkhardt, 91-108-225
1997

Leiter der Arbeit:
Dr. Andreas Geppert
Institut für Informatik der Universität Zürich
Prof. Dr. Oscar Nierstrasz
Institut für angewandte Mathematik der Universität Bern

„The amateur software engineer is always in search of magic, some sensational method or tool whose application promises to render software development trivial. It is the mark of the professional software engineer to know that no such panacea exists. Amateurs often want to follow cookbook steps; professionals know that right approaches to development usually lead to inept design products, born of a progression of lies, and behind which developers can shield themselves from accepting responsibility for earlier misguided decisions. The amateur software engineer either ignores worrying more about the substance they contain. The professional acknowledges the importance of creating certain documents, but never does so at the expense of making sensible architectural innovations.“ [Booch 94, p. 229]

Vorwort

Die vorliegende Diplomarbeit wurde von der Gruppe „Objektorientierte Datenbanken“ am Institut für Informatik der Universität Zürich wie folgt ausgeschrieben:

Erweiterung objektorientierter Methoden für den konzeptuellen Datenbankentwurf

Konzeptueller Entwurf ist eine gut verstandene Aktivität in der Entwicklung relationaler Datenbankanwendungen. Für objektorientierte Datenbanksysteme ist dies nicht der Fall, da bei in der Praxis verwendeten Entwurfsverfahren wie Booch oder OMT einige notwendige Entwurfskonstrukte und -richtlinien nicht unterstützt werden. Die Verwendung typischer objektorientierter Entwurfsverfahren ist somit in der Regel für die objektorientierten Datenbanksysteme nicht zufriedenstellend.

Ausgehend von einem objektorientierten Verfahren (z.B. Booch), sind Schwachstellen zu bestimmen und durch angemessene Entwurfskonstrukte zu beseitigen. Schliesslich soll ein grafisches Entwurfswerkzeug um diese neuen Entwurfskonstrukte erweitert werden.

Die Arbeit bietet die Möglichkeit, mit aktuellen und praxisrelevanten Aspekten der Datenbanktechnologie vertraut zu werden. Die Arbeit wird in Kooperation mit Wirtschaftspartnern bei der CSS-Versicherung (Luzern) durchgeführt.

Im Bereich des Datenbankentwurfes wurden bereits viele Forschungsanstrengungen unternommen. Die Arbeiten beziehen sich aber meistens auf das relationale Modell, oder führen komplett neue Methoden ein (vgl. [Kappel et. al. 91a]), die aber aufgrund von fehlenden Implementationen in der Praxis nicht angewendet werden. Häufig besitzen Entwerfer bereits Erfahrungen im Umgang mit objektorientierten Entwurfsmethoden, weshalb sie nicht besonders gewillt sind, eine komplett neue Methodik zu erlernen. Vielmehr erwarten sie, dass sich neue technologische Aspekte auch in entsprechenden Konstrukten in der von ihnen eingesetzten Entwurfsmethode nierderschlagen. Die gängigen Methoden jedoch unterstützen keine der datenbankspezifischen Aspekte des Entwurfes und wenn, dann legen sie das relationale Modell zugrunde. Damit findet man in der Industrie häufig die Situation, dass die Systementwickler eine objektorientierte Methode für den Applikationsentwurf und für den Datenbankentwurf anwenden, aber bezüglich des Entwurfes der datenbankspezifischen Aspekte wie persistente Objekte, Transaktionensentwurf und Konsistenzsicherung, Objekt- und Schemaevolution allein gelassen sind.

Es macht daher Sinn, bestehende und akzeptierte Entwurfsmethoden zu erweitern, damit mit ihnen auch die Spezifika des Datenbankentwurfes berücksichtigt werden können.

Für die Erreichung des in der Diplomarbeit gesteckten Ziels einer derartigen Erweiterung einer bestimmten Methode für den konzeptuellen Datenbankentwurf musste das Problem in die folgenden Teilprobleme unterteilt werden:

Teil I: Erweiterung des konzeptuellen Datenbankentwurfes:

1. Auswahl einer bestimmten Entwurfsmethode.
2. Auswahl einer bestimmten Datenbankimplementierung.
3. Analyse der Schwachstellen der gewählten Entwurfsmethode bei deren Anwendung für den konzeptuellen Datenbankentwurf.

4. Abgleich der gefundenen Schwachstellen mit den in der Literatur erwähnten und in der Parxis aufgetretenen Unzulänglichkeiten.
5. Auswahl verschiedener Schwachstellen (Evaluation) anhand forschungs- und praxisrelevanter Kriterien (Fallstudien und Projekte der CSS-Versicherung).
6. Erweiterung der ausgewählten Entwurfsmethode (Anpassung bestehender oder Einführung neuer Konstrukte, Formulierung von Richtlinien für deren Verwendung).
7. Bewertung der Ergebnisse (inwiefern konnten die Schwachstellen beseitigt werden?).
8. Anwendung der erweiterten Methode auf die in der Analyse verwendeten Fallstudien.

Teil II: Implementation der Erweiterungen:

1. Erweiterung eines bestehenden Zeichnungswerkzeuges für den softwaregestützten Entwurfsprozess.
2. Definition eines Umsetzungskonzeptes (wie können die Konstrukte der grafischen Entwurfssprache in das gewählte Datenschema abgebildet werden?).
3. Implementation eines Übersetzers, welcher die gezeichneten Diagramme in eine konkrete Schemadefinitionssprache überträgt.

Die Arbeit wurde als eine herausfordernde Aufgabe angesehen. Trotz der grossen Anzahl von Papieren, die zu diesem Thema geschrieben wurden, blieb ein befriedigendes und vor allem in der Praxis anwendbares Resultat noch ausstehend. Dennoch ist davon ausgegangen worden, dass im Rahmen dieser Diplomarbeit einiges erreicht werden kann.

Die Betreuung wurde von Dr. Andreas Geppert, Oberassistent am Institut für Informatik der Universität Zürich, Andreas Behm, Assistent am Institut für Informatik der Universität Zürich, und Prof. Nierstrasz, Professor am Institut für angewandte Mathematik der Universität Bern übernommen, wofür ich mich an dieser Stelle noch einmal ausdrücklich bedanken möchte.
Im weiteren konnte ich während meiner Arbeit von der grossen praktischen Erfahrung von Herrn Thomas Wüst, ehemaliger Assistent am Institut für Informatik der Universität Zürich und heutiger Mitarbeiter der CSS Versicherung mit der Entwicklung von grossen auf objektorientierten Datenbanksystemen basierenden Informationssystemen betraut, profitieren.
Zudem sei erwähnt dass die Unterstützung der „Software Composition Group“ am Institut für angewandte Mathematik der Universität Bern durch ihr Feedback viel zu dieser Arbeit beigetragen hat.
Mein Dank gilt aber auch der Firma TeTrade (Informatik) AG, insbesondere aber Michael Liebi, Roberto Liviero und René Ritter, für die Nutzung der Infrastruktur und Andreas Münger, Jean-Luc Nottaris, für die fachlichen und Eva Burkhardt und Dolores Denaro für die sprachlichen Korrekturlesungen.

Inhaltsverzeichnis

1 Einleitung

1.1 Objektorientierte Methoden

Wegen der Allgemeinheit des Wortes „Objekt“ läuft auch der Begriff „objektorientiert“ Gefahr, bei unvorsichtiger Verwendung zu einem Allgemeinplatz zu verkommen. Besonders gilt es zu beachten, dass ein Objekt im nachfolgend skizzierten Sinn als technisches Konzept zu verstehen ist, mit dessen Hilfe die diskutierte Umwelt modelliert werden kann. Es ist daher strikt von den „Objekten“ der betrachteten Umwelt selber abzugrenzen, obwohl man natürlich hofft eben diese so genau wie möglich auf die Systemobjekte abbilden zu können.

1.1.1 Objektorientierung

Identität und Schnittstelle

Ausgehend vom Konstrukt der abstrakten Datentypen schafft die Objektorientierung einen Rahmen um die Daten und die für sie definierten Operationen und fasst diese unter dem Begriff des Objektes zusammen. Ein derartiges Objekt ist anschliessend von aussen nur über dessen eindeutige Identität und dessen Schnittstelle sichtbar.

Diese Eigenschaft kann durch folgende drei Grundprinzipien beschrieben werden:

Abstraktion und Autonomie

1. Abstraktion und Autonomie: Ein Objekt besitzt eine wohlunterscheidbare Identität, einen inneren Zustand und eine Anzahl von Meldungen, über welche mit dem Objekt kommuniziert werden kann. Der Zustand ist durch eine Datenstruktur modelliert. Die Meldungen bilden die Schnittstellen zum Objekt und sind intern als Methoden, also Programmfunktionen implementiert.

Klassifikation

2. Klassifikation: Häufig trifft man Objekte, die sich hinsichtlich ihres Verhaltens nicht unterscheiden, wohl aber hinsichtlich ihres internen Zustandes. Derartige Objekte lassen sich in Objektklassen zusammenfassen. Einzelne Objekte werden alsdann als unterscheidbare Instanzen einer solchen Klasse betrachtet.

Taxonomie

3. Taxonomie: In vielen Fällen sind sich Instanzen zweier Klassen soweit ähnlich, als die einen sämtliche Eigenschaften der anderen besitzen und darüber hinaus noch über weitere speziellere Eigenschaften verfügen. Auf diese Weise lassen sich Klassenhierarchien bilden, indem man aus Basisklassen durch Vererbung neue Klassen erhält, die als Spezialisierungen gesehen werden können.

Zur Verwendung eines Objektes muss also lediglich die Schnittstelle zu ihm bekannt sein (syntaktisch und semantisch). Der Zustand des Objektes ist also nur dem Objekt selber bekannt und ist von aussen nicht oder nicht direkt ersichtlich. Diese Eigenschaft wird auch als Abstraktion oder Kapselung verstanden. Dem Prinzip der Abstraktion und Autonomie steht keineswegs entgegen, dass ein Objekt seinerseits Meldungen an andere Objekte

verschickt, also Teile seines Verhaltens durch das Verhalten anderer Objekte abstützt. Damit entstehen Assoziationen zwischen Objekten, oder anders gesagt: Objektstrukturen.
Verschiedentlich, aber nicht zwangsläufig, wird die Klasse auch als Extension, also als die Gesamtheit der tatsächlich existierenden Instanzen einer Klasse betrachtet, welche selber wieder als Objekt angesehen werden kann.

Vererbung und Mehrfachvererbung

Der Begriff der Taxionomie kann auch Vererbung genannt werden, wenn man berücksichtigt, dass eine Unterklasse von ihrer Oberklasse alle Eigenschaften übernimmt oder erbt. Neben der einfachen Vererbung ist auch das Konstrukt der mehrfachen Vererbung anzutreffen, jedoch unterscheiden sich die verschiedenen Implementierungen in diesem Bereich konzeptuell zuweilen erheblich.

1.1.2 Entwurf

Klassierung

Die Darstellung von Entwurfsentscheidungen in einem objektorientierten System läuft häufig nicht in derselben Reihenfolge wie der zu grundeliegende Erkenntnisprozess ab. So betrachtet man in der Umwelt Objekte als Instanzen von Klassen, während man im Entwurf anhand von beobachteten konkreten Objekten eine Klassierung herausdestilliert.

natürliche Modellierung

Die Vorteile, die man sich von der Verwendung von objektorientierten Paradigmen verspricht, liegen vor allem in der Chance, eine gegebene Umwelt natürlich modellieren zu können, da die in dieser Umwelt betrachteten Sachverhalte als Objekte im Softwaresystem dargestellt werden können.

kapseln von Zuständen und Eigenschaften

Im weiteren hat man die beklagenswerte „künstliche“ Trennung von Daten und Funktionen soweit überwunden, als dass man sowohl Zustände als auch Eigenschaften in einem Objekt kapseln kann. Das Verhalten ist somit weitgehend von der Instanz selber festgelegt. Darüber hinaus erwächst dem Entwerfer der Vorteil, lediglich die semantisch sinnvollen Operationen für das Objekt von aussen sichtbar zu machen und somit die Abstraktion bei der Systemgestaltung zu unterstützen.
Schliesslich kann mit Hilfe der Taxionomie die Wiederverwendbarkeit von Komponenten (einzelne Klassen oder ganze Klassenbäume als Rahmenwerke) verbessert werden, indem man neue Klassen als Spezialisierungen von bereits bestehenden Klassen und deshalb so tief wie möglich im Ableitungsbaum hinzufügt.

1.2 Objektorientierte Entwurfsmethoden

Eine beachtliche Anzahl objektorientierter Entwicklungsmethoden sind in den letzten Jahren eingeführt worden, um erweiterbare, wiederverwendbare und robuste Software zu entwickeln. Nachfolgend sollen einige davon aufgezählt sein und auf mögliche Probleme eingegangen werden.

1.2.1 Objektorientierte Softwareentwicklung

Analyse, Entwurf und Implementation

Die objektorientierte Softwareentwicklung hat es sich zum Ziel gemacht, die objektorientierten Methoden in den Phasen der Analyse, des Entwurfs und der Implementation umzusetzen. Dabei sollen während dieses Ablaufs alle Prozesse auf dem einmal erstellten Modell basieren, was eine höhere Wiederverwendbarkeit von Resultaten aus vorgelagerten Phasen unterstützt, und dieses lediglich verfeinern.

Analyse

In der Analyse sollen die Entitäten der realen Welt möglichst getreu auf die Objekte innerhalb des Modells abgebildet werden. Zudem ist in dieser Phase die Strukturierung des meist ungeordnet vorliegenden Problems und dessen Zerlegung in Teilprobleme oder Subsysteme von eminenter Wichtigkeit. Als Resultat steht die Definition der Objekttypen und deren Hierarchie, die aus der Klassifizierung des Wissens über die beobachtete Problemstellung entsteht, im Vordergrund. Auf diese Weise können auf der einen Seite weitgehend wiederverwendbare Klassenstrukturen entstehen und auf der anderen Seite bereits bestehende Basisklassen durch Spezialisierung in den neuen Klassenbaum eingeflochten werden.

Entwurf

Während sich die Analyse auf die Definition der Objekte beschränkt, werden in der Phase des Entwurfes die Beziehungen zwischen den identifizierten Objekten unter die Lupe genommen. Als Typen sind strukturelle Abhängigkeiten (Spezialisierung, Generalisierung, Enthaltung) und Interaktionen (Meldungsverbindungen) zu unterscheiden.

1.2.2 Übersicht der objektorientierten Analyse- und Entwurfsmethoden

Nachfolgend seien einige repräsentative Beispiele von objektorientierten Analyse- und Entwurfsmethoden angegeben, die sich alle ihren Platz in der Liste der „state-of-the-art“-Methoden gesichert haben (vgl. [Aksit et. al. 92]).

Booch

In der objektorientierten Entwurfsmethode von **Booch** [Booch 94] werden Diagramme für Klassen, Klassenkategorien, Verhalten (state-transition), Objekte, Module, Subsysteme, Prozessoren und Prozesse vorgeschlagen. Alle diese Diagramme können zum einen durch die Verwendung von wohldefinierten Konstrukten grafisch oder zum anderen durch Beschreibung von Vorlagen textuell dargestellt werden. Die Diagrammnotation bietet eine bessere Übersicht, während die freitextliche Charakterisierung wesentlich detaillierter ist. Die eingeführte Methode läuft nach den Schritten Identifizierung der Objekte und der Klassen, Definition der Objektsemantik, Beschreibung der Objektbeziehungen und Implementierung der entworfenen Sachverhalte ab.

Champeaux Die objektorientierte Analyse und „Top-Down“ Softwareentwicklung **Champeaux** [Champeaux 91] wählt den Ansatz vom Grossen ins Kleine zu gelangen, indem man das Konstrukt des „Ensembles“ verwendet. Die eingeführten „Ensembles“ sind Subsysteme und vergleichbar mit Objekten. Der Hauptunterschied liegt darin, dass sie interne Konkurrenz zulassen, während dies bei Objekten nicht zugelassen ist. Die Methode besteht aus drei Komponenten, namentlich der Information, die in Form eines Objektmodells mit seinen strukturellen Beziehungen dargestellt ist, des Zustandes, der das dynamische Verhalten einer Objektes definiert und des Prozesses, der die Interaktionen zwischen den Objekten beschreibt. Diese drei Bestandteile sind schon in früheren Methoden wie „Objektorientierte Systemanalyse“ [Mellor 88] angewandt worden.

Coad & Yourdon Die Methode von **Coad & Yourdon** namens „Objektorientierte Analyse und Objektorientierter Entwurf„ ist in zwei separate Bereiche aufgeteilt. Der erste Teil behandelt die Analyse (vgl. [Coad et. al. 91a]), indem fünf vertikale Schichten - Subjekte, Klassen, Objekte, Strukturen, Attribute und Dienste - betrachtet werden. Diese fünf Ebenen werden später in der Entwurfsphase (vgl. [Coad et. al. 91b]) wiederverwendet und mit vier horizontalen Bereichen - Problemdomäne, Benutzerinteraktion, Prozessverwaltung und Datenspeicherung - durchsetzt, woraus die Matrixarchitektur resultiert.

Rumbaugh **OMT** („Object-Oriented Modeling and Design“) [Rumbaugh et. al. 91] stützt sich auf drei Modelle und eine Methode, die deren Aufbau und Verwendung angibt. Zuerst wird das Objektmodell erstellt, anschliessend wird das dynamische Verhalten durch Zustandsdiagramme veranschaulicht, und zum Schluss werden die funktionalen Eigenschaften durch Datenflussdiagramme näher spezifiziert.

Wirfs-Brock Der Ansatz von [Wirfs-Brock 90] genannt „The Responsability-Driven Approach“ definiert sechs Aktivitäten. Die erste zielt darauf ab, die Klassen und deren Hierarchie zu identifizieren. Im zweiten Schritt werden die Operationen oder die sogenannten Verantwortlichkeiten spezifiziert. Als drittes gilt es, die Objektinteraktionen - genannt Kollaborationen - zu definieren. Die vierte Aktivität hat zum Ziel, die Wiederverwendbarkeit durch Verfeinerung der Klassenhierarchie zu erreichen. Im fünften werden Klassen in Subsystemen gruppiert und im sechsten und letzten Teil werden die Objektschnittstellen in Form von Protokollen niedergeschrieben.

Erwähnenswert sind im weiteren die Methoden „Johnson and Foote“ [Foot et. al. 88], „The Demeter system“ [Holland et. al. 89] und [Lieberherr et. al. 91] und „Object-Oriented Role Analysis, Synthesis and Structuring“ [Johnson et. al. 90], auf die aber hier nicht speziell eingegangen werden soll.

1.2.3 Probleme mit objektorientierten Entwurfsmethoden

[Askit et. al. 92] erwähnen in ihrer Arbeit verschiedene Probleme, welche in der Phase der Analyse und der Vorbereitungsarbeit im Zusammenhang mit objektorientierten Entwurfsmethoden

auftreten. Sie werden im Folgenden den verschiedenen Entwurfsphasen zugeteilt und summarisch angesprochen.

1.2.3.1 Analyse

Analyse des Problembereiches der Realwelt:

- Je kleiner das Wissen über die Sachverhalte der untersuchten Domäne ist, wie dies in wissenschaftlichen Gebieten oft der Fall ist, desto schlechter lässt sich eine klare Klassenhierarchie für dessen Abbildung finden.
- Der Entwurf der Klassenhierarchie kann Objekte und Strukturen entstehen lassen, die für den zu analysierenden Problembereich nicht relevant sind.

Identifikation der Subsysteme und der Objekte:

- Die Unterteilung eines Problems in Teilprobleme vor der Objektidentifikation verursacht in gewissen Situationen suboptimale Schnittstellen zwischen den Subsystemen. Werden erst Objekte identifiziert und anschliessend Untersysteme gebildet, können sich eventuell unüberschaubare Strukturen ergeben.
- Die Unterscheidung von Subsystemen und Objekten ist im Allgemeinen sehr schwierig. Bei der Verfeinerung des Entwurfes werden vielfach Objekte aufgrund ihrer komplexen internen Struktur in Subsysteme konvertiert. Auf die gleiche Weise können während der Analyse definierte Subsysteme bei der genaueren Betrachtung als Objekte modelliert werden, da ihre interne Struktur einfach und übersichtlich ist. Die Konvertierung von Subsystemen zu Objekten und Objekten zu Subsystemen verursacht eine Änderung der Elementsemantik.
- Die Klassenhierarchie und die Subsystemdefinition können sich überlagern, d.h. in einem Subsystem befindet sich nicht ein ganzer zusammenhängender Teilbaum der Klassenhierarchie (der Grossvater einer Klasse liegt z.B. in demselben Subsystem, der Vater in einem anderen).
- Die Definition der Subsysteme anhand von Objektinteraktion ist vielfach intuitiv und versagt oft bei grossen Systemen. Deshalb sind automatische (algorithmische) Methoden empfehlenswert.

1.2.3.2 Entwurf der strukturellen Relationen

Gemeinsames Verhalten:

- Meldungsdelegationen und Delegationshierarchien (anstelle von Vererbungshierarchien) sind in heutigen objektorientierten Entwurfsmethoden nicht unterstützt.

Atomarität versus Vererbung

- Bei Mehrfachvererbung von Klassen, die atomare Transaktionen im Interface exportieren, müssen im Extremfall sämtliche Kombinationen der Funktionen beider Klassen atomar gemacht werden, damit sie von der

abgeleiteten Klasse exportiert werden dürfen. Die Deklarationsanzahl steigt daher exponentiell.

Vererbungsmechanismen

- Bei der Vererbung können Methoden nur überschrieben und nicht erweitert werden. (Beispiel des Rechners als Superklasse und des wissenschaftlichen Rechners als Unterklasse: der Rechner stellt eine Additionsfunktion, basierend auf natürlichen Zahlen, zur Verfügung. Die entsprechende Additionsfunktion des wissenschaftlichen Rechners muss aber auch andere Argumente unterstützen. Wünschenswert wäre eine Erweiterung der Basisfunktion in der Subklasse, welche die Grundfunktionalität der Vaterklasse übernehmen würde. Dieses Verhalten muss jedoch mit der Überschreibung abgebildet werden.) Der Einbezug dieser Sachverhalte würde bedeuten, dass man bereits beim Design an die Implementation denken müsste, was bestimmt nicht wünschenswert wäre.

Vererbung versus Zustände

- Zustände einer Instanz der Superklasse können in Unterklassen anders interpretiert werden, d.h. der Zustand der Oberklasse wird durch den Zustand der Unterklasse überschrieben (Beispiel eines limitierten Puffers: die Superklasse implementiert einen ein-elementigen Zugriff, während das Derivat zwei-elementig operiert. Die Zustandsangabe „Ende des Puffers“ kann nun nicht mehr von beiden Klassen gleich behandelt werden.)

1.2.3.3 Entwurf der Objektinteraktionen

Mehrfache Ansicht

- Die Mehrfachansicht eines Objektes kann nicht mit den objektorientierten Konstrukten abgebildet werden. (Beispiel einer Schuladministration mit Lehrern, die gleichzeitig Schüler sind: die beiden Verhalten können nicht in zwei verschiedenen Klassen beschrieben werden, da ansonsten ein Objekt als Doppelinstanz aufgefasst werden müsste. Ein Objekt braucht aber immer eine eindeutige Identifikation.)

Koordiniertes Verhalten

- Koordiniertes Verhalten kann nicht durch Vererbung erreicht werden. Die Kooridinationsklasse wird vielmehr von den Beteiligten verwendet (Beispiel Kreuzungsüberquerung von mehreren Verkehrsteilnehmern von mehreren Typen.)

1.3 Objektorientierte Datenbanken

Solange nicht bestimmte Vorkehrungen getroffen werden, leben Datenelemente von Programmen nur solange, wie diese in der Ausführung begriffen sind, d.h. die Inhalte gehen bei Programmende verloren. Daten können dauerhaft (persistent) zur Verfügung stehen, wenn diese in Dateien abgelegt werden. Programme können aber dennoch nicht direkt auf derartig gespeicherte Datenelemente zugreifen, vielmehr müssen sie bei

der Verwendung explizit kopiert, bzw. bei deren Manipulation überschrieben werden.

1.3.1 Konzepte von Datenbanksystemen

Mit dem Schritt von Dateien zu Datenbanken wurde zwar prinzipiell die Notwendigkeit des Kopiervorganges nicht beseitigt, jedoch liegen die Daten im Hintergrundspeicher nun dauerhaft, zuverlässig, unabhängig und strukturiert vor und erlauben eine komfortable und flexible Verwendung, selbst im Mehrbenutzerbetrieb zu.

Zuverlässigkeit

Als zuverlässig soll in diesem Zusammenhang die Eigenschaft der selbständigen Sicherung von Konsistenz und Integrität gedeutet werden, während unter flexibel die Existenz einer strukturierten Abfragesprache zu verstehen ist.

Datenmodell und Schema

Der Datenbank liegen ein Datenmodell und ein Schema, die als konkrete Beschreibung eines Datenmodells für einen Einsatzfall zu betrachten ist, zu Grunde. Datenmodelle werden durch die Datendefinitionssprache, bzw. Datenmanipulationssprache erstellt, bzw. verändert. Für das Einspeichern, Auffinden, Ändern und Löschen steht eine Abfragesprache zur Verfügung.

hierarchische, netzwerkartige und relationale Datenmodelle

Unter den heute gängigen Datenmodellen findet sich neben dem hierarchischen und dem netzwerkartigen vor allem das relationale. Letzteres modelliert Umweltobjekte als Tupel (Zeilen) in Tupelmengen (Tabellen), wobei die einzelnen Tupelelemente als Attribute zu verstehen sind. Derartige Mengen werden auch Relationen genannt und basieren auf der Relationenalgebra.

1.3.2 Erweiterte Anforderungen

Aus der Verwendung von herkömmlichen Datenbanksystemen werden Forderungen nach Erweiterungen laut, von denen nachfolgend beispielhaft zwei herausgegriffen werden sollen:

zusammengesetzte Objekte

Oftmals trifft man gerade im Anwendungsbereich zusammengesetzte Umweltobjekte, also Objekte, die als Sammlung von Teilobjekten angesehen werden können. Derartige Kompositionen können auch rekursiv auftreten und werden daher mit beispielsweise dem relationalen Modell nur schwerlich mehr handhabbar. (Man beachte, dass zur Verarbeitung solcher Objekte viele Verbindungsoperationen notwendig sind, was die Leistung gängiger Systeme schnell an Grenzen stossen lässt.)

komplexe Datentypen

Üblicherweise stellen Datenbanksysteme eine Reihe von elementaren Datentypen zur Verwendung zur Verfügung. Diese Liste deckt aber selten die tatsächlichen Bedürfnisse, insbesondere nach komplexen Datentypen, ab. Auch hier wäre prinzipiell eine Modellierung mit relationalen Mitteln möglich, jedoch ist sie nur durch Zuhilfenahme von weiteren ausserhalb der Datenbank liegenden Hilfsmittel erreichbar, was sich auf Kosten der geforderten Sicherheiten äussern würde.

1.3.3 Objektorientierte Datenbanksysteme

Im Manifesto für objektorientierte Datenbanksysteme [Atkinson et. al. 89] ist zwar keine Definition gegeben, die sich auf eine allgemeine Anerkennung stützen könnte, vielmehr sind einige Punkte genannt, welche von einem derartigen System gefordert werden sollten:
Ein objektorientiertes Datenbankmanagementsystem muss alle funktionalen Eigenschaften eines klassischen Datenbankmanagementsystems beinhalten, es muss also insbesondere

- dauerhafte Verwaltung von Datenelementen bieten,
- den vollen verfügbaren Hintergrundspeicher ausnutzen können,
- durch ein Transaktionskonzept Mehrbenutzerbetrieb zulassen,
- für Wiederanlaufsmöglichkeiten im Fehlerfall sorgen, und
- mengenorientierte deklarative Anfragen unterstützen.

Anders gesagt soll das System keine der bereits erzielten Errungenschaften einbüssen.
Zudem werden aber an die neue Datenbankfamilie weitere wesentliche Anforderungen gestellt, z.B. muss sie

1. ein objektorientiertes Datenmodell aufweisen,
2. Objekte eindeutig identifizieren, d.h. Objektidentitäten verwalten,
3. berechnungsvollständig sein,
4. zusammengesetzte Objekte unterstützen,
5. das Klassenkonzept zur Verfügung stellen,
6. Einkapselung umsetzen,
7. das Konzept der Klassenhierarchie und der Vererbung implementieren und
8. Überladung, Überschreibung und spätes Binden beinhalten.

Man beachte, dass die Forderungen 5-7 direkt aus den Eigenschaften der Objektorientierung, wie sie unter dem Titel „objektorientierte Methoden“ ausgeführt wurden, ableitbar sind.
Die Forderung nach den zusammengesetzten Objekten (4) findet seine Entstehung in den Anforderungen an weitergehende Datenbanksysteme. Das im selben Atemzug genannte Bedürfnis nach komplexen Datentypen ist implizit in den obigen Anforderungen enthalten, da sämtliche Klassen-Instanzen persistent gemacht werden können.

1.3.3.1 Objektidentität

Objektidentität, Zustand und Botschaften

Jedes Objekt soll als Tripel <Objektidentität (OID), Zustand, Botschaften> auffassbar sein. Durch das, zumindest implizite, Vorhandensein einer OID kann jedes Objekt im System eindeutig identifiziert werden. Diese Indentität bleibt auch bei der Veränderung des Zustandes erhalten. Selbst bei der Löschung eines Objektes wird dessen OID nicht erneut vergeben. OIDs wirken also als sogenannte „Surrogate“.

Objektidentität versus Schlüsselattribute

Im Gegensatz zum Umgang mit Schlüsselattributen erlaubt diese Strategie die Wahrung der einmaligen und eindeutigen Beziehung zwischen dem Realweltobjekt und dem Objekt auf der Datenbank. Würde nämlich die Identifikation des Objektes rein auf dessen Wertausprägung beruhen, so wäre keinesfalls gewährleistet, dass nach dessen Löschung kein anderes Objekt mit ein und derselben Schlüsselwertidentität entstehen könnte. In diesem Fall wäre nachträglich nicht mehr ersichtlich, ob es sich noch um dasselbe Objekt handelte.

1.3.3.2 Zusammengesetzte Objekte

komplexe Datentypen

Mit diesem Konstrukt soll der erwähnten Forderung nach der Repräsentierbarkeit von Objekt-/Unterobjektstrukturen Rechnung getragen werden. Der Wert eines Objektes soll also aus Werten anderer Objekte aufgebaut sein dürfen. Auf diese Weise lassen sich aus primitiven Datentypen beliebig komplexe Datentypen zusammensetzen. Insbesondere soll auch erlaubt sein, Objektteile mehrfach zu verwenden, also diese in mehreren verschiedenen Objekten zu verwenden. Darüber hinaus müssen auch rekursive Beziehungen modelliert werden können (ein Objekt kann sich selber als Teil enthalten).

generische Operatoren

Zur Verwendung derartiger Objekte benötigt man entsprechend auch generische Operatoren, die transitiv auf deren Unterobjekte wirken. Sollen Assoziationen zwischen Objekten dargestellt werden, also Beziehungen, die nicht zu Kompositionsobjekten führen, muss auch ein explizites Beziehungskonzept angeboten werden.

1.3.3.3 Klassen und Klassenhierarchien

Klassenbibliothek

Das Datenbanksystem muss das Klassenkonzept nach den objektorientierten Gesichtspunkten unterstützen. Es reicht nicht aus, eine vorgefertigte Klassenbibliothek in das System einzubauen, vielmehr sollte den Systemadministratoren und -benutzern die Möglichkeit gegeben werden, selber Klassen, insbesondere als Spezialisierungen von gelieferten Basisklassen, zu erstellen.

Die Konzepte der Klassenvererbung und zum Aufbau von Klassenhierarchien sollen, wie in der Objektorientierung vorgeschlagen, im System integriert sein.

1.3.3.4 Berechnungsvollständigkeit

Schleifen, Fallunterscheidungen und Rekursionen

Die der Datenbank zu Grunde liegende Sprache muss berechnungsvollständig sein, d.h. sie muss sich zur Beschreibung von beliebigen Algorithmen eignen. Es müssen mit ihr also Schleifen, Fallunterscheidungen und Rekursionen beschrieben werden können.

Bisherige Implementationen von Datenbanksprachen waren dieser Forderung nicht ausgesetzt, da sie in Applikationen in andere Sprachen eingebettet auftraten. Diese Heterogenität soll nun durch den Zwang zur Definition einer kompletten Sprache ausgeräumt werden.

1.3.3.5 Einkapselung

verbergen von Zuständen und Methoden

Gemäss den Grundsätzen der Objektorientierung müssen Zustände und Methoden von den Objektbenutzern verborgen werden können. Diese Forderung steht vordergründig im Gegensatz zu der, dass gewisse Objekte aufgrund ihres Zustandes - man stelle sich beispielsweise Abfragen auf Objektmengen vor - aufgefunden werden können. (Gesucht ist die Person namens „Andreas“, wobei das Attribut „Name“ in der Klasse „Person“ nicht öffentlich zugänglich sein sollte.)

Definition der Schnittstelle

Durch die Definition der Schnittstelle können die Zugriffe auf derartige Informationen gewährt werden. Obschon dies den Einkapselungsbegriff in gewissem Sinne aufweicht, kann trotzdem die Kontrolle bei Lese- und Schreiboperationen erhalten werden.

Ausserdem kann die Einkapselung in ihrer Strenge - man denke an private und öffentliche Eigenschaften und Methoden - variiert werden. Dem Entwerfer bleibt es überlassen, welche Form dieses „information hidings“ er für den Anwendungsfall als sinnvoll erachtet.

1.3.3.6 Überladen, Überschreiben und spätes Binden

Die Überladung, die Überschreibung und das späte Binden stammen aus den erweiterten gegenüber objektorientierten Systemen geäusserten Bedürfnissen und stellen - jedes Konstrukt für sich - ein Aspekt des Polymorphismus’ dar.

Überladen

Das Überladen gestattet es, denselben Namen für verschiedene Botschaften zu verwenden. Die passenden Methoden unterscheiden sich üblicherweise durch deren Signaturen (eine Unterscheidung auf dem Typ des Rückgabewertes allein ist in den meisten objektorientieren Sprachen nicht unterstützt).

Überschreiben

Das Überschreiben von Methoden in abgeleiteten Klassen dient dazu, ein Verhalten, welches für die Oberklasse bereits definiert ist, in der Unterklasse zu spezialisieren oder anzupassen. Dieses Standardverhalten kann in der Vaterklasse auch abstrakt vorgegeben sein. Mit diesem Mittel wird dem Entwickler ein Werkzeug in die Hand gelegt, welches ihm erlaubt, Botschaften an ganze Objektmengen zu schicken.

spätes Binden

Diese Mengen können auch heterogen sein, d.h. die Mengenelemente müssen nicht zwingenderweise vom gleichen Typ sein. Insbesondere muss auch zur Entwicklungszeit nicht vorgegeben sein, um welchen Typ es sich in einem speziellen Fall handelt. In dieser Situation muss die Fähigkeit des Systems zur späten Bindung, Bindung der Methode an eine Botschaft erst zur Ausführungszeit, ausgenutzt werden.

1.3.4 Ein Standard für objektorientierte Datenbanksysteme

Die „Object Data Mangement Group“ hat in ihrem Standardisierungsvorschlag ODMG 93 folgende Komponenten für den Einsatz von objektorientierten Datenbanksystemen vorgeschlagen:

ODL

1. Ein „Object Definition Language“ (ODL) für die Spezifikation von Objekttypen, ihrer Struktur sowie der Signaturen ihrer Methoden.

OQL

2. Eine „Object Query Language“ (OQL) für die Formulierung von deklarativen Anfragen.

OML

3. Eine Datenmanipulationssprache (OML), die an eine bereits bestehende objektorientierte Programmiersprache (C++, SamallTalk, usw.) angebunden ist, und so die flüchtigen Objekte, welche über Botschaften mit den dauerhaften Objekten kommunizieren, zur Verfügung stellt.

Häufig findet man in marktfähigen Implementationen die dritte Komponente ebenfalls im Datenbanksystem integriert (als eigene Sprache, die meist auf einer bestehenden basiert). O_2 stellt beispielsweise eine vollständige Sprache O_2C für diese Belange zur Verfügung.

1.3.4.1 Objektdefinitionssprache ODL

Die Spezifikation der Datenbankschemata kann in der Sprache ODMG 93-ODL erfolgen. Diese ist als Erweiterung der Interfacedefinitionssprache OMG-IDL zu betrachten [Cattel 96].

Ein Objekttyp wird durch die Angabe von folgendem Code-Fragment definiert:

```
interface ClassName : ParentClasseName
(
                    extent: ClassExtensionName;
                    key(KeyAttribName1, KeyAttribName2, ...)
);
{
attribute Type AttribName1;
attribute Type AttribName2;
...

relatoinship [Set]ForeignClassName RelationName
                              [inverse ForeignClassName::InverseRelationName]
...

Type MethodName1(Type ParameterName1, ...)
...
}
```

Syntaxdiagramm 1-1: Interfacedefinition mit ODMG 93-ODL

Legende

interface

Das Schlüsselwort öffnet eine Schnittstellendefinition, in der die Extension und die Attribute, Relationen und Methoden angegeben sind.

ClassName

Name der Klasse.

ParentClassName

Name der Vaterklasse, von der geerbt wird.

extent

Schlüsselwort für die Deklaration des Extensionsnamens.

ClassExtensionName

Name der Extension für diesen Objekttyp. Für jeden Objekttyp kann explizit eine Extension deklariert werden,

über die der Zugriff auf die Menge aller in der Datenbank gespeicherten Instanzen dieses Typs möglich ist.

key

Definition des Schlüsselattributs oder einer Schlüsselattributskombination für die Definition der Extension.

KeyAttribNameN

Name eines im unteren Teil der Definition angegebenen Attributes, welches allein oder in Kombination mit anderen Attributen den Schlüssel für die Objektsammlung festsetzt.

attribute

Schlüsselwort für die Deklaration der Attribute (Zustandsvariablen).

Type

Typendefinition für ein Attribut, einen Parameter oder eine Methode. Er kann von einem elementaren, in der Datenbank zur Verfügung stehenden Typ, wie **integer**, **string**, **enumeration**, oder von einem zusammengesetzten Typ (**struct** oder eine bereits definierte Klasse) sein. Auch mengenwertige Typen, namentlich **set**, **bag**, **list** und **array**, werden unterstützt.

AttribNameN

Namen der Attribute

relationship

Schlüsselwort für die Deklaration von Beziehungen.

ForeignClassName

Name derjenigen Klasse, zu welcher eine Beziehung aufgebaut werden soll. Dies kann insbesondere auch die eigene sein (Selbstbeziehung).

RelationName

Name der Beziehung. Könnte im übertragenen Sinn auch als Attribut vom speziellen Typ „Relation“ angesehen werden. Da eine Relation auch mengenwertig sein kann, sind alle denkbaren Kardinalitäten für Beziehungen beschreibbar.

inverse

Schlüsselwort für die Deklaration von inversen Beziehungen.

InverseRelationName

Name der inversen Beziehung. Dieser Name muss mit der Klasse, zu welcher man eine reflexive Beziehung unterhalten will, in der Sektion **relationship** deklariert sein.

MethodNameN

Name der Methode.

ParameterNameN

Name des N-ten Aufrufparameters.

inverse Beziehungen und Klassenextensionen

Es fällt auf, dass einige wichtige Elemente für den Umgang mit Datenbanken bereits in die Standardsprache Einzug erhalten haben. So findet man beispielsweise ein Konstrukt für die Definition von inversen Beziehungen. Dank der Deklarationsmöglichkeit von Klassenextensionen erhält man ein Werkzeug in die Hand gelegt, welches vergleichbar mit einem einfachen Sichtenkonzept ist und dem Anwendungsentwickler den Umgang mit den Objektmengen erheblich vereinfacht.

1.3.4.2 Abfragesprache OQL

Mit der Sprache OQL wurde ein Anweisungssatz geschaffen, welcher den deklarativen Zugriff auf objektorientierte Datenbanken ermöglichen soll. Sie basiert auf dem Objektmodell, wie es von der ODMG 93 vorgeschlagen worden ist, und kann sowohl als interaktive Datenbankschnittstelle als auch eingebettet in eine Programmiersprache verwendet werden.
Nachfolgend soll neben dem Syntax auch die Semantik beschrieben sein.

SELECT - FROM - WHERE

Abfragen werden grundsätzlich in der Form **SELECT** ... **FROM** ... **WHERE** abgesetzt, wobei die Ähnlichkeit zu SQL nicht als Kompatibilität zu missverstehen ist. Im **SELECT**-Teil wird das gewünschte Resultat spezifiziert, welches dementsprechend in Form eines Wertes, eines Objektes oder einer Sammlung (Menge, Liste, Multimenge) von Werten, bzw. Objekten zurückgeliefert werden kann. Der **FROM**-Teil definiert zum einen die Objektmenge, idealerweise eine in ODL formulierte Extension, über welche die Datensuche ausgeführt wird, und zum anderen ein Alias, der für die weitere Verwendung des Objektes während der Ausführung die betrachtete Instanz repräsentiert. Schliesslich können in der **WHERE**-Klausel einschränkende Bedingungen für die Selektion formuliert werden.

Innerhalb einer Abfrage können, was als ein entscheidender Vorteil von OQL gegenüber seines relationalen Pendants angesehen werden darf, auch Botschaften an das Objekt integriert sein. Methoden können sowohl im Resultatteil als auch in der Bedingungssektion aufgerufen werden.
In Fällen, in denen man einfachste Anfragen absetzen will, kann auch vom strengen **SELECT** ... **FROM** ... **WHERE** abgewichen werden. So lässt sich beispielsweise die Menge aller Objekte eines gewissen Typs auch direkt durch Angabe des Namens der Extension beschaffen.
Zwar existiert in OQL kein Sichtenkonzept, welches sich mit demjenigen von SQL messen könnte, doch findet man immerhin im Sprachschatz die Möglichkeit, Abfragen unter Zuhilfenahme des **define**-Konstruktes zu definieren und zu benennen. Derartige „named queries“ sind jedoch nicht in der ODL formulierbar, vielmehr handelt es sich dabei um eine Erweiterung von OQL.

exists
for all

Werden als Resultate von Abfragen zumeist Mengen von Werten oder Objekten erstellt, existieren dementsprechend auch darauf wirkende Operationen als fester Bestandteil von OQL. Der Existenzquantor **exists** ... **in** ... erlaubt die qualitative Entscheidung, ob ein ausgezeichnetes Element in der Menge enthalten ist, während der Allquantor **for all** ... **in** ... auf alle Elemente einer Menge wirkt.

sort
group

Für den Umgang mit Mengen stehen noch weitere Ausdrücke zur Verfügung. So kann eine Menge durch **sort** ... **in** ... **by** ... sortiert oder durch **group** ... **in** ... **by** ... **with** ... gruppiert werden, wobei der **with**-Teil ähnlich einer Klausel **having** in SQL eine einschränkende Bedingung enthalten kann.

union
intersect
except

Verschiedene aber gleichartige Menge können durch **union** verbunden (Vereinigungsmenge), durch **intersect** geschnitten (Durchschnittsmenge) oder durch **except** voneinander abgezogen werden (Differenzmenge).

element
flatten

Will man die innere Struktur einer Menge verändern, so existieren **element**, zur Schaffung einer Menge aus einem Wert, **flatten** für das Ausflachen einer homogenen Multimenge zu einer Menge, oder **flatten** zur Überführung einer Liste zu einer Menge als Möglichkeiten.

1.3.4.3 Anbindung an Programmiersprachen

ODL versus OQL

Anders als in der relationalen Welt, in der bekanntlich SQL als eine Obermenge von DDL gesehen werden kann, ist ODL kein Bestandteil von OQL. Insbesondere ist ODL in den meisten Fällen nicht in der Implementation der Datenbank zu finden, vielmehr muss für deren Verwendung eine bestehende Programmiersprache wie C++ oder Smalltalk herangezogen werden.

spezifische Spracherweiterung

Diese stellt dann die Möglichkeit der Schemadefinition als spezifische Spracherweiterung PL-ODL (Programming Language - ODL) zur Verfügung. Dabei ist diese bestrebt, das Datenmodell des Datenbanksystems möglichst eng in sein Typensystem zu integrieren. Damit kann im allgemeinen ein Schema einerseits direkt innerhalb der Datenbankumgebung oder andererseits durch Klassendeklaration in PL-ODL erstellt werden. Eine Schemadefinition in der sprachspezifischen ODL wird von einem Deklarationspräprozessor in das Datenbankschema und in den Deklarationsteil der Programmiersprache, in Form von beispielsweise C++-Header-Dateien, übersetzt.

OML

Im Implementationsteil der Programmiersprache schliesslich sind die Zugriffe auf die gewünschten persistenten Objekte unterzubringen. Um derartige Aufrufe möglichst eng an die Programmiersprache anlehnen zu können, sind von der ODMG 93 auch entsprechende Spracherweiterungen (Object Manipulation Language OML) definiert worden.

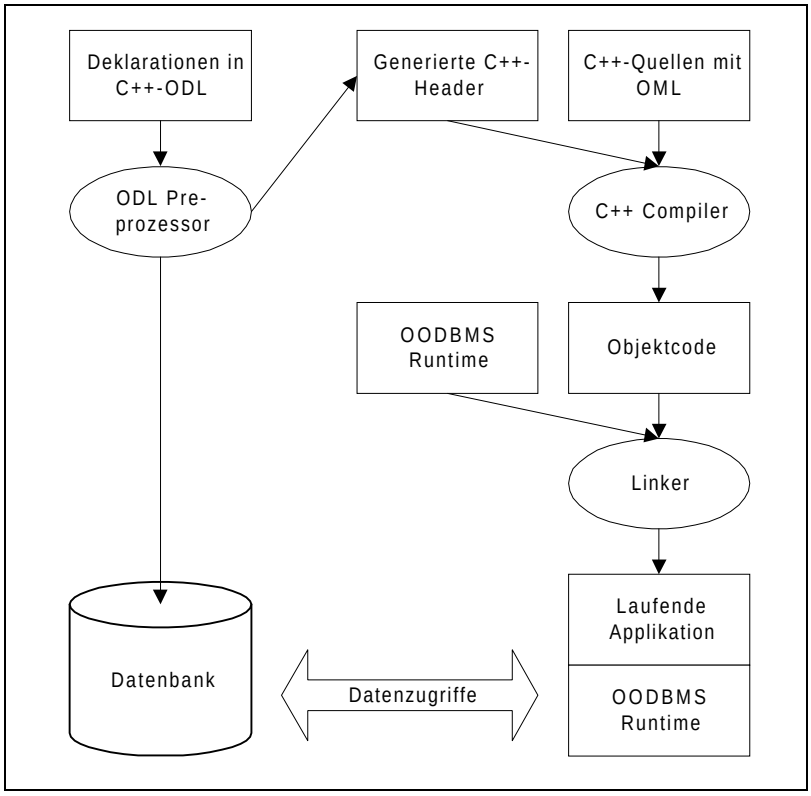


Abbildung 1-1: ODMG-Datenbank mit C++ Anbindung [Grotehen 95, p. 51]

Für weitere Informationen über die C++-spezifische Spracherweiterungen sei auf [Grotehen95] verwiesen.

1.3.4.4 C++-ODL

C++-ODL ist eine Erweiterung des Typensystems von C++. Mit den zusätzlichen Konstrukten wie Ref<T>, Set, Bag, List und Array können Zeiger auf persistente Objekte in der Datenbasis und in Objektsammlungen verwaltet werden.

1.3.4.5 C++-OML

Die C++-OML definiert einige Klassen wie

- Ref für Referenzen auf persistente und transiente Objekte,
- Persistent_Object für Referenzen auf persistente Objekte,
- Collection, Set, Bag, List, Array für Sammlungen,
- Iterator für Iterierung (sequentieller Zugriff),
- Transaction Schnittstelle für Transaktion und
- Database für Verwaltung mehrerer Datenbanken

und einige Operatoren und Funktionen wie

- new, bzw. delete für Allokation, bzw. Deallokation und
- Start, Commit, Abort für Transaktionen

1.3.4.6 C++-OQL

Anfragen können gegenüber einer Sammlung oder auch direkt an eine objektorientierte Datenbank abgesetzt werden. Mit Hilfe einer speziellen Methode
int query(Ref<Collection<T>>& result,
const char* predicate);

formuliert man eine Filterung der Kollektion unter dem Kriterium predicate. Datenbankanfragen werden unter Zuhilfenahme der Methode
int oql(TYPE result, const char* query);
abgesetzt, wobei die Zeichenkette query eine OQL Anfrage definiert und das Resultat in Abhängigkeit der Anfrage als ein Sammlungstyp Ref<Persistent_Object>&, Collection&, etc. oder als ein primitiver Typ char&, int&, etc. zurückgeliefert wird.

1.3.4.7 Zukünftige C++-Anbindungen

Die in ODMG-93 definierten Konzepte machen bezüglich der Ausdrucksmächtigkeit noch Abstriche, weshalb bereits zukünftige Erweiterungen vorgeschlagen worden sind, welche die folgenden zusätzlichen Eigenschaften besitzen sollen:

- Alle Klassen können persistente Instanzen besitzen.
- Die Unterscheidung von Referenzen auf persistente und transiente nicht mehr nötig.
- Die Anfragesprache OQL soll nicht als Spracherweiterung eingebunden sein, sondern in die zugrundeliegende Programmiersprache integriert werden, damit Konstrukte der Anfragesprache frei in der Programmiersprache verwendet werden können und umgekehrt.
- Anstelle eines C++-ODL Präprozessors soll ein C++-ODL/OML Präprozessor zur Verfügung gestellt werden, der C++-Quelldateien mit eingebetteten ODL/OML-Anweisungen verarbeiten und in C++ übersetzen kann.

1.3.4.8 Fazit

Zusammenfassend werden verschiedene Punkte als erfreulich angesehen oder bemängelt.

> *„Der wesentliche Beitrag dieses Vorschlags ist wohl die Beschreibung der Anfragesprache OQL, während sich die anderen Komponenten noch in einem verbesserungsbedürftigen Zustand befinden.“*
> [Grotehen 95, p. 55]

Als verbesserungsbedürftig werden vor allem die Vorschläge bezüglich der ODL gesehen. Zum einen findet man die Definition von ODL und zum anderen die zusätzlichen PL-ODLs, was den Betrachter in Staunen versetzt, zumal in dieser Unterscheidung der Wunsch nach einer programmiersprachen-unabhängigen Datenbankdefinitionssprache (wie SQL) gegenüber demjenigen nach einer möglichst engen Anlehnung an das sprachabhängige Typensystem im Widerspruch steht. ODL ist also nicht so gut gelungen wie IDL (Interface Definition Language), was als Tribut an die Standardisierungsbemühungen der OMG zu sehen ist.

Trotzdem setzt ODMG-93 einen wichtigen Meilenstein in der Entwicklung der objektorientierten Datenbanksysteme und steigert die kommerzielle Akzeptanz dieser Technologie.

1.3.5 Aktuelle Implementierungen

Es gibt bereits eine gewisse Anzahl von marktfähigen Implementierungen von objektorientierten Datenbanksystemen, welche die gemäss obigen Kapitel erforderlichen Eigenschaften aufweisen und sich in der Praxis mehrfach bewährt haben. In [Meier et. al. 95] werden sechs derartige Systeme auf den Prüfstand gehoben und anhand eines Bewertungsrasters benotet. Zusammenfassend seien hier einerseits anhand der Kandidaten die grosse Vielfalt der kommerziellen Produkte veranschaulicht und andererseits die Kriterien der Bewertung beschrieben.

1.3.5.1 Ausgewählte Produkte

Im folgenden sollen einige Produkte kurz charakterisiert werden. Es wird dabei vor allem auf die Besonderheiten eingegangen.

GemStone ist eines der wenigen Datenbanksysteme, welches auf der Linie von Smalltalk aufbaut.

Itasca ist eine objektorientierte Erweiterung von LISP.

O_2 basiert auf dem Wisconsin Storage System WiSS. Es beinhaltet eine spezielle Sprache O_2C für die Objektmanipulation und unterstüzt OQL nach dem Standard ODMG-93.

ObjectStore ist ein von vielen CASE-Werkzeugen verwendetes Speicherungssystem für verteilte Objekte. Die zentrale Sprachschnittstelle bildet C++, in der Zwischenzeit ist aber auch eine Smalltalk-Anbindung implementiert.

Ontos ist das Nachfolgeprodukt von VBase und basiert auf C++. Anfragen an die Datenbank werden über eine proprietäre Sprache ObjectSQL formuliert.

Versant realisiert eine Objektserverarchitektur mit Sprachanbindungen für C++ und Smalltalk. Als Anfragesprache wurde Versant ObjectSQL implementiert.

Alle betrachteten Systeme unterstützen die elementaren Konzepte für objektorientierte Datenbanksysteme, jedoch sind die Konzepte für die Objektorientierung, die Modelleigenschaften, die Sprachaspekte und die Systemarchitektur in unterschiedlicher Ausprägung und Konsequenz zu finden.

1.3.5.2 Bewertungsraster

Aus diesem Grund wurde von [Meier et. al. 95] ein Bewertungsraster vorgeschlagen, der vor allem aus der Sicht eines zukünftigen Datenbankadministrators sowohl bei der Evaluation des geeigneten Systems, also auch bei der Bewertung von Systemevolutionen helfen kann.

Mit Hilfe der nachfolgenden Tabellen, die jeweils ein bestimmtes Kriterium und die beobachtbaren

Ausprägungen einander gegenüberstellt, soll aufgezeigt werden, wie beispielsweise das Datenbanksystem O_2 in dieser Bewertung abgeschnitten hat (die Ausprägungen für O_2 sind jeweils unterstrichen). (In [Meier et. al. 95] ist für jede Gruppe von Eigenschaften ist eine eigene Gegenüberstellung zu finden, was die Bewertungspunkte in einer übersichtlichen Sammlung strukturiert.)

Objektorientierung und Modelleigenschaften

Kriterium	**Ausprägungsarten**
Vererbung	einfach, mehrfach
Polymorphismus	Überschreiben, Überladen, generische Funktionen, generische Klassen
Binden	dynamisch, statisch
Kapselung	Attribute, Attribute optional, Methoden

Tabelle 1-1: Objektorientierung und Modelleigenschaften

O_2 unterstützt ein- und mehrfache Vererbung, währenddem Polymorphismus nur in Form der Überschreibung im Produkt enthalten ist. Dynamisches (spätes) Binden, sowie die Kapselung von Attributen und Methoden sind innerhalb von O_2 gut unterstützt.

Modell und Integrität

Kriterium	**Ausprägungsarten**
Objekte	komplex, zusammengesetzt
Referenzsicherheit	implizit erfüllt, mit „smart pointers“
Rückbezügliche Beziehungen	unterstützt, nur mit zusammengesetzten Objekten
Metamodell	Struktur, z.T. Verhalten, Verhalten
Schemaevolution	Attribute, Methoden, Klassen

Tabelle 1-2: Modell und Integrität

In O_2 können beliebig komplexe Objekte definiert werden, während der Umgang mit zusammengesetzten Objekten nicht vorgesehen ist. Die Referenzsicherheit ist implizit erfüllt, rückbezügliche (inverse) Beziehungen hingegen sind nicht unterstützt. Die Forderung, die Beschreibung von Klassen und Methoden selbst als Metamodell den Anwendungsentwicklern zur Verfügung zu stellen, ist in O_2 zumindest zur Entwicklungszeit erfüllt. Wo sich hingegen eine Schwäche von O_2 abzeichnet, ist im Bereich der Schemaevolution, in welchem der Administrator auf keinerlei Hilfe durch das System zählen kann.

Datenbank- und Abfragesprachen

Kriterium	**Ausprägungsarten**
Sprachunabhängigkeit	gewährleistet, nicht gewährleistet
Interne Datenbanksprache	Smalltalk-Erweiterung, SmalltalkDB, LISP-Erweiterung, C-Erweiterung, C++-Erweiterung
Typenkonzept	schach typisiert, streng typisiert, orthogonal
C++-Anbindung	Definition, Manipulation
Smalltalk-Anbindung	Definition, Manipulation

Weiter Sprachen	C, LISP, Eiffel
Abfragesprache	proprietär, eingebettet, frei, OQL, ObjectSQL
Autorisierung	teilweise unterstützt, voll unterstützt, minimal unterstützt

Tabelle 1-3: Datenbank- und Abfragesprachen

O_2 gewährleistet die Sprachunabhängigkeit und bietet als interne Sprache eine C-Erweiterung names O_2C an. Das Typenkonzept ist nicht nur streng typisiert sondern sogar orthogonal. Mit C++ können persistente Objekte sowohl definiert als manipuliert werden, während eine entsprechende Anbindung durch Smalltalk nicht implementiert ist. Durch die Verwendung von weiteren Sprachen, wie C, LISP, Eiffel sind Zugriffe auf die Datenbestände möglich. Als Abfragesprache ist das standardisierte OQL der ODMG 93 vollumfänglich integriert und kann von der Wirtssprache entweder frei oder eingebettet verwendet werden. In den Bereichen Autorisierung und Sichtenkonzept hingegen besteht noch ein gewisser Handlungsbedarf.

Komponenten der Systemaritektur

Kriterium	**Ausprägungsarten**
Serverkonzept	Objekt-Server, Seiten-Server
OID-Implementation	Surrogat, Wertindex, Pfadindex, Objektindex, Adresse, Objekt-ID
Versionen	möglich, check_in, check_out
Objektverteilung	unterstützt, nicht unterstützt
Transaktionen	langandauernd, geschachtelt

Tabelle 1-4: Komponenten der Systemaritektur

Die Arbeitsverteilung zwischen dem Server- und dem Clientprozess beschränkt sich in O_2 auf die Bereitstellung von physischen Seiten. Die Verteilung von Objekten auf verschiedenen Servern ist unterstützt. Objekte können anhand von Objekt-IDs, Wertindex und Objektindex identifiziert werden. Damit verletzt O_2 die Forderung nach Unveränderbarkeit und Ortsunabhängigkeit. Eine Versionskontrolle ist nicht unterstützt. Auch in Bezug auf die Transaktionsverarbeitung sieht man sich bei der Verwendung von O_2 einem einschneidenden Manko gegenübergestellt. Dies betrifft sowohl langandauernde als auch geschachtelte Transaktionen.

1.3.5.3 Fazit

Zusammenfassend geben die Autoren an, dass objektorientierte Datenbanksysteme einen Reifegrad erreicht haben, der den punktuellen Einsatz solcher Systeme in der Praxis durchaus zu rechtfertigen vermag. Insbesondere aber in den Bereichen wie Schemaevolution und Sprachanbindung sind einige ihrer Exemplare noch stark verbesserungsfähig. Zudem sollen die physischen Aspekte der Clusterung, Indexierung, Objektallokation und

Sperrgranularität von den höherwertige Konzepten klarer getrennt werden.

„Trotz dieser Mängel belegen einige der kommerziell verfügbaren objektorientierten Datenbanksysteme, dass diese zukunftsträchtige Datenbanktechnologie das Arbeiten mit objektorientierten Methoden und Werkzeugen ideal ergänzen kann.“
[Meier et. al. 95, p. 40]

1.3.6 Stand der Technik

Kinderkrankheiten

Die anfänglichen Kinderkrankheiten, an denen die ersten marktreifen Implementationen häufig litten, sind heute vielerorts bereits überwunden und können durchaus als der Geschichte angehörend angesehen werden. Tatsächlich erobern objektorientierte Datenbanksysteme nach und nach auch diejenigen Bereiche der Informationsverarbeitung, in denen sie von den heftigsten Kritikern, meist aus dem Lager der eingefleischten Verfechter der relationalen Paradigmen, als unzulänglich verschrien wurden, namentlich im Umfeld der sehr grossen und langlebigen Datenbeständen (vgl. [Dittrich et. al. 95]).

kontinuierliche Verbesserung

Diesen Umstand haben die Anbieter vor allem der kontinuierlichen Verbesserung des Leistungsverhaltens zu verdanken. Nicht zuletzt hat auch die Definition einer einheitlichen Abfragesprache, welche den Zugriff auf die dauerhaften Objekte erheblich vereinfacht, zu dem verbesserten Ruf beigetragen.

günstige Marktprognosen

In Bereichen, wo der Einsatz von relationalen Systemen bislang nicht zu befriedigenden Resultaten geführt hat, erfreuen sich objektorientierte Ansätze bereits heute einer grossen Beliebtheit. Bedenkt man dazu noch, dass sich heute nicht einmal 20% der auf Rechnern gespeicherten Informationen in Datenbanken befinden [Brodie 89], so lässt sich hier ein Potential von beachtlichem Ausmass ausmachen. Entsprechend günstig präsentieren sich auch die professionellen Marktprognosen [Guilfoyle el. al. 91].

1.3.7 Nutzen von objektorientierten Datenbanksystemen

Abschliessend lassen sich durch die Verwendung von objektorientierten Datenbanksystemen zusammenfassend folgende Aspekte in bezug auf den Nutzen erwähnen:
Objektorientierte Datenmodelle helfen gerade bei der Modellierung von komplexen Sachverhalten, sowohl hinsichtlich struktureller als auch verhaltensmässiger Aspekte. Grosse Systeme werden dadurch einfacher plan-, implementier- und wartbar.

„Objektorientierte Datenbanksysteme sind eine technische und marktpräsente Realität und werden sich mittelfristig durchsetzen.“
[Dittrich et. al. 95], p. 22

Integration relationaler und objektorientierter Systeme

Relationale Systeme werden jedoch keineswegs verdrängt, vielmehr finden Datenbankdienste dank ihrer objektorientierten Vertreter Einzug in Gebiete, die bis anhin gar nicht oder nur schlecht unterstützt wurden. In denjenige Bereichen, in denen Datenbestände bereits durch Datenbanken verwaltet wurden, ergänzen objektorientierte ihre Vorgänger, was darauf hinausläuft, dass der Integration der beiden Systemarten in der Zukunft eine grosse Bedeutung zuzumessen sein wird.

organisatorischer und methodischer Rahmen

Effektivität im Umgang mit der neuartigen Technologie bedingt auch eine entsprechende Anpassung des organisatorischen und methodischen Rahmens. Es gilt dabei zu bemerken, dass die wissenschaftlichen Kenntnisse darüber noch in keinem Fall ausreichend sind.

Die Arbeit an der Definition der Entwurfsmethodik durch, wie in dieser Arbeit beispielsweise vorgeschlagen, adäquate Erweiterung bestehender Vorgehensweisen soll nicht zuletzt eben dieser Anforderung genügen und helfen eine Lücke in der Erkenntnis zu schliessen.

1.4 Konklusion: Objektorientierter Datenbankentwurf

„Objektorientierte Datenbankmodelle nehmen für sich in Anspruch, eine natürlichere Modellierung der Anwendung zu ermöglichen, als im Relationenmodell oder den bisherigen Entwurfsmodellen, meist Versionen des Entity-Relationship-Modells (ER-Modell), üblich.“
[Heuer 93, p. 96]

Ausgehend von diesem Ansatz kann man weitere Vorteile aufführen:

1. Das Entwurfsmodell kann als ein mögliches Implementierungsmodell betrachtet werden, da die verwendeten Konstrukte von den Datenbanksystemen weitgehend unterstützt werden. Es ist also nicht mit Verlusten der Semantik zu rechnen, wie dies in bisherigen Umsetzungstechniken der Fall war.
2. Im Gegensatz zu bisherigen Entwurfsmodellen, welche das Verhalten in aussenstehenden Programmen spezifiziert haben, können diese dynamischen Eigenschaften in den Entwurf mit einbezogen werden.

Vereinigung des Datenbank- und des Softwareentwurfs

Die objektorientierte Modellierung vereinigt den Datenbank- und den Softwareentwurf. Man identifiziert Objekte, ihre Attribute und Methoden, fasst diese zu (Objekt-)Klassen zusammen und strukturiert die so gefundenen Klassen in Klassenhierarchien und Komponentenhierarchien. Klassen können darüber hinaus auch in Bereiche (Subsysteme) aufgeteilt werden. Die derart definierten Strukturen werden von den objektorientierten Sprachen gut unterstützt.

1.4.1 Objektorientierte Analyse und objektorientierter Entwurf

Das Verfahren zur objektorientierten Modellierung eines Anwendungsbereichs muss in die objektorientierte Analyse (was soll modelliert werden?) und den objektorientierten Entwurf (wie soll es modelliert werden?) aufgeteilt werden. Die gängigen

Verfahren lassen sich somit in rein analysierende oder rein entwerfende oder eigenständige Verfahren für objektorientierte Modelle (etwa die Methode von Booch [Booch 94]) einordnen. Die Verfahren müssen drei Problembereiche berücksichtigen:

- die Modellierung von Strukturen (Klassen und ihre Hierarchien),
- die Modellierung von Funktionen (Klassenmethoden) und
- die Modellierung des Gesamtverhaltens des Systems.

Die gängigen objektorientierten Analyse- und Entwurfsverfahren unterstützen diese Bereiche sehr gut. Für objektorientierte Datenbankmodelle hingegen müssen einige weitere Eigenschaften einbezogen werden. So muss der objektorientierte Datenbankentwurf

1. Objektevolution (Rollenwechsel eines Objektes),
2. Schemaevolution (Veränderung des Schemas unter Beibehaltung der darin gespeicherten Objekte und deren Informationen) und
3. Sichten (durch Abfragen abgeleitete Klassen)

modellieren können. Die beiden ersten Punkte sind für langfristig angelegte Datenbankanwendungen von zentraler Bedeutung, während sich der dritte Punkt auf eine typische Datenbankfunktion bezieht. Die in einer Datenbasis abgelegten Informationen werden häufig durch Sichtendefinitionen für einen bestimmten Verwendungszweck aufbereitet. Derartige Sichtenklassen spezifiziert man hingegen selten in den Entwurfsdiagrammen oder den Schemata, weil sie die Darstellungen, bzw. Datenbasen unnötig durch redundante Informationen aufblähen.
Während die Integration des bisher getrennten Entwurfs von Strukturen und Funktionen und die Beschreibung des Gesamtverhaltens eines Systems gut gelungen sind, sind datenbankspezifische Bereiche in den gängigen, eher auf objektorientierte Programmiersprachen zugeschnittenen Verfahren nicht oder nicht ausreichend berücksichtigt.

TEIL I
DEIMOS

DEIMOS - **DE**s**I**gn **M**ethod for **O**bject-oriented database **S**ystems
DEIMOS - Gott der Furcht, ständiger Begleiter des Ares, Gott des Krieges.
„Homer sagt, nichts erfreue ihn so sehr wie wildes Kampfgeschrei und das Getümmel der Schlacht. Bewaffnet von Kopf bis Fuss, die mächtige Lanze schwingend, auf dem Haupte den Helm mit dem wehenden Busch, stürmt er über das Blachfeld und sät Tod und Verderben, Phobos, der Schrecken, Eris, der Streit, die Keren, furchtbare Göttinnen des Schlachtentodes sind seine Begleiter. Seine Wildheit und Gewalttätigkeit ist allen Göttern, sogar seinem Vater Zeus verhasst.“ [Peterich 58, pp. 27]

Wenn im Zentrum der Betrachtung nicht eine bestimmte Entwurfsmethode und entsprechend eine konkrete Datenbankimplementation steht, läuft die Diskussion über die Schwachstellen von Vorgehensweisen und deren Behebung in Gefahr, auf einer Metaebene abzulaufen und für den Leser schwammig zu erscheinen, d.h. ohne konkrete Aussagen zu beinhalten.
Aus diesem Grund muss aus der Vielzahl der bekannten Entwurfsmethoden eine bestimmte herausgegriffen und auf ein gewähltes Datenbanksystem abgebildet werden. Zudem muss, um Schwachstellen auf der logischen Ebene finden zu können, eine konkrete Datenbankimplementierung untersucht werden. Darüber hinaus ist es unerlässlich, für die Beschreibung des zweiten Teils auf eine bestimmte Schemadefinitionssprache abzielen zu können.

Auswahl der Entwurfsmethode

Es existiert eine Vielzahl objektorientierter Entwurfsmethoden (vgl. “

Entwurf

> Während sich die Analyse auf die Definition der Objekte beschränkt, werden in der Phase des Entwurfes die Beziehungen zwischen den identifizierten Objekten unter die Lupe genommen. Als Typen sind strukturelle Abhängigkeiten (Spezialisierung, Generalisierung, Enthaltung) und Interaktionen (Meldungsverbindungen) zu unterscheiden.

Übersicht der objektorientierten Analyse- und Entwurfsmethoden” auf Seite 3), welche sich unterschiedlicher Beliebtheit erfreuen. Unter den am breitesten anerkannten Methoden finden sich die Vorschläge von Booch [Booch 94] und Rambough (OMT) [Rambough et. al. 91].
Die vielen Methoden unterscheiden sich meist nur geringfügig. Die Unterschiede liegen entweder im Verwendungszweck (einige sind eher für die Analyse, andere eher für den Entwurf geeignet) oder in der unterschiedlichen Detaillierung bei der Betrachtung eines bestimmten Aspektes. Häufig sind die Abweichungen zweier Methoden sogar auf das unterschiedliche Aussehen semantisch äquivalenter Konstrukte in einer bestimmten Notation beschränkt. Vielfach bewegen sich verschiedene Methoden im Rahmen ihrer jeweiligen Weiterentwicklung aufeinander zu. Der berühmteste dieser Fälle ist der von Booch’s Methode und OMT, welcher auch von den Autoren der Methoden erkannt worden ist und diese veranlasst hat, gemeinsam eine vereinheitlichte Methode zu definieren. Das Resultat dieser Vereinheitlichungsanstrengungen wurde als UM (Unified Method), Version 0.9 der Öffentlichkeit zugänglich gemacht. Da Jacobson (OOSE) als weiterer Autor in das Gremium berufen worden war, wurde UM zugunsten der neuen Methode UML (Unified Modeling Language) zurückgestellt .
In die engere Auswahl wurden Booch, OMT und UML aufgenommen. UML war aber zu Beginn der Diplomarbeit nicht in einer konsolidierten Version verfügbar, weshalb sie aus dem Rennen ausscheiden musste. Welche von den übrigbleibenden

Alternativen gewählt wurde, war im Grunde unerheblich, da bei der Fertigstellung der Arbeit keine mehr aktuell sein wird. Deshalb sei zum Schluss der Arbeit dem Unterschied zwischen UML und Booch ein Anhang gewidmet.
Für die folgenden Ausführungen wird Bezug auf die Methode von Booch genommen.

Auswahl des Datenbanksystems

Zur Auswahl stehen eine Vielzahl von aktuellen Implementierungen. In [Meier et. al. 95] wurden die verschiedenen Produkte einander gegenübergestellt und unter mannigfaltigen Kriterien beleuchtet. O_2 sticht aus dieser Auswahl darum hervor, weil es in allen Kategorien gut abschneidet, ohne in einem bestimmten Bereich schwere Unzulänglichkeiten zu besitzen. Darüber hinaus wird O_2 im Institut für Informatik der Universität Zürich eingesetzt. Die dort beschäftigten Mitarbeiter haben grosse Erfahrung in der Arbeit mit diesem System, was sich beispielsweise in den diversen Implementationen (vgl. [Geppert 96]) äussert.

2 Die Methode von Booch

2.1 Einleitung

In den folgenden Kapiteln, insbesondere den Kapiteln „Schwachstellenanalyse“ und „Die Methode DEIMOS“, wird häufig Bezug auf die Methode von Booch genommen, weshalb in diesem Kapitel die wichtigsten Elemente und Prozesse der Methode von Booch zusammengefasst werden sollen.
Das Kapitel ist eine qualifizierte Zusammenfassung des Buchteils „The Method“ aus [Booch 94, pp. 169-266], wobei die Notation nicht vollständig und das Vorgehen nur prinzipiell wiedergegeben sind.
Eine Methode ist als Gesamtheit aller Notationen und des entsprechenden Vorgehens zu betrachten, weshalb dieses Kapitel - analog zum booch’schen Buchteil - in einen Abschnitt „Notation“ und einen Abschnitt „Vorgehen“ unterteilt ist.

2.2 Die Notation

Das Zeichnen eines Diagramms macht nicht die Analyse oder das Design aus. Ein Diagramm zeigt vielmehr eine Ansicht des Systemverhaltens (Analyse) oder die Vision einer Architektur (Design). Häufig reift ein System im Kopf des Entwerfers und wird lediglich durch die Notierung auf einem Medium wie Wandtafel, Tischtuch, Serviette, Windel oder Rückseite eines Briefumschlages [Shear 88] festgehalten.

Beschreibung der architektonischen Vision

Trotzdem ist es wichtig, eine wohldefinierte Notation für die Entwicklung von Software zu haben. Erstens wird es dem Designer durch die Verwendung einer Standardnotation möglich, ein Szenario oder eine architektonischen Vision zu beschreiben und diese Entscheidungen unmissverständlich anderen involvierten Personen und Institutionen zu kommunizieren. („Zeichnen Sie einen elektrischen Schaltplan, und jeder Elektriker der Welt wird ihn lesen können.“) Zweitens erwähnt der Mathematiker Whitehead [Whitehead 58], dass nur die Anwendung einer guten Notation erlaubt, sich auf die weiterführenden Probleme zu konzentrieren, da sie den Kopf von unnötiger Arbeit freihält. Und drittens hilft eine ausdrucksstarke Notation, Fehler im Entwurf frühzeitig zu eliminieren, indem die Konsistenz und die Korrektheit durch automatisierte Werkzeuge überprüft werden kann.

2.2.1 Die Elemente der Notation

2.2.1.1 Die Notwendigkeit mehrfacher Ansichten

Es ist unmöglich, alle Details eines komplexen Systems in einer einzigen Ansicht zu fassen. Kleyn und Gingrich [Gingrich et. al. 88] vergleichen dieses Phänomen mit der Übertragung eines Fussballspiels: man muss mit verschiedenen Kameras aus verschiedenen Winkeln beobachten, um die gesamte Handlung erfassen zu können. Jede Kamera gibt dabei zu einem gewissen Zeitpunkt einen speziellen Gesichtspunkt des Geschehens wieder, welcher gerade im Interesse des Zuschauers liegt, was beim Einsatz einer einzigen Übersichtskamera nicht möglich wäre.

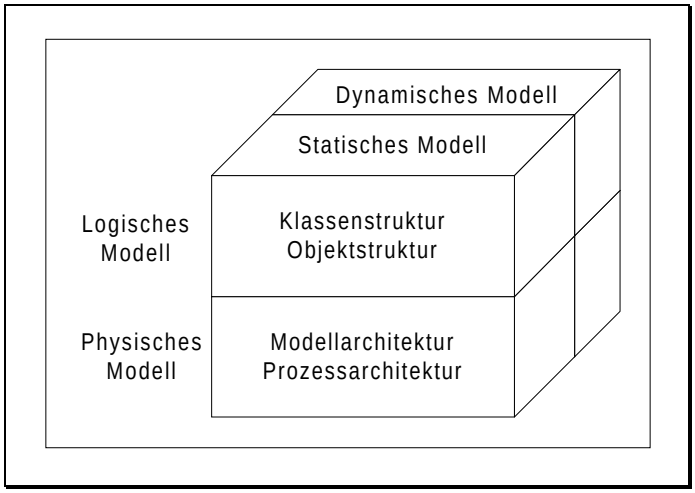


Abbildung 2-1: Die Modelle des objektorientierten Entwurfes [Booch 94, p. 172]

Ausdrucksstärke und Komplettheit

Die Ergebnisse von Analyse und Design sind im oben abgebildeten Würfelmodell (Abbildung 2-1) ausgedrückt. Damit sind sie ausdrucksstark genug, um alle interessanten strategischen und taktischen Entscheidungen zu fassen, sowohl während der Systemanalyse als auch während der Formulierung ihrer Architektur, und komplett genug, um als Vorlagen für die Implementation für fast jede objektorientierte Sprache zu dienen.

Der Umstand, dass die Notation detailliert ist, bedeutet nicht, dass jeder Aspekt von ihr stets zu berücksichtigen ist. Häufig ist auch nur eine Untermenge aller Ansichten nötig, um einen grossen Prozentsatz der Analyse und des Designs auszudrücken. Man sollte sogar nur diejenigen Sichten anwenden, die zum Verständnis beitragen. So wie es gefährlich ist, die Anforderungen überzuspezifizieren, ist es gefährlich, die Lösung zu detailliert zu definieren.

Sprachunabhängigkeit

Oft benutzen Programmiersprachen verschiedene Sprachelemente, um dasselbe Konzept umzusetzen. Die Notation sollte aber weitgehend sprachunabhängig sein. Trotzdem kann die Situation entstehen, dass gewisse Konstrukte der Notation von der betrachteten Zielsprache nicht unterstützt werden. In diesen Konstellationen sollte deshalb davon abgesehen werden, die nicht umsetzbaren Konstrukte in den Entwurf mit einzubeziehen.

2.2.1.2 Modelle und Sichten

logische/physische und statische/dynamische Sicht

Wie in der Abbildung 2-1 angedeutet, sollen Klassen und Objekte in zwei Dimensionen betrachtet werden. Die eine Dimension stellt deren logische und physische Sicht einander gegenüber, während die zweite Dimension deren statische und dynamische Sicht vergleicht. Beide Dimensionen sind aber notwendig, um die Struktur und das Verhalten eines objektorientieren Gesamtsystems zu spezifizieren.

Für jede Dimension ist eine Reihe von Diagrammen vorgesehen. In diesem Sinne repräsentiert das Systemmodell die Gesamtsicht für die Klassen, Beziehungen, usw., während die einzelnen Diagramme eine Projektion dieser Totalen darstellen. Daraus folgt, dass die

jeweiligen Diagramme - wenn sich das Modell in einem stabilen Zustand befindet - sowohl mit dem Systemmodell als auch untereinander konsistent sein müssen.

2.2.1.3 Logische vs. physische Modelle

Die logische Sicht beschreibt die Existenz und Bedeutung der Schlüsselabstraktionen und Mechanismen, die den Problemraum beschreiben oder die Systemarchitektur definieren. Das physische Modell beschreibt die konkrete Software- und Hardwarekomposition im Kontext des Systems oder der Implementation.
Während der Analyse müssen folgende Fragen beantwortet werden:

- Welches ist das Verhalten des Systems?
- Was sind die Rollen und die Verantwortlichkeiten der Objekte, damit dieses Verhalten erzielt wird?

Objektdiagramme und Klassendiagramme

Während der Analyse werden Szenarien verwendet, um das Verhalten eines Systems zu untersuchen. Im logischen Modell werden Objektdiagramme herangezogen, um diese Szenarien zu beschreiben. In den Klassendiagrammen werden die Abstraktionen der Objekte und deren gewöhnliches Verhalten strukturiert beschrieben.

Während des Designs müssen in Abhängigkeit der Systemarchitektur die folgenden Fragen beantwortet werden:

- Welche Klassen existieren und wie sind die Klassen verbunden?
- Welche Mechanismen werden verwendet, um die Zusammenarbeit der Klassen zu regulieren?
- Wo sollen Klassen und Objekte deklariert werden?
- Zu welchem Prozessor soll ein Prozess zugeordnet werden und wie sollen die Prozesse für einen bestimmten Prozessor gesteuert werden?

Die Antworten auf diese Fragen sind in den folgenden Diagrammen unterzubringen:

- Klassendiagramme
- Objektdiagramme
- Moduldiagramme
- Prozessdiagramme

2.2.1.4 Statische versus dynamische Semantik

Erzeugung, Löschung und Senden von Meldungen

Die obigen Diagramme sind eher statischer Natur und genügen daher nicht für die Beschreibung der Ereignisse, die sich in Softwaresystemen dynamisch ergeben. Unter Ereignis sei in diesem Zusammenhang beispielsweise die Erzeugung, bzw. Löschung eines Objektes, das Senden von Meldungen zwischen Objekten oder die von äusseren Entitäten verursachten Aktionen, die gewisse Ereignisse für ein oder mehrere Objekte auslösen, zu verstehen. Es überrascht dabei nicht, dass die Beschreibung dieser dynamischen Ereignisse in einem statischen Medium wie einem Blatt Papier nicht gerade einfach ist.

In der objektorientierten Softwareentwicklung wird die dynamische Semantik eines Systems mit Hilfe von zwei weiteren Diagrammen ausgedrückt:

- Zustandsübergangsdiagramm
- Interaktionsdiagramm

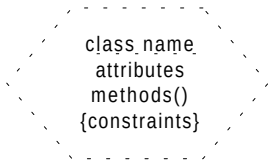
zeit- und ereignisorientierte Meldungsverarbeitung

Jede Klasse kann ein Zustandsübergangsdiagramm für die Beschreibung der ereignisgesteuerten Verhalten der Instanzen haben. Auf ähnliche Weise werden die Objektdiagramme, die ein Szenario beschreiben, mit den Interaktionsdiagrammen verbunden, um die zeit- und ereignisorientierte Meldungsverarbeitung zu spezifizieren.

2.2.2 Klassendiagramme

Ein Klassendiagramm wird verwendet, um die Existenz von Klassen und deren Beziehungen in der logischen Sicht eines Systems darzustellen. Ein einzelnes Klassendiagramm repräsentiert dabei eine Ansicht der Klassenstruktur eines Systems. Während der Analyse werden Klassendiagramme verwendet, um die Rollen und Verantwortlichkeiten der Entitäten zu identifizieren, welche das Verhalten des Systems bestimmen. Im Entwurf schliesslich werden Klassendiagramme herangezogen, um die Strukturen zu erhalten, welche das System formen. Die beiden essentiellen Elemente eines Klassendiagramms sind die Klassen und deren Beziehungen.

2.2.2.1 Klassen und ihre Beziehungen



Klassen werden durch eine Wolke (der Einfachheit halber werden im Folgenden für die Abbildungen die Wolken durch Hexagone ersetzt) repräsentiert und tragen einen eindeutigen Namen. Für gewisse Klassen kann es sinnvoll sein, einige ihrer Attribute und Methoden anzugeben. In den seltensten Fällen trägt es zur Übersicht bei, wenn sämtliche Attribute und Methoden der Klassendarstellung einbeschrieben werden. Die Zeichnung zeigt also nur eine auf die wesentlichen Elemente eingeschränkte Sicht der Klassendefinition.

Namen, Typ und Standardwert

Attribute werden in einem sprachunabhängigen Syntax angegeben und bestehen entweder aus einem Namen, einem Typ oder beidem und können optional einen Standardwert besitzen. Zugelassen sind also folgende Attributsdeklarationen:

A	nur Attributname
B	nur Attributtyp
A : B	Attribut mit Name und Typ
A : B = C	Attribut mit Name, Typ und Standardwert

Der Name eines Attributes muss im Kontext einer Klasse eindeutig sein.

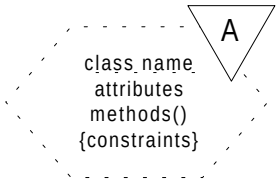
Methoden und Signaturen

Die Fähigkeiten einer Klasse werden durch deren Methoden beschrieben. Methoden werden innerhalb von Klassenikonen normalerweise mit deren Namen beschrieben und durch die Klammerung von den Attributen unterscheidbar gemacht. In manchen Fällen ist die zusätzliche Angabe von Teilen oder der gesamten Signatur sinnvoll.

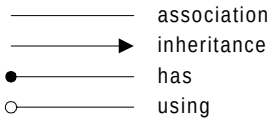
M()	nur Methodenname

R M(Arg) Methode mit Argumenten und Rückgabewert

Der Name einer Methode muss im Kontext der Klasse eindeutig sein. Wird von der betrachteten Implementationssprache die Methodenüberladung unterstützt, können gleichnamige Methoden mit unterschiedlicher Signatur koexistieren.



Klassen, von denen keine Instanzen gebildet werden können, werden als abstrakte Klassen bezeichnet. Für die Darstellung derartiger Klassen wird das normale Klassenkonstrukt um ein Dreieck mit einem einbeschriebenen A (für abstrakt) erweitert.



Klassen stehen selten allein. Vielmehr arbeiten sie in verschiedenen Arten miteinander. Die essentiellen Verbindungen zwischen Klassen sind Assoziation, Vererbung, Besitz- und Verwendungsbeziehung.

Jede Beziehung kann eine textuelle Bezeichnung tragen, welche den Namen der Beziehung oder deren Verwendungszweck dokumentiert.

Kardinalitäten

Die Assoziation beschreibt eine semantische Verbindung zwischen zwei Klassen ohne deren genauen Typ anzugeben. Klassen können auch zu sich selber eine Assoziation besitzen (reflexive Assoziation) oder mehrere Assoziationen zu derselben Klasse unterhalten. Assoziationen sind im weiteren mit Kardinalitäten behaftet, welche den folgenden syntaktischen Regeln genügen sollen:

1	genau eine
N	beliebig viele (keine, eine oder mehr)
0..N	keine oder mehr
1..N	eine oder mehrere
0..1	keine oder eine
3..7	spezifizierter Bereich
1..3,7	spezifizierter Bereich, oder genaue Anzahl

Die Kardinalitätsbezeichnung ist am Ende der Assoziation angegeben und beschreibt somit die Anzahl der Verbindungen, welche von der Ausgangsklasse zu der Zielklasse erlaubt sind. Fehlt eine Kardinalitätsangabe, so wird eine beliebige Anzahl erlaubter Assoziationen angenommen.

Vererbungs-, Besitz- und Verwendungs-beziehungen

Die drei verbleibenden Beziehungen werden als Verfeinerungen der generellen Assoziation gezeichnet. In der Tat werden während des Entwurfes erst undefinierte Verbindungen zwischen zwei Klassen erkannt und durch taktische Entscheidungen in Vererbungs-, Besitz- oder Verwendungsbeziehungen umgesetzt.

Vererbungsbeziehung

Die Vererbung beschreibt eine Generalisierungs- / Spezialisierungsbeziehung und wird mit einem Pfeil dargestellt. Die Pfeilspitze zeigt dabei auf die Superklasse während das entgegengesetzte Ende von der Subklasse ausläuft. Durch die Einrichtung einer Vererbungsbeziehung werden die Struktur und das Verhalten der Superklasse auf die Subklasse übertragen. In Abhängigkeit der betrachteten Implementationssprache können Klassen von einer Klasse (Einfachvererbung) oder mehreren Klassen (Mehrfachvererbung) abgeleitet sein. Beim Entwurf der Vererbungshierarchie muss darauf geachtet werden, dass keine Zyklen entstehen.

Besitzbeziehung

Die Besitzbeziehung beschreibt eine Aggregation. Dabei besitzt die Verbindung beim Aggregat einen ausgefüllten Kreis. Die Instanzen der Klasse am anderen Ende der Beziehung bilden die Teile, aus welcher das Aggregat aufgebaut ist. Reflexive und zyklische Aggregation ist zugelassen, da mit ihr nicht zwingenderweise das physische Enthaltensein ausgedrückt wird.

Verwendungsbeziehung

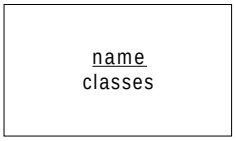
Die Verwendungsbeziehung beschreibt eine Kunden/Anbieter-Verbindung, wobei diejenige Klasse, welche den leeren Kreis enthält, der Kunde ist. Diese Assoziation wird typischerweise verwendet, um zu beschreiben, dass die Kundenklasse Methoden enthält, welche Instanzen der Anbieterklasse entweder als Argumente oder als Parameter besitzen.

2.2.2.2 Klassenkategorien

Dekomposition

Für die Dekomposition eines Systems ist die Klasse zwar ein notwendiges aber nicht ausreichendes Konstrukt. Wenn ein System zirka zehn Abstraktionen übersteigt, drängt sich die Unterteilung des Systems in kollaborierende, aber lose gekoppelte Bereiche auf. Diese Bereiche werden Klassenkategorien genannt.

Die meisten objektorientierten Programmiersprachen (mit Ausnahme von Smalltalk) unterstützen linguistisch keine derartigen Konstrukte. Jedoch wird dem Entwerfer damit ein wichtiges architektonisches Element in die Hände gelegt, welches nicht direkt mit der Implementationssprache ausgedrückt werden kann. Klassen und Klassenkategorien können im selben Diagramm gezeichnet werden. Meistens werden aber Klassendiagramme verwendet, welche ausschliesslich Klassenkategorien beinhalten, um auf einer hohen Stufe die logische Architektur zu repräsentieren.



Eine Klassenkategorie ist ein Aggregat von einerseits Klassen aber andererseits auch anderen Klassenkategorien. Jede Klasse muss dabei einer einzigen Klassenkategorie angehören oder auf der obersten Ebene des Diagramms angegeben sein. Klassenkategorien werden durch Rechtecke repräsentiert und enthalten den Namen sowie die beinhalteten Klassen oder Klassenkategorien.

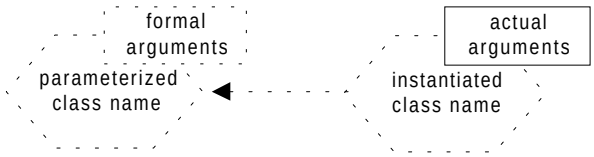
Beziehungen zwischen den Klassenkategorien werden als Verwendungsbeziehungen aufgefasst. Aus diesem Grund wird das bereits eingeführte Konstrukt konsistent

weiterverwendet. Werden Klassen einer Klassenkategorie von allen oder fast allen anderen Klassenkategorien verwendet, können diese auch als Mitglieder einer globalen Klassenkategorie modelliert werden.

vertikalen und horizontale Unterteilung

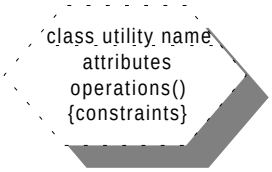
Mit Hilfe der Klassenkategorien kann neben der vertikalen Unterteilung eines Systems in Teilbereiche auch die horizontale Trennung in verschiedene Abstraktionsschichten erreicht werden. Somit bietet das Klassendiagramm auch eine nützliche Visualsierung der Systembereiche und -schichten.

2.2.2.3 Parametrisierbare Klassen



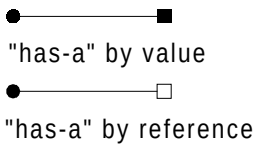
Parametrisierbare Klassen sind Bestandteil von vielen objektorientierten Sprachen und beschreiben eine Familie von Klassen, deren Struktur und Verhalten unabhängig von den formalen Parametern definiert ist. Durch die Instantiierung werden die formalen Parameter mit aktuellen ersetzt und eine konkrete Implementation eines Familienmitgliedes gebildet, welche dadurch Instanzen ausbilden kann.

2.2.2.4 Hilfsklassen



Verschiedene objekorientierte Programmiersprachen erlauben die Implementation von sowohl prozeduralen als auch objektorientierten Programmteilen. Die ungebundenen Prozeduren, auch freie Unterprogramme genannt, werden in Hilfsklassen modelliert. Hilfsklassen werden aber auch herangezogen, um Klassen zu beschreiben, welche nur aus statischen Methoden und Attributen bestehen (für Programmiersprachen, die keine prozeduralen Teile enthalten dürfen) und demnach keine sinnvollen Instanzen ausbilden können.

2.2.2.5 Physisches Enthaltensein



Die Aggregation ist eine verstärkte Form der Assoziation, welche nichts über das physische Enthaltensein der Teile im Aggregat aussagt, und erlaubt insbesondere eine Navigation vom Aggregat zu seinen Teilen. Die Wahl der Aggregation ist üblicherweise ein Analyse- oder architektonischer Entscheid. Insbesondere ist der Aggregationstyp des physischen Enthaltenseins aus zwei Gründen ein taktischer Entschluss. Zum einen spielt er bei der Erzeugung und Zerstörung des Aggregates eine Rolle und zum anderen ist er für die Generierung von sinnvollem Programmcode aus dem Entwurf notwendig.

Es muss zwischen zwei verschiedenen Arten von physischem Enthaltensein unterschieden werden:

By value

Das Aggregat enthält den Teil als einen Wert (die

Aggregationsbeziehung trägt ein ausgefülltes Quadrat auf der Seite des Teils).
By reference
Das Aggregat enthält eine Referenz auf den enthaltenen Teil (die Aggregationsbeziehung trägt ein leeres Quadrat auf der Seite des Teils).
Manche objektorientierte Programmiersprachen (z.B. Smalltalk) implementieren alle physischen Enthaltenseinsbeziehungen als Referenzen.

2.2.2.6 Schlüssel

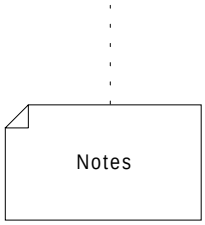
[KeyAttribute]

Assoziationen können mit Schlüsselattributen versehen sein. Der Schlüssel ist ein Attribut, welches ein bestimmtes Zielobjekt eindeutig identifizieren soll. Das Schlüsselattribut wird also verwendet, um durch die Menge der Teilobjekte zu navigieren und ein bestimmtes Exemplar ausfindig zu machen. Im Allgemeinen muss ein Schlüssel mit einem Attribut der Teilklasse, welche Ziel der Assoziation ist, übereinstimmen. Mehrfache Schlüssel sind zugelassen, die Werte müssen aber eindeutig sein.

2.2.2.7 Einschränkungen (Constraints)

Eine Einschränkung ist ein semantischer Ausdruck, welcher bezüglich der Klasse als Invariante zu betrachten ist und deshalb stets erfüllt sein muss. Stets bedeutet in diesem Zusammenhang, dass die Bedingung gelten muss, wenn sich das System in einem stabilen Zustand befindet (während Zustandsübergängen ist es erlaubt, die Invarianten temporär zu verletzen).
Einschränkungen sind in geschweifte Klammern eingefasst und können sowohl für Klassen als auch für Assoziationen formuliert werden.

2.2.2.8 Notizen



Während der Analyse und dem Entwurf werden laufend Annahmen gemacht und Entscheidungen getroffen, welche nicht direkt einem Konstrukt einbeschrieben werden können und deshalb häufig im Kopf des Entwerfers bleiben. Diese Praktik ist sehr gefährlich, weshalb das Diagramm um ein Notizenelement bereichert wird, in welchem derartige Informationen niedergeschrieben werden können. Die Notizen können dabei - dargestellt durch eine gestrichelte Verbindungslinie - bezug auf ein bestimmtes Element im Diagramm nehmen oder für das gesamte Diagramm gelten.

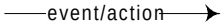
2.2.3 Zustandsübergangsdiagramme

Ein Zustandsübergangsdiagramm wird verwendet, um den Zustandsbereich einer bestimmten Klasse, die Ereignisse, welche einen Zustandsübergang verursachen, und die Aktionen, die aus einem Zustandswechsel resultieren, zu veranschaulichen. Die hier beschriebenen Diagramme sind stark angelehnt an die Arbeit von [Harel 87]. Ein bestimmtes Diagramm repräsentiert eine Ansicht des dynamischen Modells einer einzelnen Klasse oder des gesamten Systems. Nicht jede Klasse hat ein signifikantes

ereignisgesteuertes Verhalten, weshalb diese Art von Diagrammen nur für Klassen mit einem derartigen Verhalten oder für das Gesamtsystem sinnvoll sind. Während der Analyse werden das Gesamtsystemdiagramm für die Modellierung des dynamischen Verhaltens des Systems verwendet, während im Entwurf die Zustandsübergangsdiagramme der einzelnen Klassen im Zentrum des Interesses stehen. Die beiden zentralen Elemente des Diagramms sind die Zustände und die Zustandsübergänge.

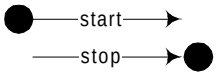
2.2.3.1 Zustände und Zustandsübergänge

Der Zustand repräsentiert das Resultat des Verhaltens eines Objektes. Zu einem gewissen Zeitpunkt besteht der Zustand aus allen Eigenschaften (statisch) und aus den aktuellen Werten der Eigenschaften (dynamisch). Unter Eigenschaften sei hier die Gesamtheit der Attribute und der Beziehungen eines Objektes zu verstehen. Jeder Zustand muss einen im Kontext eindeutigen Namen tragen. Zudem werden in der Zustandsikone die möglichen Aktionen aufgeführt.



Ein Ereignis ist ein Vorkommnis, welches das System veranlasst, seinen Zustand zu wechseln. Dieser Wechsel wird Zustandsübergang genannt. Jeder Übergang verbindet zwei Zustände. Üblicherweise gehen von einem Zustand mehrere Übergänge aus, welche eindeutig identifizierbar sein müssen, damit beim Eintreten eines Ereignisses nicht mehr als ein Übergang ausgelöst wird.

Eine Aktion ist eine Operation, welche sich üblicherweise im Aufruf einer Methode, in der Auslösung eines Ereignisses oder im Starten oder Stoppen einer Aktivität äussert. Für die Modellierung des Zeitverhaltens wird dabei angenommen, dass der Aufruf einer Methode oder das Auslösen weiterer Ereignisse keine Zeit in Anspruch nimmt, während die Aktivität eine gewisse Zeitspanne für dessen Komplettierung benötigt.



In jedem Zustandsübergangsdiagramm muss genau ein Startzustand enthalten sein. Dieser wird mit einem ausgefüllten Kreis dargestellt und durch einen nicht bezeichneten Übergangspfeil mit einem Zustand verbunden. Üblicherweise erreicht ein System nie einen Endzustand, vielmehr befindet es sich in einem der definierten Zustände, bis das umspannende Objekt zerstört wird. Soll dennoch ein Endzustand modelliert werden, kann dem Diagramm ein Endknoten - dargestellt mit einem ausgefüllten Kreis - hinzugefügt werden, zu welchem Übergangspfeile, die ein Ereignis für das Erreichen des Endzustandes beschreiben, zu zeichnen sind.

2.2.3.2 Weiterführende Konzepte

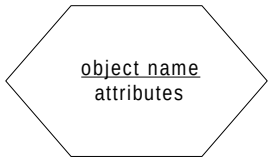
Die bisher beschriebenen Elemente sind meist nicht ausreichend für die Modellierung von komplexen Systemen. Aus diesem Grund müssen die Diagramme erweitert werden, um die Semantik der Harel'schen Zustandsdiagramme nachbilden zu können. Einige der zusätzlichen Konstrukte sollen nachfolgend beschrieben werden:

Zustandsaktionen
Beim Eintritt in einen Zustand bzw. Austritt aus einem Zustand können Aktionen gestartet bzw. gestoppt werden. Derartige Zustandsaktionen werden als Aktionen in der Zustandsikone durch die Verwendung der Schlüsselwörter entry und exit niedergeschrieben.
Bedingte Zustandsübergänge
Zustandsübergänge werden üblicherweise durch Ereignisse beschrieben. Reicht ein Ereignis als Voraussetzung für einen Zustandsübergang nicht aus, kann sie durch die Angabe einer bool'schen Bedingung, welche in eckigen Klammern eingefasst dem Ereignisnamen nachgestellt wird, erweitert werden.
Eingeschlossene Zustände
Durch die Einschliessung einer Gruppe von Zuständen in einen höher abstrahierten Superzustand kann dem Zustandsübergangsdiagramm eine gewisse Tiefe verliehen werden. Derartige Superzustände können vor allem die Strukturierung eines Diagramms fördern und bilden wiederverwendbare Komponenten für die Beschreibung von stets wiederkehrenden Mustern.

2.2.4 Objektdiagramme

Objektdiagramme werden verwendet, um die Existenz von Objekten und deren Beziehungen im logischen Entwurf eines Systems zu zeigen. Sie repräsentieren dabei eine zeitliche Momentaufnahme der ansonsten dynamischen Konfiguration des Systems. Ein Objektdiagramm agiert deshalb als Prototyp und untersucht die Interaktionen und strukturellen Beziehungen einer Menge von Objekten unter einem bestimmten Szenario. Die beiden zentralen Elemente dieses Diagrammtyps sind die Objekte und deren Beziehungen.

2.2.4.1 Objekte und ihre Beziehungen



Die Objektikone zur Darstellung eines Objektes innerhalb der Objektdiagramme lehnt sich stark an diejenige der Klasse an, wobei die gestrichelte Linie mit einer ausgezogenen ersetzt worden ist. Einbeschrieben wird der Wolke zum einen der Name des Objektes, welcher in der Form

A	nur Objektname
: C	nur Objekttyp
A : C	Objektname und Objekttyp

anzugeben ist, und zum anderen eine optionale Untermenge der Objektattribute, welche in bereits für die Klassenattribute beschriebenen Form zu spezifizieren sind. Auch Instanzen von abstrakten Klassen oder Hilfsklassen können im Objektdiagramm enthalten sein; sie werden analog zu den entsprechenden Klassenikonen dargestellt.

messages

Verbindungen zwischen zwei Objekten werden als simple Linien dargestellt und sind als Instanzen der entsprechenden Assoziationen zwischen den instantiierten Klassen zu betrachten. Verbindungen werden also als Kommunikationspfade zweier Objekte angesehen, über welche Meldungen verschickt werden. Implizit besitzt jedes Objekt zu sich selber einen derartigen Kanal und kann sich demnach auch selber Nachrichten senden.

Eine Meldung besteht immer aus drei Elementen, nämlich aus:

- einem Synchronisationssymbol, welches die Richtung der Meldung angibt,
- einer Operation und
- optional einer Sequenznummer.

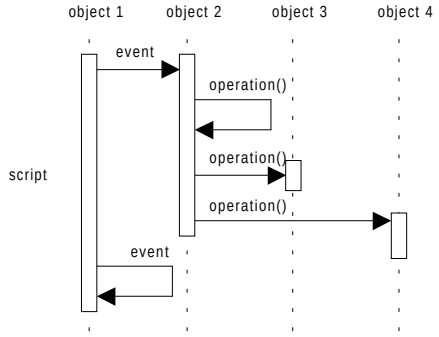
Dieser Meldungstyp zeigt die einfachste Form eines Meldungsaustausches und kann seine Semantik nur dann garantieren, wenn das System nur in einem einzigen Kontrollfluss abläuft. Arbeiten mehrere asynchrone Verarbeitungsprozesse konkurrenzierend zusammen, müssen weiterführende Konzepte [Booch 94, pp. 212-217] für die Synchronisation herangezogen werden.

2.2.5 Interaktionsdiagramme

Meldungsaustausch, Attributwerte, Rollen, Datenflüsse und Sichtbarkeit

Ein Interaktionsdiagramm wird für die Veranschaulichung der schrittweisen Abarbeitung eines gewissen Szenarios - meist dargestellt in einem Objektdiagramm - verwendet. In der Tat ist ein Interaktionsdiagramm bis zu einem gewissen Grad eine andere Repräsentation des Objektdiagramms, welche aber den Vorteil mit sich bringt, dass die relative Reihenfolge des Meldungsaustausches besser abgelesen werden kann. Zudem können im Interaktionsdiagramm zusätzlich Informationen über Verbindungen, Attributwerte, Rollen, Datenflüsse und Sichtbarkeit enthalten sein.

2.2.5.1 Objekte und Interaktion



Ein Interaktionsdiagramm wird in einer Art Tabelle dargestellt. Die im Zentrum des Interesses liegenden Objekte werde als Spaltenbeschriftungen angegeben. Interaktionen werden in Form von horizontalen Verbindungen der unterhalb der Objekte gezeichneten vertikalen gestrichelten Linien definiert. Dabei wird die Notation aus den Objektdiagrammen übernommen. Die

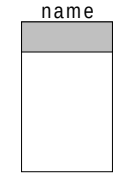
Interaktionssequenz wird durch die horizontale Position der entsprechenden Interaktionen definiert und kann deshalb direkt aus dem Diagramm gelesen werden.

2.2.6 Moduldiagramme

Ein Moduldiagramm wird verwendet, um die Allokation der Klassen und Objekte im physischen Entwurf eines Systems zu verdeutlichen. Ein einzelnes Moduldiagramm repräsentiert die Ansicht der Modulstruktur eines Systems, welche in der Implementierung die Aufteilung der Architektur in physische Bereiche und Schichten erlaubt.

2.2.6.1 Module und ihre Abhängigkeiten

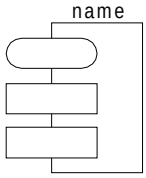
Module repräsentieren Dateien von drei verschiedenen Typen, die sich in ihrer Funktion unterscheiden:



main program

Hauptprogramm

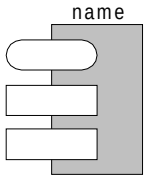
Das Hauptprogramm beinhaltet die Wurzel des Programmes (in C++ ist dies meist die ungebundene Funktion main). Typischerweise existiert in jedem Programm genau ein Hauptprogramm.



specification

Spezifikationen

Die Spezifikation enthält die Deklaration von Entitäten (in C++ sind dies meist Headerdateien mit der Extension *.h)



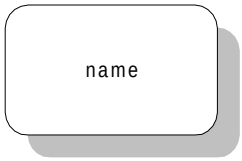
body

Rumpf

Der Rumpf enthält die Definition der deklarierten Entitäten (in C++ sind dies meist Implementationsdateien mit der Extension *.cpp)

Jedes Modul besitzt einen eindeutigen Namen, der idealerweise mit dem physischen Namen der Datei übereinstimmt. Die Extension wird dabei weggelassen, da sie aus dem Modultyp gelesen werden kann.
Sind zwei Module miteinander verbunden, so ist dies als eine Anweisung für den Compiler zu interpretieren. Zeigt nämlich ein Modul mit einem Pfeil auf ein anderes Modul, so besteht zwischen den beiden eine Abhängigkeit, was bedeutet, dass für die Kompilation des einen das andere bekannt sein muss (in C++ werden derartige Abhängigkeiten in include Direktiven umgesetzt).

2.2.6.2 Subsysteme

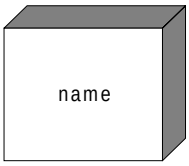


Ein grosses System kann mehrere Hundert, wenn nicht mehrere Tausend physische Module besitzen. Ein solches System zu überblicken ist ohne weitere Strukturierung nicht möglich. Analog zu den Klassenkategorien, welche Klassen innerhalb eines Bereiches oder einer bestimmten Schicht zusammenfassen, lassen sich Module in Subsysteme einordnen. Subsysteme beinhalten Module und/oder weitere Subsysteme. Dabei muss Modul in genau einem Subsystem enthalten oder global definiert sein. Die Namensgebung folgt keiner speziellen Regel, da Subsysteme nur Strukturierungshilfen sind und deshalb deren Namen nicht in physische Dateinamen umgesetzt werden.

2.2.7 Prozessdiagramme

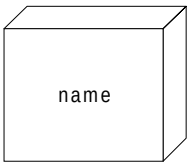
Prozessdiagramme werden verwendet, um die Zuordnung der Prozesse zu Prozessoren im physischen Entwurf zu zeigen. Ein einzelnes Prozessdiagramm repräsentiert eine Ansicht der Prozessstruktur eines Systems. Während der Entwicklung werden Prozessdiagramme verwendet, um die physische Sammlung der Prozessoren und Geräte, welche uns als Plattform für die Ausführung zur Verfügung stehen, zu identifizieren.

2.2.7.1 Prozessoren und Geräte



Prozessor

Ein Prozessor ist als Teil der Hardware, der ein Programm ausführen kann, zu betrachten. Dem Prozessor werden Prozesse zugeordnet, welche mit den Hauptprogrammen aus dem Moduldiagramm übereinstimmen.



Gerät

Ein Gerät hingegen wird als Hardwarekomponente aufgefasst, welche keine Programme ausführen kann, aber dennoch für die Verarbeitung notwendig ist.

Prozessoren werden mit Geräten verbunden, um eine Koppelung der beiden Komponenten zu verdeutlichen. Verbindungen können sich dabei sowohl auf dieselbe Maschine beziehen (Festplatte und Prozessor) oder als auch auf geographisch weit entfernte Rechnungseinheiten (Netzwerk).

2.2.8 Anwendung der Notation

Typischerweise wird ein System mit Hilfe einer Menge von Objektdiagrammen (für die Modellierung des Verhaltens eines Systems in diversen Szenarien), Klassendiagrammen (für die Modellierung der Rollen und Verantwortlichkeiten der beteiligten Entitäten) und Zustandsübergangsdiagrammen (für die Modellierung des ereignisgesteuerten Verhaltens der Entitäten) analysiert. Für den Entwurf werden zusätzlich Moduldiagramme und Prozessdiagramme miteinbezogen.

Verbindungen zwischen den Diagrammen

Zwischen den verschiedenen Diagrammen existieren Verbindungen. So ist es beispielsweise möglich, von der Implementation ausgehend die Anforderungen an das System zu identifizieren. Startet man beispielsweise von einem Prozessor, für welchen ein bestimmter Prozess zur Ausführung vorgesehen ist, findet man im Moduldiagramm ein entsprechendes Hauptprogramm, welches seinerseits verschiedene Module aufruft. Diese Module beinhalten Klassen bzw. Objekte, deren Eigenschaften aus dem Klassendiagramm bzw. Objektdiagramm ausgelesen werden können. Zum Schluss versinnbildlichen die Definitionen der Klassen die Anforderungen des Systems, da diese im Allgemeinen direkt die Objekte im Problembereich reflektieren.

Werkzeuge für die Sicherung der Komplettheit und der Konsistenz

Die Anwendung der beschriebenen Notation kann manuell erfolgen, jedoch kann der Entwurfsprozess in grösseren Systemen durch Werkzeuge für die Überprüfung der Komplettheit und der Konsistenz erheblich beschleunigt und sicherer gemacht werden. Zudem eignet sich die Methode sowohl für die Beschreibung von kleinen (ca. zehn Klassen) als auch von grossen (mehrere Tausend Klassen) Systemen. Die weitgehend erreichte Sprachunabhängigkeit fokussiert nicht zu früh auf eine spezifische Implementationssprache und lässt die Wahl der Zielsprache bis zum Abschluss des Entwurfes offen.

2.3 Das Vorgehen

Erfolgreiche Softwareprojekte zeichnen sich dadurch aus, dass sie zum einen einer starken architektonischen Vision folgen und zum anderen einen gut verwalteten iterativen und inkrementellen Entwicklungsprozess anwenden.

architektonische Vision

Die architektonische Vision ist deshalb von essentieller Wichtigkeit, da sie die Verständlichkeit, die Erweiterbarkeit, die Fähigkeit, ein System zu reorganisieren, testen und warten, unterstützt. Gute Architekturen zeichnen sich dadurch aus, dass sie

- aus mehreren wohldefinierten Schichten zusammengesetzt sind, wobei jede Schicht eine kohärente Abstraktionsstufe repräsentiert, durch eine kontrollierte Schnittstelle zugänglich ist und auf einer gleichsam wohldefinierten Schnittstelle der darunterliegenden Schicht aufsetzt.
- innerhalb jeder Schicht klar zwischen Schnittstelle und Implementation unterscheiden und so dem Entwickler ermöglicht, die Implementation zu ändern, ohne die Schnittstelle zu verletzen.
- einfach sind, d.h. einfaches Verhalten durch einfache Abstraktion umsetzen.

Objektorientierte Architekturen tendieren dazu, die genannten Eigenschaften guter Architekturen besser zu unterstützen als andere, was aber nicht zur Irrmeinung verleiten sollte, dass einzig die objektorientierten diesbezüglich gute Resultate liefern oder jeglicher objektorientierter Architekturentwurf gut ist.
Während des Architekturentwurfsprozesses müssen laufend Entscheidungen getroffen werden. Dabei gilt es zwischen strategischen und taktischen Entscheidungen zu unterscheiden.

Strategische Entscheidungen

Strategische Entscheidungen wirken auf die Gesamtarchitektur eines Systems und beeinflussen nicht selten sogar die übergeordneten Organisationsformen und -strukturen. Bespiele hierfür sind Mechanismen für Fehlerentdeckung und -behebung, Politik der Speicherverwaltung oder Schnittstellenphilosophie.

taktische Entscheidungen

Dahingegen haben taktische Entscheidungen meist nur lokale Implikationen und beeinflussen lediglich die Details von Implementationen, bzw. Schnittstellen. Die Änderung der Signatur einer Methode gehört beispielsweise in diese Kategorie von Entscheidungen.

iterativer und inkrementeller Entwicklungszyklus

Erfolgreiche Entwicklungsprojekte verfolgen häufig sowohl einen iterativen, als auch einen inkrementellen Entwicklungszyklus. Iterativ bedeute dabei, dass die objektorientierte Architektur sukzessive verfeinert wird, wobei die Erfahrungen aus der einen Iteration auf die nächste übertragen werden, während inkrementell bedeutet, dass jeder Durchlauf Analyse/Design/Evolution die strategischen und taktischen Entscheidungen unter der Berücksichtigung von zusätzlichen Anforderungen verfeinert, um am Ende die echten Bedürfnisse des Benutzers befriedigen zu können, und dennoch eine einfache und adaptierbare Lösung zu erzielen.

„top-down“ und „bottom-up“ Vorgehen

Der iterative und inkrementelle Entwicklungszyklus steht im Gegensatz zum traditionellen Wasserfallmodell und ist dementsprechend weder ein striktes „top-down“ noch „bottom-up“ Vorgehen. Vielmehr handelt es sich um ein „round-trip gestalt design“, welches die iterative und inkrementelle Entwicklung eines Systems als Ganzes - durch die Verfeinerung von verschiedenen noch konsistenten logischen und physischen Sichten - betont.

Rationales Entwurfsverfahren

Um den beschriebenen Entwicklungszyklus, auch Rationales Entwurfsverfahren genannt, erfolgreich umsetzen zu können, muss die betraute Softwareorganisation über einen gewissen Reifegrad [Humphrey 89, p. 5] verfügen.

2.3.1 Mikroentwicklungsprozess

Der Mikroentwicklungsprozess wird vor allem von den Szenarien und architektonischen Entscheidungen gesteuert, die vom Makroentwicklungsprozess stammen und dort auch schrittweise verfeinert werden. Er spielt sich im Mikrobereich des Systems ab, da er sich mit den täglichen Aktivitäten eines Entwicklers oder eines kleinen Entwicklungsteams beschäftigt. Von ihm sind gleichsam der Softwareingenieur, der die taktischen Entscheidungen zu treffen hat, und der Softwarearchitekt, der aus den Erfahrungen neue Alternativentwürfe ableitet, involviert.

Analyse und Design

Im Mikroprozess sind die traditionellen Phasen der Analyse und des Designs stark vermischt. Insbesondere kann kein Kochbuchrezept aufgezeigt werden, welches die Intelligenz und die Erfahrung eines versierten Entwerfers ersetzen kann.

Trotzdem tendieren die Vorgehensweisen darauf hin, die folgenden Aktivitäten zu unternehmen:

- Identifikation der Klassen und Objekte auf einer bestimmten Stufe der Abstraktion,
- Identifikation der Semantik dieser Klassen und Objekte,
- Identifikation der Beziehungen zwischen diesen Klassen und Objekten,
- Spezifikation der Schnittstellen zwischen diesen Klassen und Objekten und
- Implementation dieser Klassen und Objekte.

Für detailliertere Angaben zu den einzelnen Aktivitäten sei direkt auf [Booch 94, pp. 235-248] verwiesen.

2.3.2 Makroentwicklungsprozess

Risiko, Zielerreichung, Termin und Kosten

Der Makroentwicklungsprozess umspannt einem Rahmen gleich den Mikroentwicklungsprozess und kontrolliert ihn stets bezüglich Risiko, Zielerreichung, Termin und Kosten, um frühzeitig korrigierend auf ihn einwirken zu können. Die meisten dieser Aktivitäten, beispielsweise Konfigurationsmanagement, Qualitätssicherung, Dokumentation, usw., gleichen den üblichen Managementaufgaben und sind deshalb keineswegs auf objektorientierte Systeme allein zutreffend.

technisches Management

Der Makroprozess beschäftigt sich also mit dem technischen Management der Entwicklungsteams, indem er die Optik des Endbenutzers berücksichtigt, der weniger an technischen Details interessiert ist, als an Qualität, Vollständigkeit und Korrektheit. Er kümmert sich also um das Risiko und die architektonische Vision, welches die beiden Elemente sind, die den grössten Einfluss auf die genannten Kundenwünsche haben.

Der Ablauf ist im Gegensatz zum „kleinen“ Prozess streng geordnet und beinhaltet folgende Schritte:

- Erfassen der Grundanforderungen an die Software (Konzeptualisierung),
- Entwicklung eines Modells des vom System gewünschten Verhaltens (Analyse),
- Entwurf der Architektur für die Implementation (Design),
- Weiterentwicklung der Implementation durch sukzessives Verfeinern (Evolution) und
- Verwaltung der Weiterentwicklungen nach Auslieferung (Wartung).

Für die meisten Entwicklungsprozesse wiederholt sich dieser Prozess nach jeder Auslieferung einer Softwareversion. Die Philosophie dahinter ist, dass jede Folgeversion aus der Vorversion und einiger zusätzlicher Funktionalität bestehen soll.
Für detailliertere Angaben zu den einzelnen Schritten sei direkt auf [Booch 94, pp. 250-264] verwiesen.

3 Schwachstellenanalyse

3.1 Einleitung

Für die Identifikation der Schwachstellen wurde zuerst die Literatur herangezogen (vgl. beispielsweise [Heuer 93]). In den verschiedenen in das Studium einbezogenen Artikeln und Büchern wurden vor allem die Aspekte der langfristig ausgerichteten Datenbankanwendungen diskutiert.
Anschliessend mussten für die genauere Untersuchung einerseits eine bestimmte Methode und andererseits eine konkrete Datenbankimplementierung ausgewählt werden. Die Wahl musste so getroffen werden, dass konkrete Aussagen möglicht sind und dennoch keine allzu starken Einschränkungen der Allgemeinheit daraus resultieren.
Bei dem nachfolgenden Versuch, die Entwurfsmethode auf das Datenbanksystem abzubilden, wurde untersucht, welches die Unzulänglichkeiten sind, dies sowohl auf der konzeptuellen, als auch auf der logischen Ebene.
Erfahrungen wurden vor allem durch die Anwendung der Entwurfsmethode auf geeignete Fallstudien gewonnen. Untersucht wurden im Speziellen

- die „Institutsadministration“, die bereits für das Datenbanksystem O_2 implementiert und in [Geppert 96] diskutiert worden ist,
- die „Nationale Hockeyliga“, welche als Beispiel einer relationalen Datenbank im Rahmen eines Datenbankpraktikums an der Universität Zürich entworfen und implementiert worden ist,
- das „Hydroponic Gradening System“, welches als das leitende Beispiel in den Ausführungen von [Booch 94] gewählt und entsprechend (partiell) entworfen wurde und
- die „Konferenzadministration“, welche in verschiedenen Artikeln, insbesondere aber [Geppert 96], als Veranschaulichung herangezogen worden ist.

Anschliessend wurden die gefundenen Schwachpunkte mit den Erfahrungen von Personen, die wissenschaftlich und in der Praxis diesem Problem gegenübergestanden haben, abgeglichen, ergänzt und detailliert. Die Erfahrungsberichte wurden durch Interviews mit

- Andreas Geppert, stellvertretender Leiter des Instituts für Informatik, Autor des Buches [Geppert 96] und Dozent in verschiedenen Praktika der Veranstaltung „Objektorientierte Datenbanken“,
- Andreas Behm, Mitarbeiter in verschiedenen Projekten, welche sich wissenschaftlich mit den objektorientierten Datenbanken auseinandersetzen, und Assistent von diversen Veranstaltungen, die als Themenschwerpunkt die objektorientierten Datenbanken behandeln, und
- Thomas Wüst, ehemaliger Assistent am Institut für Informatik der Universität Zürich und gegenwärtiger Mitarbeiter der CSS-Versicherungen und dort betraut mit der Leitung von Entwicklungsprojekten, welche den Entwurf, die Implementation, den Betrieb und die Wartung von grossen ODBMS-basierten Informationssystemen zum Gegenstand haben,

ermittelt und in die Ausarbeitung aufgenommen.

3.1.1 Typisierung

Wenn von Schwachstellen die Rede ist, sollte dieser Begriff etwas näher beleuchtet werden:

Auf der einen Seite finden wir den konzeptuellen Entwurf und auf der anderen Seite die logischen (physischen) Datenschemata.
Schwachstellen können durch folgende Situationen auftreten:
Schwachstellen auf logischer Ebene
Der konzeptuelle Entwurf enthält ein Konstrukt, welches auf der Ebene der Datenbanken nicht unterstützt wird.
Schwachstellen auf konzeptueller Ebene
Der logische Entwurf enthält ein Konstrukt, welches auf der konzeptuellen Ebene nicht unterstützt wird.
Inkompatibilitäten
Sowohl der konzeptuelle als auch der logische Entwurf enthalten ein Konstrukt, welches aber in den beiden verschiedenen Welten eine unterschiedliche Semantik trägt.
Vermisste Konstrukte
Im Standard ODMG-93 sind Konstrukte definiert, die weder in den Methoden für den konzeptuellen Entwurf noch in den aktuellen Implementationen enthalten sind. Zudem sind von Datenbankadministratoren und Entwicklern von grossen Systemen aus der Praxis Konstrukte gefordert worden, die aus ihrer Sicht auf der einen Seite ihre Arbeit erleichtern und andererseits die Systeme stabiler und besser wartbar machen würden.
Dazu kommen noch sämtliche Schwachstellen, die von verschiedenen Autoren (vlg. dazu z.B. [Aksit et. al. 92]) aufgezeigt worden sind, die

- den Softwareentwurf,
- die Datenmodellierung, insbesondere den objektorientieren Entwurf,
- den Datenbankentwurf

im Allgemeinen betreffen. Diese Probleme werden in der vorliegenden Ausarbeitung ausser Betracht gelassen. Sie sollen Gegenstand weiterer Forschungen im Bereich des Softwareentwurfes sein.
Die Schwachstellen auf der logischen Ebene haben ihre Ursache in Unzulänglichkeiten seitens des untersuchten Datenbanksystems und werden im entsprechenden Abschnitt nur summarisch wiedergegeben.
Inkompatibilitäten zwischen der gewählten Entwurfsmethode und der Datenbankimplementation wurden keine gefunden. Vielmehr wurden Konstrukte, bei denen ein gewisses Verdachtsmoment bestanden hat (z.B. inverse Beziehungen), aufgrund der starken Differenz den vermissten Konstrukten zugeteilt.
Die konzeptuellen Schwachstellen und vermissten Konstrukte hingegen sind im folgenden detailliert diskutiert. Zuerst ist die Problematik unter Angabe gewisser begrifflicher Definitionen im Allgemeinen umrissen, anschliessend die Konsequenzen für den konzeptuellen Entwurf aufgezeigt und schliesslich die Schwachstelle für die weitere Verwendung in den folgenden Kapiteln zusammengefasst.
In der Evaluation schliesslich werden die gefundenen Schwachstellen in bezug auf die Bedürfnisse des konzeptuellen Entwurfes gewichtet und eine Auswahl für die weitere Bearbeitung getroffen.

3.2 Schwachstellen auf logischer Ebene

3.2.1 Aggregation

Es gibt keine Aggregation, wie sie im objektorientierten Paradigma gefordert ist. Durch die Definition von komplexen Attributen in Form von Strukturen (Tuples) kann ein ähnliches Verhalten erzwungen werden. Wird ein Attribut vom Typ Klasse A einer Klasse B zugeordnet, wird dies lediglich als Zeiger auf die Klasse A interpretiert.

3.2.2 Überladung

Es gibt keine Überladung. Polymorphismus ist nur durch Überschreiben und spätes Binden implementiert. Überschreibung kann nur durch Spezialisierung gemacht werden.

3.2.3 Konstruktoren und Destruktoren

In O_2 gibt es keine Konstruktoren oder Destruktoren. Der Konstruktor wird mit der Implementierung eine Init()-Methode simuliert. Diese kennt aber keine Überladung. Der Destruktor kennt keine Entsprechung in Form einer DeInit()-Methode oder etwas Ähnlichem.

3.2.4 Weitere Schwachstellen

Das betrachtete Datenbanksystem unterstützt nicht alle in der Entwurfsmethode eingeführten Klassentypen. So entbehren beispielsweise abstrakten Klassen, Freundschaftsklassen (friend) oder eingebettete Klassen (nesting) einer Entsprechung auf der logischen Ebene.

3.3 Schwachstellen auf konzeptueller Ebene

3.3.1 Persistenz

Ein Objekt wird persistent genannt, wenn es auch über die Lebenszeit des Programmes, in dem es erzeugt wurde, hinaus gespeichert bleibt. Persistenz ist demnach die zentrale Forderung an ein OODBMS. In den verschiedenen Datenbanksystemen ist aber die Persistenz unterschiedlich implementiert.

Persistenz, Transienz und Persistenzfort-pflanzung

Um Missverständnissen vorzubeugen, ist es aber sicherlich zuerst sinnvoll, die Begriffe Persistenz und Transienz zu definieren. Zudem wird die Persistenzfortpflanzung für das Datenbanksystem O_2 erläutert.

Persistenz

Können Instanzen einer Klasse persistent sein, so heisst diese Klasse **persistenzfähig**. Der Einfachheit halber wird aber häufig nur der Begriff persistent verwendet. Persistente Klassen müssen also nicht ausschliesslich persistente Instanzen erzeugen.

Transienz

Im Gegensatz dazu wird der Begriff der Transienz für Klassen verwendet, deren Instanzen nicht persistent gemacht werden dürfen.

Persistenzfortpflanzung

Innerhalb der Datenbankumgebung O_2 gilt der Grundsatz der „Persistenz durch Erreichbarkeit“ [Geppert 96, p. 36]. Im übertragenen Sinne heisst das, dass sich die Persistenz von Instanzen längs der Referenzen fortpflanzt. Enthält also die Klasse A eine

Referenz auf die Klasse B, so wird die Persistenz, sofern die Instanz A_n der Klasse A bereits persistent ist, auf die Instanz B_m übertragen.

3.3.1.1 Einstiegspunkt

persistente Namen

Diese Art der Fortpflanzung verlangt für eine Schemadefinition mindestens einen persistenten Einstiegspunkt. Dieser muss eine Instanz einer persistenzfähigen Klasse sein und kann demnach als globales Objekt betrachtet werden. Die globale Instanz wird über den sogenannten persistenten Namen identifiziert.

Wurzel des Persistenzgraphen

Ist eine solcher Einstiegspunkt einmal definiert, werden sämtliche von ihr referenzierten Instanzen gleichfalls persistent. Der Einstiegspunkt kann also bezüglich der Persistenzfortpflanzung als Wurzel des Persistenzgraphen betrachtet werden.

3.3.1.2 Schwachstelle

Die Definition von persistenten Einstiegspunkten wird in der Schemadefinition, also mit ODL, erledigt. Aus diesem Grund muss die Persistenzdeklaration auch in der Entwurfssprache enthalten sein. Ein derartiges Konstrukt steht jedoch nicht zur Verfügung.

Zudem fehlen in der Entwurfsnotation Konstrukte, welche es dem Entwerfer erlauben, die Kontrolle über die Persistenzfortpflanzung zu übernehmen. Bei der betrachteten Entwurfssprache kann nicht zwischen den Assoziationen mit oder ohne Einfluss auf die Persistenz unterschieden werden.

3.3.2 Applikationsklassen und -instanzen

Hand in Hand mit der Unterscheidung zwischen persistenten und transienten Klassen geht die Unterscheidung zwischen Datenklassen, welche über die Lebenszeit der Applikation hinaus erhalten bleiben sollen, und Applikationsklassen, welche für die Modellierung der Anwendung nötig sind.

Bei dieser Abgrenzung müssen für den Entwurf die Problembereiche der Persistenz, der Schnittstellendefinition und der Plazierung der Methode genauer betrachtet werden.

3.3.2.1 Persistenz

In gewissen Datenbanksystemen sind die Applikationen selber nicht als Klassen implementiert. Trotzdem können sie während ihrer Lebenszeit Objekte, welche gewisse Funktionalitäten kapseln oder auch nur zur temporären Speicherung von Daten dienen, allozieren. Diese Objekte aber müssen nach der Terminierung der Anwendung allesamt zerstört worden sein und dürfen sich somit nicht mehr in der Datenbasis befinden.

Die Entwurfsmethode sollte also dem Entwerfer erlauben, diesen Umstand bereits während des Designs zu berücksichtigen.

3.3.2.2 Definition der Schnittstellen

Booch kennt in seiner Entwurfsmethode das Konstrukt der Modul-Diagramme. In diesen Diagrammen werden die physischen Module voneinander abgegrenzt. Zusätzlich können Subsysteme definiert werden.

Diese Methode kennt aber die Schwachstelle, dass Schnittstellen zwischen den einzelnen Modulen nicht genauer spezifiziert werden können. Vielmehr ist der Zusammenhalt, das Zusammenspiel der Elemente völlig offen gelassen.
Würde man beispielsweise einerseits diejenigen Klassen, die das logische Datenbankschema modellieren, in ein Modul (Subsystem) verpacken und andererseits diejenigen, welche die Applikation versinnbildlichen, in ein anderes Modul einbetten, so wäre das Entscheidende und Interessante die Definition der Schnittstellen zwischen den beiden Subsystemen.
Im weiteren müsste auch die Art der Zusammenarbeit durch die Spezifikation der Verantwortlichkeiten geregelt sein.
Das Modul der logischen Datenschemata wird zudem idealerweise persistent gehalten, während die Module der Applikationen transient implementiert sein sollen.

3.3.2.3 Plazierung der Methoden

Wenn die Verantwortlichkeiten der Subsysteme (Module) geregelt sein soll, muss ein Vorgehensschema definiert werden können, welches unter folgenden Gesichtspunkten optimiert ist:

- Der Grundsatz der Datenkapselung soll nicht aufgeweicht werden, d.h. Daten sollten nur durch wohldefinierte, öffentliche Methoden zugänglich sein.
- Das Verhalten einer Klasse soll auf der richtigen Ebene modelliert sein, d.h. die Anforderung der Wiederverwendbarkeit von Komponenten sollte nicht durch zu hohe Spezialisierung von Klassen eingeschränkt sein.
- Transaktionen sollen von der Verwenderklasse implementiert werden, insbesondere sollen keine Methoden von Klassen innerhalb des logischen Schemas Transaktionen beinhalten.

3.3.2.4 Beispiel „Bibliothek“

Sei das Modul des logischen Schemas mit den Klassen „Bibliothek“, „Student“ und „Buch“ und die Applikation mit der Klasse „BibliotheksManager“, welche eine Benutzerschnittstelle zur Verfügung stellt, ausgestattet.
Wie werden neue Bücher in die Bibliothek aufgenommen? Wird dies von der Applikation oder von der Bibliotheks-Klasse erledigt?
Sowohl als auch. Die Applikation muss dem Benutzer eine Methode zur Verfügung stellen und ruft in dessen Implementierung die entsprechende Methode der Bibliotheksklasse auf. Daraus folgt, dass eine Funktion, sie heisse AddStudent(), in die Schnittstelle aufgenommen werden muss.
Wie wird eine Liste aller ausgeliehener Bücher aufbereitet? Wie wird diese visualisiert?
Die Aufbereitung der Liste ist von einer Sichten-Klasse innerhalb des Applikationsmoduls zu erledigen. Der Zugriffskanal zur Beschaffung der Daten muss in der Schnittstelle definiert sein. Die Visualisierung

muss ebenfalls von der Applikation geregelt werden, ansonsten wäre die Wiederverwendbarkeit bereits nur noch für gleichartige GUIs gesichert.
Wie wird die (Trans-) Aktion des Buchausleihens implementiert?
Der Auf- bzw. Abbau der Referenzen der Objekte innerhalb der persistenten Daten muss von den Methoden der Datenbankklassen eigenständig geregelt werden können. Jedoch muss dem Benutzer ein Interface zur Verfügung stehen, in welchem er den Studenten und das Buch auswählen und dann die Transaktion mit diesen beiden Referenzen gegenüber der Datenbank absetzen kann.

3.3.2.5 Schwachstelle

In der Entwurfsnotation fehlen die Konstrukte, welche es erlauben, zwischen transienten und persistenzfähigen Klassen zu unterscheiden. Zudem ist das Zusammenspiel der Applikationsklassen mit den Klassen des Schemas durch die mangelhafte Fähigkeit der Schnittstellenspezifikation nicht geregelt. Die Entwurfssprache muss aber derartige Möglichkeiten bieten, da bei der betrachteten Datenbankumgebung die Applikationen Teil des Schemas sind und deshalb in den Entwurf mit einbezogen werden müssen.

3.3.3 Transaktionen

Der Entwurf hat die Abbildung der Sachverhalte des Problembereiches auf ein semantisch äquivalentes Datenschema als zentrales Ziel. Genauso wichtig aber ist die Gewährleistung der Datenintegrität während der Manipulation der beteiligten Objekte.

3.3.3.1 Datenintegrität

Die Datenintegrität muss unter zwei verschiedenen Gesichtspunkten beleuchtet werden: der operationalen und der semantischen.

physische Konsistenz
Concurrency Control
Recovery

Die operationale Datenintegrität bezieht sich auf die physische Konsistenz der Daten. Sie könnte beispielsweise durch den Mehrbenutzerbetrieb verletzt werden. Dies ist aber in Datenbanksystemen, welche ein Transaktionskonzept implementiert haben, nicht möglich. Sie beinhalten bereits Mechanismen für die Kontrolle konkurrenzierender Zugriffe (Concurrency Control) und Wiederanlauf im Fehlerfall (Recovery). Insbesondere kann also die operationale Datenintegrität nicht durch den Entwurf, die Umsetzung oder durch den Betrieb gefährdet werden.

Transaktion als Einheit der Konsistenz

Anders verhält es sich bei der semantischen Datenintegrität, welche verletzt werden kann. Das Datenbanksystem bietet zwar Transaktionen, welche als Einheit der Konsistenz, an deren Beginn und Ende die Datenintegrität gilt, zu betrachten sind, im konzeptuellen Entwurf müssen aber dennoch Sachverhalte der modellierten Systemwelt auf Transaktionen abgebildet werden. Aus diesem Grund müssen dem Entwerfer entsprechende Konstrukte und Vorgehensrichtlinien zur Verfügung stehen.

3.3.3.2 Objektmanipulationen

Create, Read, Update, Delete

Ein Objekt unterliegt stets dem Zyklus CRUD (Create, Read, Update, Delete). Daraus lassen sich die verschiedenen Modifikationen festlegen.
Erzeugung (Create)
Kann als Schreibzugriff gesehen werden, der aber

erschwerenderweise noch weitere Schreibzugriffe verursachen kann: man denke an die Instanzierung von Aggregationsklassen.
Lesen (Read)
Reiner Lesezugriff. Die Datenbankumgebung muss für die Aktualität der Daten im Mehrbenutzerbetrieb sorgen.
Änderung der Werte (Update)
Die Aktualisierung ist ein Schreibzugriff. Man beachte, dass in den meisten Fällen auch die Erzeugung oder die Löschung anderer Objekte zu einer Aktualisierung eines betroffenen Objektes führen können (Änderung der Referenzattribute).
Löschung (Delete)
Kann auch als Schreibzugriff gesehen werden. Es müssen - wie beim Create - Fortpflanzungseffekte berücksichtigt werden.
In Mehrbenutzersystemen, wie sie in DBMS vorliegen, müssen die Zugriffsrechte geregelt sein. Die Datenintegrität muss geschützt werden können und darf insbesondere nicht durch den Mehrbenutzerbetrieb gefährdet werden. Diesem Konzept muss bereits in der Entwurfsphase Beachtung geschenkt werden können.

3.3.3.3 Schutzmechanismen

öffentliche, geschützte und private Attribute und Methoden

Dank des Grundsatzes der Datenkapselung ist in den bekannten objektorientierten Sprachen und der entsprechenden Entwurfmethoden bereits ein Schutzmechanismus implementiert. Der Zugriff kann durch die Typisierung der Attribute und Methoden in öffentliche, geschützte und private geregelt werden.

Benutzerprofil

Dieses Konzept ist aber nur begrenzt für Datenbankanwendungen genügend, da im konzeptuellen Entwurf die Möglichkeit bestehen sollte, die Zugriffsrechte in Abhängigkeit des Profils des aktuell angemeldeten Benutzers zuzuteilen.

3.3.3.4 Schwachstelle

Transaktionen bilden innerhalb von Datenbanksystemen die Einheit der Konsistenz. Die betrachtete Entwurfsmethode verfügt aber weder über entsprechende Konstrukte zur Definition von Transaktionsaufrufen noch über Vorgehensrichtlinien für die Formulierung von Konsistenzbedingungen zur Sicherung der semantischen Datenintegrität. Zudem sollte für den konzeptuellen Entwurf die Zugriffstypen (schreibend, lesend) für die öffentlichen Methoden mit berücksichtigt werden können.

3.3.4 Weitere Schwachstellen

3.3.4.1 Physische Aspekte

Für den physischen Datenbankentwurf sollten die Aspekte des **Clustering**, der **Indexierung**, der **Distribution** und der **Fragmentierung** berücksichtigt werden können. Die betrachtete Entwurfssprache bietet zwar für den physischen Entwurf die Modul- und die Prozessdiagramme an, die aber den für die Datenbanken erhöhten Anforderungen nicht gerecht werden können.

3.3.4.2 Vererbungshierarchien

In O_2 müssen alle Klassen von `Object` abgeleitet sein. Zudem können für die Implementation der gewöhnlichen Eigenschaften einer Klasse weitere abstrakte Klassen (z.B. `DBObject`) eingeführt werden. Diese impliziten Vererbungsbeziehungen sollten aber die Lesbarkeit des Entwurfsdiagramms nicht belasten.

3.4 Vermisste Konstrukte

3.4.1 Inverse Beziehungen

In der Methode von Booch ist kein explizites Konstrukt für den Entwurf von inversen Beziehungen vorgesehen. Zwar erlaubt er, dass bei der Verwendung einer beliebigen Beziehung an beiden Enden der Verbindung Kardinalitäten angegeben werden. Jedoch unterliegen diese keiner explizit definierten Semantik.
Dieser Missstand sei am Beispiel eines sehr einfachen Informationssystems für eine Studentenbibliothek veranschaulicht:

3.4.1.1 Beispiel „Bibliothek“

Für das Informationssystem existieren folgende Vorgaben:

- Studenten sind Mitglieder in einer Bibliothek
- Bücher gehören der Bibliothek
- Ein Student kann 0 - N Bücher ausleihen
- Ein Buch kann entweder von genau einem Studenten oder gar nicht ausgeliehen sein

Für die Modellierung dieses Systems seien drei Klassen („Bibliothek“, „Student“ und „Buch“) vorgeschlagen. Zwischen je zwei von diesen Klassen muss eine Beziehung eingerichtet werden, wobei diejenige zwischen dem Studenten und dem Buch invers sein soll. Dies bringt den Vorteil, dass auf direkte Weise (d.h. nicht über andere Klassenextensionen), sowohl der Name des Studenten, der ein bestimmtes Buch ausgeliehen hat, als auch die Liste aller Bücher, die von einem bestimmten Studenten ausgeliehen worden sind, ausgedruckt werden kann.

3.4.1.2 Beziehung „by reference“

Modelliert man diese Beziehung innerhalb der Booch’schen Methode als „has-a by reference“-Verbindung, die auf der Seite des Studenten die Kardinalitätsangabe 1 trägt, und auf der Seite des Buches eine vom Typ „0+“, ergeben sich einige Unstimmigkeiten.
Dem Entwickler öffnen sich nun beim Interpretieren des Klassendiagramms Ermessensspielräume, was sicherlich nicht im Interesse des Entwerfers ist. Verdeutlicht sei dies an den folgenden Situationen, die das Klassendiagramm nicht verletzen aber semantisch wohl nicht beabsichtigt waren:

- Student A leiht Buch 1, Buch 1 wird ausgeliehen von Student B (Gesamtzahl der Beziehungen korrekt, aber nicht invers)
- Student A leiht Buch 1, Buch 1 wird von niemandem ausgeliehen (Beziehung unvollständig)

- Student A leiht Buch 1, Buch 1 wird ausgeliehen von Student 2 und Student B leiht Buch 2, Buch 2 wird ausgeliehen von Student A (Gesamtzahl der Beziehungen korrekt, Beziehungen vollständig, aber nicht invers)

Die Beziehung ist also auf diese Weise lose verbunden, d.h. der Entwerfer muss durch die Angabe von informellen Notizen genauer spezifizieren, wie die gezeichnete Beziehung zu interpretieren ist. Auf der anderen Seite muss der Programmierer in Situationen, in denen eine inverse Beziehung gefordert ist, selber dafür sorgen, dass die Inversität nicht verletzt wird.
Die muss er dadurch erreichen, dass er überprüft, ob folgende Kriterien zu erfüllen sind:

1. Kardinalitäten erfüllt,
2. Gesamtzahl der Beziehungen korrekt,
3. Beziehungen vollständig und
4. Beziehungen invers.

Zudem ist der Auf- und Abbau der Beziehung nicht geregelt. Syntaktisch können beide Teilnehmer gleichberechtigt Beziehungen auf- und abbauen, was semantisch in gewissen Situationen (das Buch wird von einem Studenten ausgeliehen) zuweilen fragwürdig ist.
Angenommen, die Ausleihe-Beziehung soll von der Klasse „Student“ aus aufgebaut werden, muss innerhalb dieser Methode das entsprechende inverse Gegenstück dazu gleichfalls gesetzt werden. Somit muss die Klasse „Buch“ entweder einen öffentlichen Zeiger auf die Klasse „Student“ oder eine öffentliche Beziehungsaufbau-Methode zur Verfügung stellen. In jedem Fall aber steht dies im Widerspruch zu der Forderung, dass die Beziehung nur einseitig aufgebaut werden darf.
Durch die Deklaration einer „friend“-Klasse könnte diese Klippe einigermassen elegant umschifft werden. Dieses Konstrukt steht aber in den betrachteten objektorientierten Datenbanksystemen nicht zur Verfügung, was die Forderung der Objektorientierung nach maximaler Kapselung entschieden aufweicht.
Soll die Beziehung von beiden Seiten her aufgebaut werden können, muss eine an einer derartigen Verbindung teilnehmende Klasse zwei unterschiedliche Methoden - eine für den aktiven und eine für den passiven Aufbau - anbieten. Dabei sollten aber gleichsam die passiven Aufbaufunktionen vor unerlaubten Zugriffen geschützt werden können, was mangels Mächtigkeit der Datendefinitionssprache - wie oben erläutert - nicht realisierbar ist.

3.4.1.3 Beziehungen „by value“

Modelliert man die Ausleih-Beziehung innerhalb der Booch’schen Methode als „has-a by value“-Verbindung und setzt man auf der Seite des Besitzers eine Kardinalitätsangabe, die weder „1“ noch „C“ ist (was von der Konstruktsyntax her erlaubt ist), verliert das Konstrukt damit semantisch jeglichen

Sinn. Eine Komponente *kann nicht* in mehreren Aggregaten enthalten sein. Zudem muss aus naheliegenden Gründen die rückbezügliche Beziehung vom Typ „by reference“ sein.
Durch Beziehungen „by value“ werden Existenzabhängigkeiten beschrieben. Dies bedeutet für den Beziehungsaufbau, dass innerhalb dieser Methode das Objekt, zu dem eine derartige Beziehung aufgenommen werden soll, erst noch kreiert werden muss. Die Funktion muss demnach als Parameter sämtliche für die Konstruktion eines Objektes notwendige Information in der Schnittstelle mit berücksichtigen.
Werden für eine Aggregationsklasse gar zwingende Komponenten gefordert (ein Auto muss einen Motor haben), entstehen sowohl für die Konstruktoren als auch für die Verbindungsaufbaufunktionen komplizierte Fortpflanzungseffekte. Diese sind zwar durch die Formulierung von inhärenten Konsistenzbedingungen (keine Zyklen in derartigen Verbindungsgraphen) realisierbar, verlangen aber dem nicht durch eine seriöse Methodik unterstützten Entwickler einiges an Arbeit ab.
Beinhaltet der Entwurf dazu noch zwingende Beziehungen vom Typ „by reference“, lässt sich die Konsistenz meist nur noch durch geschickte Zusammenfassung von mehreren Aktionen in derselben Transaktion sichern.

3.4.1.4 Schwachstelle

Im Standard der ODMG-93 werden inverse Beziehungen gefordert. In der betrachteten Implementation fehlt die Fähigkeit zur Formulierung derartiger Beziehungen. Darüber hinaus sind in der Methode von Booch inverse Beziehungen nicht spezifizierbar. Rückbezügliche Beziehungen sind aber für beispielsweise die Löschfortplanzung von eminenter Wichtigkeit (Löschung von referenzierten Objekten).

3.4.2 Extensionen und Schlüssel

Objektidentität versus (Werte-) Schlüssel

Ein Objekt wird in einem ODBMS über seine Objektidentität (OID) eindeutig von anderen Objekten unterscheidbar gemacht. In Zusammenhang mit Datenbankanwendungen wäre aber eine andere Art der Identifikation wünschenswert, nämlich über einen (Werte-) Schlüssel. Insbesondere zum Auffinden von gewissen Instanzen in Objektmengen sollte, basierend auf einer Wertemenge, abgefragt werden können.
Zudem ist in der ODL, wie sie von der ODMG-93 vorgeschlagen wurde, der Schlüssel Bestandteil der Objektschnittstellendefinition.
Aus der Definition der Schlüsselbeziehungen lassen sich Konsistenzbedingungen ableiten. Die automatische Überprüfung dieser zusätzlichen Sicherheiten wäre wünschenswert.

Fremdschlüssel-beziehungen

Die Fremdschlüsselbeziehungen, wie sie aus dem Umgang mit relationalen Systemen bekannt sind, werden in OODBMS bekanntlich über Beziehungen zwischen Objekten modelliert. Eine Überprüfung der Kardinalitäten sowie die Validierung einer Beziehung ist elementare Voraussetzung für den Schutz der Datenintegrität.

3.4.2.1 Schwachstelle

In konkreten ODBMS Implementierungen, wie z.B. O_2, ist aber die Schlüsseldefinition kein Bestandteil der O_2C-ODL. Auch in der betrachteten Entwurfssprache ist dieses Konstrukt nicht vorgesehen. Zudem fehlen die Konzepte und Konstrukte für die Beschreibung von Extensionen, welche für das Auffinden von bestimmten Instanzen, für die Überprüfung von Eindeutigkeitskriterien und das Formulieren von summarischen Abfragen von zentraler Bedeutung sind.

3.4.3 Fortpflanzung und Konsistenzsicherung

Eine zentrale Aufgabe eines Datenbanksystems ist der Schutz der operationalen Datenintegrität. Darüber hinaus soll das DBS auch Konstrukte und Mechanismen zur Verfügung stellen, mit Hilfe derer die semantische Konsistenz der Daten definiert und überwacht werden können. Als Konsequenz müssen im logischen Entwurf gleichsam Hilfsmitte angeboten werden, um die Sicherung der Integrität und Fortpflanzungseffekte modellieren zu können. Nachfolgend sollen einige weitere Erläuterungen zu der Propagation und Konsistenzsicherung ausgeführt sein, um schliesslich die Schwachstelle des logischen Entwurfes aufzuzeigen.

3.4.3.1 Fortpflanzung

Exklusivität und Existenzabhängigkeit

Unter einem Fortpflanzungseffekt versteht man die Eigenschaft einer Datenbank, automatisch die Konsistenz für existentielle Beziehungen aufrechtzuerhalten. Beispielsweise ist dieser Effekt bei der Löschung von Objekten, die als Aggregat von untergeordneten (enthaltenen) Objekten auftreten, erwünscht. Dabei sind aber die verschiedenen Arten von Aggregation (Exklusivität und Existenzabhängigkeit) zu unterscheiden.

In den folgenden Situationen ist die Fortpflanzung der Erzeugung bzw. Löschung semantisch sinnvoll:

exklusiv, existenzabhängig

Ein Auto enthält genau einen Motor (exklusive, existenzabhängige Aggreagtion).

- Erzeugung des Aggregates: Wenn das Auto erzeugt wird, soll auch sein Motor erzeugt werden.
- Erzeugung der Komponente: Wenn ein Motor erzeugt wird, muss auch ein Auto erzeugt werden.
- Löschung des Aggregates: Wenn das Auto gelöscht wird, soll auch sein Motor gelöscht werden.
- Löschung der Komponente: Wenn der Motor gelöscht wird, muss auch das Auto gelöscht werden.

nicht-exklusiv, existenzabhängig

In einer Autowerkstatt werden Fahrzeugtypen und Motortypen gewartet. Einem Auto ist immer genau ein Motor zugeordnet (nicht-exklusive, existenzabhängige Aggregation).

- Erzeugung des Aggregates: Wird ein neuer Fahrzeugtyp aufgenommen, muss ihm entweder ein bestehender Motor zugeordnet werden oder es muss ein neuer Motortyp erzeugt werden.
- Erzeugung der Komponente: Wird ein neuer Motortyp erzeugt, muss dieser einem bestehenden Auto zugeordnet

werden (Austausch des Motors kann ev. eine Löschung verursachen) oder es muss ein neues Auto erzeugt werden.

- Löschung des Aggregates: Wird ein Auto nicht mehr gewartet, muss auch die entsprechende Aggregationsbeziehung abgebaut werden. Wird der Motor nun von keinem anderen Auto mehr verwendet, wird er gleichfalls gelöscht.
- Löschung der Komponente: Wird ein Motor gelöscht, werden sämtliche (einmotorigen) Autos, welche diesen Motortyp eingebaut haben, gleichfalls gelöscht.

In den folgenden Situationen ist von der Fortpflanzung der Erzeugung bzw. Löschung abzusehen:

exklusiv, nicht-existenzabhängig

Jeder Mitarbeiter hat seinen Arbeitsplatz in einem Einzelbüro (exklusive, nicht existenzabhängige Aggregation).

- Erzeugung des Aggregates: Wenn ein neuer Mitarbeiter angestellt wird, soll nicht ein neues Büro gebaut werden. Vielmehr wird diesem Mitarbeiter ein leeres Büro zugeordnet.
- Erzeugung der Komponente: Werden neue Büroräumlichkeiten hinzugemietet, müssen nicht zwingend neue Mitarbeiter angestellt werden.
- Löschung des Aggregates: Tritt ein Mitarbeiter aus der Firma aus, hat dies im Allgemeinen keinen Einfluss auf die Menge der Büros.
- Löschung der Komponente: Aus der Auflösung von Mietverträgen resultieren selten Entlassungen. Vielmehr werden lediglich die überflüssigen (leeren) Büroräume freigegeben.

nicht-exklusiv, nicht-existenzabhängig

Jeder Mitarbeiter arbeitet in einem oder mehreren Teams (nicht exklusive, nicht existenzabhängige Aggregation).

- Erzeugung des Aggregates: Wird eine neues Team ins Leben gerufen, müssen nicht zwingenderweise neue Mitarbeiter eingestellt werden. Vielmehr werden Teams aus bestehenden Mitarbeitern gebildet.
- Erzeugung der Komponente: Wird ein neuer Mitarbeiter eingestellt, wird dieser in ein oder mehrere der bestehenden Teams integriert.
- Löschung des Aggregates: Bei Projektende wird das Projektteam aufgelöst. Die Mitarbeiter werden aber nicht entlassen, sondern in neue Teams eingeteilt.
- Löschung der Komponente: Tritt ein Mitarbeiter aus der Firma aus, wird das Team nicht aufgelöst, selbst dann nicht, wenn es nun keine Mitglieder mehr zählt. Die Existenz eines Teams ist eher an eine Aufgabe gebunden, das heisst es werden ihm neue Mitarbeiter zugeteilt.

Aus diesen konstruierten Situationen kann die Tendenz abgelesen werden, dass dort, wo die Fortpflanzung erwünscht ist, mit existenzabhängigen Aggregationsbeziehungen gearbeitet werden soll, während in den anderen Fällen die

Modellierung der Beziehung durch Referenzen vorzuziehen ist.
Da die aktuellen Implementationen von objektorientierten Datenbanksystemen kein Sprachkonstrukt für die Definition von Aggregaten mit Existenzabhängigkeit vorsehen, muss dies durch andere Strategien nachgebildet werden.

Fortpflanzungsrichtung

Die Erzeugung, bzw. Löschung einer Instanz kann sich in zwei „Richtungen" fortpflanzen:

- Vom Aggregat zu den Komponenten: Ein Auto soll gelöscht werden. Dies impliziert die Löschung des Motors, der Zylinder des Motors, des Getriebes, der Sitze, usw.
- Von einer Komponente zu einem Aggregat: Ein Motor soll gelöscht werden. Dies impliziert zum einen die Löschung der Zylinder aber zum anderen auch, falls die Bedingung formuliert wurde, dass ein Auto zwingenderweise einen Motor haben muss, die Löschung des Autos, und dementsprechend des Getriebes, der Sitze, usw.

Auf der logischen Ebene kann nun die Affektierung der Aggregate durch die Löschung ihrer Komponenten auf zwei Arten umgesetzt werden: entweder man verhindert in solchen Fällen die Löschung von existenziellen Komponenten oder man leitet den Löschbefehl direkt zum Aggregat weiter.
Einen weiteren Ausweg aus dieser Situation wäre durch die Formulierung geeigneter Transaktionen zu finden.

3.4.3.2 Konsistenzsicherung

Ein grosser Problembereich bei der Arbeit mit langlebigen Daten ist die Sicherung der Konsistenz des Datenschemas und der aktuell existenten Objekte. Es sollen in der Folge nur die persistenzfähigen Klassen betrachtet werden, da nur deren Instanzen die Lebenszeit der Applikationen überdauern. Transiente Objekten hingegen gehorchen einer minim anderen Philosophie, sie können nämlich auch ausserhalb von Transaktionen zum Leben erweckt und zerstört werden.

Initialzustand

Diese Problematik lässt sich grundsätzlich in zwei verschiedene Teilprobleme unterteilen. Zum einen muss beim Erstellen der allerersten Instanz die Konsistenz erreicht werden (Initialzustand) und zum anderen muss die Konsistenz bei allen möglichen Manipulationen erhalten bleiben.

Manipulationen

Die möglichen Manipulationen lassen sich in drei verschiedene Aktionen unterteilen.

- Die Werte der Objektattribute werden verändert. Dies passiert nicht nur bei der reinen wertemässigen Änderung der Attribute, sondern auch bei dem Auf- bzw. Abbau von Beziehungen zwischen Objekten.
- Objekte können kreiert werden und durch Anbindung an ein bereits persistent gespeichertes Objekt in die Datenbasis gelangen.
- Objekte können durch einen Löschvorgang aus der Datenbasis entfernt werden.

Lesezugriffe

Der Lesezugriff ist nicht als Manipulation aufzufassen, selbst wenn für die Zwischenspeicherung der Resultate von beispielsweise Abfragen Objekte erzeugt werden. Derartige Objekte leben nur für eine kurze Zeit und sollen insbesondere nicht in die Datenbasis gelangen.

Es muss also gewährleistet sein, dass sowohl vor als auch nach einer Transaktion, die als eine einzelne oder eine Sequenz von mehreren Manipulationen aufgefasst werden soll, die Konsistenzbedingungen erfüllt sind.

3.4.3.3 Konsistenzbedingungen

Ursprung und Gültigkeitsbereich

Die Konsistenzkriterien werden durch die Formulierung von Konsistenzbedingungen definiert. Erzwungen wird die Konsistenz durch die Überprüfung dieser Konditionen. Die Konsistenzsicherung spielt sich daher auf zwei Ebenen ab. Zum einen ist der Ursprung der Bedingungen zu betrachten und zum anderen deren Gültigkeitsbereich.

Bedingungen können verschiedenen Ursprungs sein:

Inhärente Konsistenzbedingungen
sind Bedingungen, die durch das Datenmodell bestimmt sind und deshalb problemunabhängig sind. Als Beispiel könnte herangezogen werden, dass Zyklen in den Vererbungsbeziehungen nicht zugelassen sind.

Implizite Konsistenzbedingungen
sind Bedingungen, welche im Schema spezifizierbar sind. Dazu gehören beispielsweise Kardinalitätsbedingungen oder Konditionen, welche die Referenzsicherheit gewährleisten. Unter Referenzsicherheit wird die Sicherheit verstanden, dass zum einen keine Objekte in der Datenbasis gespeichert werden, die von keinem anderen persistenten Objekt referenziert werden und zum anderen keine Referenzen auf Objekte, die bereits gelöscht sind, übrigbleiben.
Eine Kardinalitätsbedingung ist dann erfüllt, wenn die minimale (maximale) Anzahl der Beziehungen die Kardinalitätsangabe nicht unterschreitet (überschreitet).

Explizite Konsistenzbedingungen
sind Bedingungen, die der Entwerfer des Datenbankschemas explizit formuliert hat, um die Integrität seiner Basis semantisch zu sichern. Dies können Bedingungen sein, die direkt als Klasseneigenschaft (Constraint) formuliert werden und dementsprechend auch für jede Instanz gelten müssen, oder beliebige Ausdrücke, die mehrere Objekte miteinander verknüpfen.

Für den Gültigkeitsbereich von Konsistenzbedingungen sind zwei Typen zu unterscheiden:

Invariante Konsistenzbedingungen
müssen immer erfüllt sein und werden beispielsweise am Ausführungsende von öffentlichen Methoden überprüft.

Allgemeine Konsistenzbedingungen
dürfen während der Ausführungszeit einer Transaktion verletzt sein und werden am Ende einer Transaktion überprüft.

Konsistenz der „leeren" Basis

Bei der Konsistenzsicherung für den Initialzustand muss auf der einen Seite gesichert sein, dass die „leere" Basis konsistent ist. Dies ist, da sämtliche formulierbaren Bedingungen an Instanzen gebunden sind, trivialerweise immer der Fall. Auf der anderen Seite, namentlich beim Übergang auf den Initialzustand, müssen sämtliche Einstiegspunkte durch die entsprechenden Klasseninstantiierungen kreiert werden. Dieser Prozess lässt sich in eine Transaktion schachteln, die der Reihe nach die globalen Objekte instantiiert, und dementsprechend auf den zweiten diskutierten Fall der Konsistenzsicherung zurückführen.

Konsistenz bei Create, Update und Delete

Die Konsistenz kann also nur bei der Kreation (Create), der Aktualisierung (Update) und der Löschung (Delete) gefährdet sein und muss demnach nur nach einer derartigen Aktion überprüft werden. Insbesondere können reine Lesezugriffe die Konsistenz nicht gefährden.

3.4.3.4 Schwachstelle

Die Entwurfssprache von Booch lässt die Formulierung von sogenannten **Constraints** als Eigenschaft einer Klasse zu, welche semantisch als Invarianten für Objekte zu interpretieren sind. Im weiteren können durch die Angabe von Kardinalitäten bei der Definition von Beziehungen implizite Konstistenzbedingungen formuliert werden.
Für die Sicherung der Konsistenz innerhalb der Datenbankumgebung, welche selber lediglich inhärente Konstistenzbedingungen prüfen kann, sind diese Möglichkeiten zu wenig mächtig. Die Konsistenzsicherung muss in zwei Dimensionen betrachtet und spezifiziert werden können. Zum einen ist die Art der Bedingung (inhärent, implizit und explizit) zu differenzieren und zum anderen der Gültigkeitsbereich (Bedingung muss stets erfüllt sein (Invariante) und Bedingung darf während der Ausführung einer Transaktion verletzt sein).
Im weiteren fehlt im Datenbanksystem die Möglichkeit zur Formulierung von existenzabhängigen Aggregationsbeziehungen. Damit verfügt sie über keinerlei Regeln für die Fortpflanzung von Erzeugungs- oder Löschaktionen.
Im konzeptuellen Entwurf müssen also zum einen Konstrukte angeboten und zum anderen ein Vorgehen definiert werden um dieser Problematik bereits während der Entwurfsphase mächtig zu werden.

3.4.4 Objektevolution

In objektorientierten Datenbanken werden Objekte meist über einen langen Zeitraum gespeichert. In vielen Anwendungen sollten aber einige Objekte ihre Klassenzugehörigkeit wechseln können. Diese Eigenschaft wird „Objektevolution" oder „Instanzmigration" genannt.

Objektevolution durch Zerstörung und Erzeugung

Weder die betrachtete Entwurfsmethode noch die darunterliegende Datenbankimplementation unterstützen dieses Konzept. Die Problematik besteht nun darin, dass bei einer Simulation der Instanzmigration durch die Zerstörung der Ausgangsinstanz und die Erzeugung einer Zielinstanz die Objektidentität nicht erhalten werden kann. Dies bedeutet, dass sämtliche Beziehungen, welche das „alte" Objekt unterhalten hat, gelöscht werden und sämtliche Verbindungen, die zu diesem Objekt unterhalten worden sind, nun keine Gültigkeit mehr haben. Darüber hinaus sind natürlich ohne spezielles Einwirken des Programmierers sämtliche Eigenschaften des zerstörten Objektes verloren, da die Attributwerte gelöscht werden.

Es muss also ein Objektevolutionskonstrukt eingeführt werden, welches die folgenden Problematiken lösen kann:

Behandlung von Referenzen zu anderen Objekten
Es muss entschieden werden, ob die Referenzen gelöscht (erzeugt), oder beibehalten werden sollen.

Behandlung von referenzierenden Objekten
Diejenigen Objekte, die zu der zu löschenden (zu erzeugenden) Instanz Beziehungen unterhalten, müssen von der Migration informiert werden. Dies könnte, wenn die Mittel von aktiven Datenbanken nicht zur Verfügung stehen oder nicht zu komplizierten Transaktionsblöcken gegriffen werden will, mit rückbezüglichen Referenzen gelöst werden.

Behandlung von Selbstreferenzen
Können wie gewöhnliche Referenzen behandelt werden.

Behandlung von Referenzen zwischen der Ausgangs- und Zielinstanz
Können wie gewöhnliche Referenzen behandelt werden.

Datenübernahme
Es muss entschieden werden, welche der Daten auf die neue Instanz übertragen (von der alten Instanz übernommen) werden sollen.

Evolutionskontrolle
Es muss einer dritten Instanz die Kontrolle über den Evolutionsprozess übergeben werden, da weder die Ausgangs- noch die Zielinstanz diese Rolle wahrnehmen kann.

Die Datenbank kann aufgrund der fehlenden Möglichkeit der DBMS-Erweiterung nicht um die Fähigkeit der Objektevolution bereichert werden. Dennoch lässt sich durch die Einführung bereits erwähnter Konstrukte eine Migration simulieren. Die Fähigkeit einer Klasse, eine Instanzmigration einzugehen, ist dabei im statischen Teil (Modellierung der Objekteigenschaften) zu entwerfen, während der Zeitpunkt und die Bedingungen für eine Mutation im dynamischen Teil (Entwurf des Objektverhaltens) zu spezifizieren sind.

3.4.4.1 Schwachstelle

Die Schwachstelle besteht darin, dass der gewünschte Klassenwechsel eines Objektes durch eine Erzeugung und eine Löschung simuliert werden muss. Dabei geht aber die Objektidentität, welche für die Objektbeziehungen von zentraler Bedeutung ist, verloren. Es fehlt somit eine in vielen Anwendungen wichtige Eigenschaft von persistenten Instanzen.

3.4.5 Schemaevolution

Datenverlust und Integritätsverletzung

Während des Betriebes von - vor allem grösseren - Informationssystemen können Änderungen an der einer Anwendung zu Grunde liegenden Datenstruktur (Schema) erforderlich sein. Diese Anpassungen dürfen auf keinen Fall zu Datenverlusten oder Integritätsverletzungen führen.

Auf der konzeptuellen Ebene wird dieses Konzept gänzlich vermisst, während auf der logischen Ebene zuweilen Vorgehensweisen beschrieben sind um eine „altes" in ein „neues" Schema zu übertragen.
In relationalen Datenbanksystemen kann man dazu den Schemamanipulationsteil der Abfragesprache SQL zur Hilfe nehmen. Befehle wie `ALTER TABLE` dienen z.B. zur Veränderung von Datentabellen oder Relationen.

Hinzufügen, Entfernen oder semantisch Verändern

Problematisch bei der Schemaevolution sind in der Regel das Hinzufügen von neuen Relationen, Attributen oder Integritätsbedingungen, sowie das Entfernen oder semantische Verändern (Ändern von Wertebereichen). Einerseits ist der Grund hierfür in der Tatsache zu suchen, dass in SQL ein adäquater Befehlssatz für das Entfernen von Attributen fehlt, andererseits ist es im allgemeinen unmöglich, unter Beibehaltung der Konsistenzbedingungen die Integrität der Daten nach der Veränderung zu gewährleisten.

Im allgemeinen sind damit Schemaevolutionen nicht während des Betriebes der Datenbank möglich. Üblicherweise wird der Betrieb der Datenbank während des Änderungsprozesses vom Zugriff durch Benutzer geschützt.
Eine Schemaevolution müsste daher etwa in folgenden Schritten unternommen werden:

1. Datenbank von den Zugriffen der Benutzer schützen.
2. Prüfung der Konsistenzbedigungen unterdrücken.
3. Schema verändern (Zustand ist nun möglicherweise inkonsistent).
4. Daten ins neue Schema übernehmen (neu geschaffene Attribute müssen mit sinnvollen Standardwerten gefüllt werden).
5. Formulierung der Konsistenzbedigungen anpassen.
6. Prüfen der Konsistenz durch Evaluation aller Konsistenzbedingungen.

„Lazy Migration"

Für den konzeptuellen Entwurf würde das bedeuten, dass bei der Modellierung des dynamischen Verhaltens der Aspekt der Schemaevolution miteinbezogen werden müsste. Es müssten also Konstrukte oder ganze Diagrammtypen zur Verfügung gestellt werden, mit deren Hilfe ein Schemaübergang spezifiziert werden könnte. Damit würde man die Evolutionsfähigkeit einer Datenbasis zu den normalen dynamischen Eigenschaften eines Systems zählen. Dies würde vor allem für Datenbanksysteme passen, welche die „Lazy Migration" (automatische Migration beim Lesen von Objekten) implementieren.

Migrationsskript Ein anderer Ansatz würde auf einen einmaligen Migrationprozess abzielen. Man würde hierzu den bestehenden Entwurf zur Hand nehmen, diesen anpassen und sich von einem Werkzeug die Differenz in Form eines Migrationsskriptes berechnen lassen. Dazu müsste aber auf der einen Seite das Datenbanksystem einen Instruktionssatz für die Schemaevolution anbieten und auf der anderen Seite müsste das Migrationswerkzeug sicherstellen können, dass das Ausgangsdiagramm auch wirklich der aktuellen Implementation des Schemas entspricht (Reverse Engineering).

3.4.6 Sichtenkonzept

„Named Query" Weder konnte sich die ODMG-93 auf ein Sichtenkonzept einigen, welches mit seinem relationalen Pendant vergleichbar wäre, noch sind Ansätze davon in heutigen Implementationen von objektorientierten Datenbanksystemen zu finden. In OQL können zwar durch die Verwendung des Sprachkonstruktes „Named Query" Datenbankanfragen gespeichert werden, jedoch wird dadurch höchstens die Lesbarkeit von umfangreichen „SELECT"-Befehlen unterstützt.

Extensionen als Sichten Häufig werden, wie z.B. in O_2, sichtenähnliche Klassen - sogenannte Extensionen - implementiert. Diese bilden aber nicht einen Bestandteil der Datenbankumgebung, vielmehr müssen diese von den Programmieren selber aktuell gehalten werden.

Extensionen werden eingesetzt, um gewisse Objekte anhand eines Schlüsselwertes, der sich im Allgemeinen von der Objektidentität unterscheidet, in einer Menge von Objekten einzufügen oder aufzufinden.

attributierte Beziehungen Durch die Erweiterung des in der Booch'schen Notation eingeführten Konstruktes der attributierten Beziehung, oder durch die Definition von dedizierten Klassen liesse sich ein Sichtenkonzept entwerfen. Es gälte dabei zu regeln, wie die Sicht aufgebaut und wie sie aktuell gehalten werden soll, wenn die Datenbank keine Triggersprache unterstützt.

Bei der Modellierung von Sichten als Klassen müsste die folgende Funktionalität implementiert sein:

- Die Klasse muss einen Aufbaumechanismus (Konstruktor) unterstützen. Soll die Sicht als transiente Klasse modelliert sein, kann dies bereits für die Garantierung der Aktualität der Daten genügen, falls sie vor jeder Selektion kreiert wird.
- Die Klasse muss einen Aktualisierungsmechanismus zur Verfügung stellen. Da die objektorientierten Datenbanken keine Triggersprachen enthalten, müssen die beobachtenden Klassen über Meldungen für die Änderungen der beobachteten Daten informiert werden. Beachtet werden muss dabei, dass sich die Persistenz nicht unerwünscht auf diese Klassen fortpflanzt. Eine Aktualisierung könnte über die Formulierung von geeigneten Konsistenzbedingungen und den Entwurf von strukturierten Transaktionen gewährleistet werden. Der Programmierer würde sich dabei aber den ohnehin zur Verfügung stehenden Mechanismen bedienen müssen, weshalb dieser Ansatz nicht als spezielles Sichtenkonzept zu betrachten ist.
- Die Klasse muss einen Satz Zugriffsfunktionen anbieten. Diese öffentlichen Methoden sollten es erlauben, die Gesnamtmenge zu

selektieren (beispielsweise für die Formulierung eines Sub-SELECTes), über die Menge zu iterieren, oder ein bestimmtes Objekt, welches einer gewissen Bedingung genügt, aufzufinden
Für die Deklaration (Aufbau) der Klasse müssen folgende Eigenschaften definiert sein:

- Deklaration der Referenzen und deren Namen (Alias)
- Die Art der Verknüpfung von mehreren Instanzen verschiedener Klassen (Joins)
- Formulierung der einschränkenden Bedingungen (Selektion)
- Einschränkung der Sicht (Projektion)

Zusammenfassend lässt sich also festhalten, dass in den objektorientierten Datenbanken vorerst kein Konzept für Sichten existiert. Zudem schätzen die Entwerfer von grossen Systemen den Nutzen als nicht allzu gross ein. Damit lässt sich die Einführung eines Sichtenkonzeptes nicht als dringendes Bedürfnis der Praxis betrachten.
Zudem stehen dem Entwickler bei einer Unterstützung von Transaktions- und Konsistenzprüfungsmechanismen ein mächtiges Werkzeug zur Verfügung, um sichtenähnliche Klassen in das Datenbankschema mit einzubeziehen.

3.5 Weitere Schwachstellen

In diese Kategorie fallen Schwachstellen, die nicht einem speziellen Typ zugeordnet werden können, aber dennoch betrachtet werden sollten. Diese Unzulänglichkeiten stammen aus dem Erfahrungsschatz von Datenbankadministratoren, die in der Praxis bereits grosse Datenbanksysteme entworfen und in die Produktion überführt haben.

3.5.1 Abfragen auf NULL-Werten

In OQL können Abfragen definiert werden, die eine Ausnahme zur Folge haben.
Beispiel Bibliothek: Studenten können Bücher ausleihen, müssen aber nicht. Wenn eine Liste derjenigen Bücher erstellt werden soll, die von einem Studenten A ausgeliehen sind, wird folgende Abfrage gegenüber der Datenbank abgesetzt:

```
SELECT s->mat_nr, s->refBook->title
FORM s IN Student
WHERE s->name="Müller"
```

Die Situation, dass ein Student kein Buch ausgeliehen hat ist durchaus zulässig, führt aber bei genannter Abfrage zu einer Ausnahme. Vielmehr müsste zuerst die Beziehung durch die Formulierung einer Bedingung validiert werden.

```
SELECT s->mat_nr, s->refBook->title
FORM s IN Student
WHERE s->name="Müller"
AND s->refBook NOT NULL
```

Damit muss man sich aber darauf verlassen können, dass der OQL-Compiler wirklich zuerst die Bedingung überprüft, bevor er den Wert im nicht-referenzierten Objekt beschaffen will.
Noch schlimmer verhält es sich, wenn für die Prüfung der Bedingung die vielleicht nicht definierte Referenz benötigt wird. Beispielsweise

soll der Student gefunden werden, der das Buch „Faust“ ausgeliehen hat.

```
SELECT s->name
FORM s IN Student
WHERE s->refBook->title=“Faust“
```

Also müsste auch hier die Prüfung der Referenz erfolgen:

```
SELECT s->name
FORM s IN Student
WHERE s->refBook NOT NULL
AND s->refBook->title=“Faust“
```

Damit liefert man sich - wie oben aufgezeigt - der Willkür des verwendeten OQL-Interpreters völlig aus, indem man im vornherein nicht mit Sicherheit sagen kann, welcher Teil der Bedingung zuerst ausgewertet werden wird.
Sicherheit kann einem hier nur dann geboten werden, wenn man zuerst eine Selektion unter dem Kriterium der definierten Fremdbeziehung absetzt und anschliessend über diese Untermenge weiter einschränkt.

```
SELECT s->name
FORM s IN
              SELECT s
              FORM s IN Student
              WHERE s->refBook NOT NULL
WHERE s->refBook->title=“Faust“
```

3.6 Evaluation

Es soll nun entschieden und begründet werden, welche der beobachteten Schwachstellen bei der Erweiterung der Booch’schen Methode berücksichtigt werden sollen (vgl. [Booch 94, p. 173]).

Konsistenzsicherung versus Entwurfsfreiheit

Als generelles Ziel soll vor allem die Verbesserung des Schutzes der Datenintegrität (Konsistenzsicherung) verfolgt werden. Die Freiheit des Entwerfers soll dabei nicht im Vordergrund stehen.

Machbarkeit, Überprüfbarkeit

Weitere für die Auswahl wichtig Kriterien sind die Machbarkeit (zu welchem Grad lässt sich die Schwachstelle beheben?) und die Überprüfbarkeit durch automatisierte Werkzeuge (können Entwurfsfehler erkannt werden?). Zudem ist es fragwürdig, ein Problem weiterzuverfolgen, das nicht standardisiert oder auf eine bestimmte Zielumgebung zugeschnitten ist.

3.6.1 Schwachstelle auf logischer Ebene

Wie Booch in seiner Beschreibung des Prozesses angibt, sollen Entwurfskonstrukte, die in der betrachteten Zielumgebung fehlen, nicht verwendet werden.
Aus diesem Grund werden sämtliche der angetroffenen Schwachstellen auf der logischen Ebene nicht in die neue Methode einbezogen.

3.6.2 Schwachstellen auf der konzeptuellen Ebene

Derartige Schwachstellen zeigen den eklatanten Handlungsbedarf und müssen allesamt in der neuen Methode ausgeräumt sein.
Namentlich sind dies die Schwachstellen, welche im Zusammenhang mit

- der Persistenz,
- den Applikationsklassen und -instanzen und
- den Transaktionen

stehen. Von der weiteren Bearbeitung seien einzig die physischen Aspekte ausgeklammert.

3.6.3 Vermisste Konstrukte

Bei dieser Klasse von Schwachstellen gilt es, die Dringlichkeit und den Nutzen abzuschätzen.

Standard für objektorientierte Datenbanksysteme

Die Dringlichkeit definiert sich aus der Beantwortung der Frage, woher die Forderung nach diesem Konstrukt stammt. Stammt sie aus den von der ODMG-93 geforderten Standards für objektorientierte Datenbanksysteme (inverse Beziehungen, Schlüssel), soll dieser Forderung eher nachgekommen werden, als wenn sie aus den Wünschen eines Systemadministrators (Sichten, Schemaevolution) herrührt.

Nutzen für das Entwicklungsteam

Zudem kann bezüglich dem Nutzen, der einem Entwicklungsteam durch die Unterstützung eines gewissen Konzeptes entsteht, unterschieden werden. Handelt es sich um Strategien, z.B. für Fortpflanzungs- und Konsistenzprüfungsmechanismen oder Objektevolution, so ist der Nutzen für alle Anwendungsbereiche gross, während die Konstrukte für die Formulierung von sicheren Abfragen vermutlich eher mit geringerer Priorität zu behandeln sind.

Zusammenfassend werden also die Schwachstellen in den Bereichen

- Inverse Beziehungen,
- Extensionen und Schlüssel,
- Fortpflanzung und Konsistenzsicherung und
- Objektevolution

weiterverfolgt.

3.6.4 Weiteres Vorgehen

Aus oben genannten Beweggründen werden die im Folgenden aufgezählten Schwachstellen weiter untersucht. Für jede dieser Unzulänglichkeiten ist skizziert, in welcher Form Anstrengungen für deren Behebung in der erweiterten Methode unternommen werden sollen.

3.6.4.1 Konzeptuelle Ebene

Persistenz und Persistenzfortpflanzung

Persistenz und Persistenzfortpflanzung: Die Notation muss derart erweitert werden, dass die Unterscheidung zwischen persistenzfähigen und transienten Klassen ermöglicht wird. Für die Definition der persistenten Einstiegspunkte muss ein neues Konstrukt angeboten werden. Von diesem Ausgehen muss die Fortpflanzung der Persistenz durch die Hinterlegung einer entsprechenden Semantik für gewisse Beziehungstypen kontrolliert werden können.

Applikationsklassen und -instanzen

Applikationsklassen und -instanzen: Durch die Einführung von Kontrollmechanismen über die Dauerhaftigkeit von Objekten können bereits viele Aspekte dieser Schwachstelle behoben werden. Trotzdem muss die Notation die Spezifikation von Schnittstellen zwischen der Applikations- und der Datenschicht erlauben.

Transaktionen

Transaktionen: Ein zentraler Teil des Applikationsentwurfes bildet die Definition der Transaktionsblöcke. Die Notation muss also dergestalt ausgebaut werden, dass der Entwerfer durch die Verwendung gewisser Konstrukte die Kontrolle über die Datenbankoperationen in die Hand nehmen kann.

3.6.4.2 Vermisste Konstrukte

Inverse Beziehungen

Inverse Beziehungen: Das bereits vorhandenen Beziehungskonstrukt für die Modellierung von Assoziationen soll für die Belange der inversen Beziehungen ausgeweitet werden.

Extensionen und Schlüssel

Extensionen und Schlüssel: Die Notation soll nicht neue Konstrukte für die Darstellung von Extensionen einführen. Vielmehr sollen gewisse Konstellationen von Klassenstrukturen semantisch Extensionen definieren. Schlüssel sollen demzufolge nicht zu den Eigenschaften einer überwachten Klasse gezählt werden, sondern als Eigenschaft der Extensionsklasse betrachtet werden.

Fortpflanzung und Konsistenzsicherung

Fortpflanzung und Konsistenzsicherung: Damit beispielsweise die Löschfortpflanzung kontrolliert werden kann, sollen für die Verwendung von Beziehungskonstrukten strenge Regeln definiert werden. Diese grenzen die Freiheit des Entwerfers in gewisse einem gewissen Mass ein, unterstützen aber die Aspekte der Konsistenzsicherung.

Objektevolution

Objektevolution: Der Einbezug der Objektevolution affektiert nicht nur das statische Diagramm, welches sich mit dem Entwurf der Objekte, Klassen und deren Beziehungen beschäftigt, sondern auch die dynamischen Diagramme, welche die Migration als ein Verhalten eines Objektes modellieren sollen. Im strukturellen Entwurf muss ein neues Konstrukt angeboten werden, welches die Fähigkeit einer Klasse zur Instanzmigration definiert, während im dynamischen Entwurf die vorhandenen Elemente „Zustand“ und „Zustandsübergang“ genügen sollen.

4 DEIMOS

4.1 Einleitung

Während im Kapitel 2 die Methode von Booch erklärt wurde und im Kapitel 3 die Schwachpunkte bei deren Anwendung für den konzeptuellen Datenbankentwurf analysiert worden sind, soll nun in diesem Kapitel eine Erweiterung DEIMOS eingeführt werden, welche diese Schwachstellen beseitigt.

4.1.1 Unterschiede zu der Methode von Booch

Die Methode von Booch soll weitgehend übernommen werden. Dennoch sind für die speziellen Aspekte des Entwurfes von objektorientierten Datenbanksystemen einige Erweiterungen sowohl in den Bereichen der Notation als auch im Vorgehen nötig.

4.1.1.1 Die Notation

Um die im Kapitel 2 identifizierten Schwachstellen ausräumen zu können, müssen verschiedene Elemente der Booch'schen Notation verändert oder erweitert werden. Zudem müssen neue Konstrukte in die Diagramme eingeführt werden, um den erweiterten Anforderungen des Datenbankentwurfes genügen zu können.
Die speziellen Eigenschaften von Datenbankklassen sind im statischen Diagramm, welches zur Modellierung der Klasseneigenschaften und -strukturen dient, spezifizierbar, weshalb dieses Diagramm entscheidend erweitert wurde und als Schemadiagramm seinen Platz in der Methode DEIMOS erhält.
Da für den Entwurf von Datenbankapplikationen das Design der Transaktionen von zentraler Bedeutung ist, wird in Anlehnung an das Moduldiagramm ein neues Diagramm, genannt Transaktionsdiagramm, in die Notation aufgenommen.
Die Booch'schen Diagramme sollen dabei keineswegs aus ihrer Position verdrängt werden, vielmehr sollen die neuen Diagrammtypen zur Anwendung gelangen, wenn die bisherigen Diagramme die Aspekte des Datenbankentwurfes nicht genügend berücksichtigen können.

4.1.1.2 Das Vorgehen

Wie bereits im vorigen Kapitel ausgeführt, beschäftigt sich der Makroentwicklungsprozess schwergewichtig mit den gewöhnlichen Managementaufgaben für die Realisierung eines Entwicklungsprozesses. Die notwendigen Erweiterungen im Vorgehen beziehen sich darum lediglich auf den Mikroentwicklungsprozess, der gewisse Spezifika von Datenbankentwicklungen berücksichtigen muss.
Für die Identifikation der Semantik der Klassen und Objekte müssen für den Entwurf von objektorientierten Datenbanksystemen zusätzliche Aspekte in diesen Schritt

miteinbezogen werden. Zum einen muss die gewünschte Lebensdauer eines Objektes untersucht werden, was sich darin äussert, dass die entsprechenden Klassen die Eigenschaft der Persistenzfähigkeit besitzen müssen. Zum anderen ist es für den späteren Entwurf der Transaktionen unerlässlich, zwischen lesenden und schreibenden Zugriffen auf Objekte zu unterscheiden.
Nachdem bei der Identifikation der Objektsemantik zwischen persistenten und transienten Objekten unterschieden wurde, muss entsprechend für die Identifikation der Beziehungen zwischen den Klassen dieser Umstand berücksichtigt werden. Ein spezielles Augenmerk sollte also auf die Umsetzung der Persistenzfortpflanzung geworfen werden.
Zudem müssen beim Umgang mit persistenten Objekten die Konsistenzsicherung und die Fortpflanzungsmechanismen in den Entwurf miteinbezogen werden.

4.1.2 Das Beispiel FIS

Um die verschiedenen, in den nachfolgenden Abschnitten neu eingeführten oder erweiterten Konstrukte zu veranschaulichen, wird Schritt für Schritt ein jeweiliges Diagramm für die Modellierung eines Informationssystems für die Firma „Macrohard“ aufgebaut.
Dem Firmeninformationssystem (FIS) liegen folgende Ausgangsinformationen zu Grunde:

- Die betrachtete Firma ist in der Softwareentwicklung tätig. Die Entwicklungsaufträge werden von den Entwicklern bearbeitet und von den Managern koordiniert.
- Die Firma besitzt verschiedene Büros als Räumlichkeiten. Jedem Mitarbeiter steht ein Arbeitsplatz in einem der Büroräume zur Verfügung. Büros dürfen auch leer stehen.

Das System muss folgende Manipulationen unterstützen:

- Anstellung und Entlassung von Entwicklern und Managern.
- Mieten von Büroräumen und deren Kündigung.
- Mitarbeiter einem Büro zuteilen oder die Zuteilung ändern.
- Entwickler einem Projektteam (Manager) zuordnen oder Zuordnung ändern.
- Entwickler zu Managern befördern.

Der Benutzer muss über folgende Sachverhalte Auskunft erhalten können:

- Welches sind die Mitarbeiter der Firma?
- Welches sind die Mitglieder eines vom Manager XY koordinierten Projektteams?
- Welche Büros werden von welchen Mitarbeitern genutzt?

Das Firmeninformationssystem soll nun gemäss Kundenwunsch innerhalb der objektorientierten Datenbankumgebung O_2 realisiert werden und über eine benutzerfreundliche Anwendungen zugreifbar sein.

4.2 Das Schemadiagramm

In der Methode von Booch wird bei der logischen Ansicht eines Systems das Klassendiagramm verwendet, um die Existenz von Klassen und deren Abhängigkeiten zu veranschaulichen. Das Diagramm repräsentiert eine Ansicht der Klassenstruktur und dient dazu, für jede Klasse deren Rollen und Verantwortlichkeiten zu entwerfen, damit das gewünschte Verhalten des Systems erzielt werden kann.

Klassendiagramm - Schemadiagramm

Für den Entwurf eines Datenbanksystems ist dieser Schritt analog aufzufassen. Da aber nicht das Design der Applikation im Vordergrund steht - Anwendungen selber sind innerhalb der objektorientierten Datenbankschemata nicht als Klassen modelliert - sondern der Entwurf des Datenbankschemas, ist der von Booch vorgeschlagene Diagrammtyp in seiner Idee und seiner Stellung innerhalb des Prozesses übernommen, aber in Schemadiagramm umbenannt worden.

Nachfolgend sollen die Konstrukte des Diagramms vorgestellt und insbesondere deren Gemeinsamkeiten, bzw. Unterschiede zu den Booch'schen Entsprechungen erklärt werden. Dabei werden erst die Klassen und deren Varianten, die verschiedenen Beziehungen zwischen den Klassen und schliesslich die weiteren Elemente des Diagramms erläutert.

4.2.1 Klassen

Datenbankklassen, Hilfsklassen und abstrakte Klassen

Es werden drei verschiedene Klassentypen, die Datenbankklassen, die Hilfsklassen und die abstrakten Klassen, eingeführt. Datenbankklassen können Instanzen bilden, die persistent sind, während dies für Hilfsklassen nicht zugelassen ist und dementsprechend verhindert werden muss. Von abstrakten Klassen können hingegen gemäss der bekannten Auffassung von Objektorientierung keine Instanzen gebildet werden. Wird im Folgenden der Begriff „Klasse" verwendet, so ist dieser als Oberbegriff der drei erwähnten Ausprägungen zu verstehen.

Die Darstellung und die Semantik werden von der Booch-Methode weitgehend übernommen. Die Klassen werden in der Form von durch gestrichelte Linien begrenzte Wolken [Booch 94, p. 177] dargestellt. Der Einfachheit halber werden sie aber in diesem Dokument als Hexagone gezeichnet.

Der Name muss für ein Gesamtdiagramm eindeutig gewählt werden, da für Datenbanksysteme dieser Name für ein Schema eindeutig sein muss.

4.2.1.1 Eigenschaften

Attribute und Methoden

Mit der Klasse können deren Attribute und Methoden definiert werden. Üblicherweise werden in dieser Stufe der Abstraktion nicht alle notwendigen Methoden und Attribute definiert, da zum Zeitpunkt des Entwurfes die genauen Bedürfnisse nach Daten und Fähigkeiten noch nicht feststehen. Trotzdem sollten aber sämtliche von aussen sichtbare (öffentliche) Attribute und Methoden (public members) spezifiziert sein, damit bei Validierungsmethoden wie Structured Walkthrough Schwachstellen des Entwurfes aufgedeckt werden können. Zudem ist die frühe Definition der Schnittstellen für das Zusammenspiel verschiedener Komponenten von grosser Wichtigkeit.

Attribute können in der Form

A : C [= D]

definiert werden. Mit A wird der Name des Attributes angegeben, während mit C sein Typ (primitiver Datentyp oder im Schema definierte Klasse) spezifiziert wird. Bei Bedarf kann das Attribut durch einen Standardwert D initialisiert werden.

Schlüssel

Ist die Klasse C_1 ein Aggregat verschiedener Instanzen der Klasse C_2, so ist es sinnvoll, dass die Extensionsklasse C_1 ihre Komponenten mittels eines Schlüssels kontrolliert. „Schlüssel“ soll in diesem Zusammenhang bedeuten, dass jede Instanz dieser Klasse bezüglich der im Schlüssel definierten Attribute eindeutig identifizierbar sein muss. Die Eindeutigkeit ist dabei nur im Bereich der Extension gewährleistet. Attribute, welche zum Schlüsselkriterium gezählt werden sollen, sind unter Angabe der überwachten Klassen in spitzen Klammern einzuschliessen.

Class<*Attrib*[(,*Attrib*)*]>

Häufig werden implizite Attribute, z.B. Speicherplätze für das Ablegen der Referenzen auf andere Objekte, der Übersicht zuliebe nicht in den Entwurf miteinbezogen und entsprechend im Schemadiagramm den Klassenikonen nicht einbeschrieben.

Methodendeklarationen sind in der Form

[R|W] M(A : T[, A : T]*):V

zu formulieren. Dabei entsprechen die „A“s den Namen der Parameter und entsprechend die „T“s deren Typen. Mit „V“ wird der Typ des Rückgabewertes spezifiziert. „M“ sei der Name der Methode.

Steht zu Beginn der Methodendefinition ein „W“, so beinhaltet die Methodenimplementierung schreibende Zugriffe auf die Attribute oder ruft ihrerseits „W“-Methoden ihrer Komponenten auf, während bei vorangestelltem „R“ oder keiner Angabe die Attributwerte nur gelesen werden. Diese Unterscheidung ist für den Entwurf der Transaktionen von grosser Bedeutung.

Constraints

Den Klassen können auch sogenannte Constraints angehängt werden. Diese müssen in der Form

{E}

definiert werden. „E“ stehe dabei für einen beliebigen logischen Ausdruck. In diesem Zusammenhang bedeutet ein derartiger Constraint, dass keine Instanz der Klasse diese Bedingung verletzen darf. Solche Einschränkungen sollen in diesem Kontext Invarianten genannt werden. Das heisst, sie müssen sowohl direkt nach der Kreation, als auch am Ende jedes Aufrufes einer öffentlichen Methode erfüllt sein.

Invarianten

Die Invarianten können also lediglich auf die Instanz selbst wirken, d.h. es lassen sich mit diesem Konstrukt keine Interobjekt-Constraints formulieren. Letzere müssen daher beim Entwurf der Transaktionen mitberücksichtigt und entsprechend global definiert werden.

4.2.2 Beziehungen

Beziehungen sind die Verbindungen zwischen den eingeführten Klassentypen. Sie definieren das Zusammenspiel und die Zusammensetzung der zu modellierenden Entitäten. Beziehungen werden zwar zwischen Klassen gezeichnet, wirken aber tatsächlich zwischen Objekten. Aus diesem Grund muss durch die Angabe von Kardinalitäten angegeben werden können, zu wievielen Objekten eine bestimmte Instanz Relationen haben kann und wieviele auf sie selber wirken dürfen.

Kardinalitäten

Etwas abweichend zu Booch werden folgende Kardinalitäten zugelassen:

n	genau n (n>0)
n+	beliebig viele, aber mindestens n (n>=0)
C	entweder eine oder keine (0|1)
[a..b]	mindestens a, höchstens b

Sämtliche aufgelisteten Kardinalitäten liessen sich auch mit dem letzten Konstrukt [a..b] darstellen, werden aber zu Gunsten der Lesbarkeit in der Kardinalitätsliste belassen, da sie in verschiedenen anderen Methoden verwendet werden.
In den nachfolgenden Diagrammen werden vor allem die folgenden Kardinalitäten angetroffen werden.

1	genau 1
1+	eine oder mehrere
0+	beliebig viele

Kardinalitäten werden „grösser als 1“ genannt, wenn sie mindestens eine Beziehung verlangen, also eine Existenzabhängigkeit beschreiben. Dies trifft demnach auf die Bezeichnungen n und n+ mit (n>0) und [a..b] mit a>0 zu.
Wird bei einer Beziehung keine Kardinalität angegeben, wird dies analog zu Booch’s Notation nicht als Standardwert, sondern als „Nichtwissen“ interpretiert. Die genaue Spezifikation der Mengenangabe ist in einem späteren Detaillierungsgrad nachzutragen.

4.2.3 Datenbankklassen

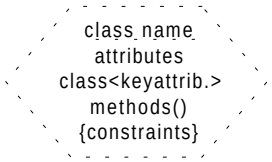
Die zentrale Aufgabe einer Datenbank besteht darin, Daten über die Lebenszeit des Prozesses, in dem sie erzeugt wurden, hinaus zu erhalten. Objektorientierte Systeme sollen als dauerhafte Daten

Objekte speichern. Sollen nun derartige Objekte in die Datenbasis gelangen, so müssen sie Instanzen von Datenbankklassen sein.

persistente Instanzen

Datenbankklassen zeichnen sich also dadurch aus, dass sie persistente Instanzen haben können. Die Datenbasis besteht also immer nur aus Objekten dieses Klassentyps, was ihm den Namen „Datenbankklasse“ eingetragen hat.

4.2.3.1 Konstrukt



In der Methode von Booch wird ein Konstrukt für die Darstellung von Klassen im Allgemeinen verwendet. Dieses Symbol wird direkt übernommen.

4.2.3.2 Beispiel FIS

Zur Identifikation der Datenbankklasse werden zuerst die Objekte betrachtet. Danach werden gleichartige Objekte zu Objektklassen zusammengefasst.
In der Firma sind Entwickler und Manager beschäftigt. Entwickler und Manager haben weder die gleichen Rechte noch die gleichen Pflichten, sie sind also unterschiedlich zu behandeln. Zudem besitzt die Firma Büros. Daraus lässt sich die Existenz von verschiedenen „Entwickler“-, „Manager“- und „Büro“- Objekten ableiten. Der Firma können Fähigkeiten wie das Anstellen von Entwicklern/Managern oder das Mieten von Büros zugeordnet werden, was sie selbst auch zu einem Objekt werden lässt.
Die einzelnen Entwickler unterscheiden sich - zumindest aus der Betrachtungsweise des FIS - nicht, was die Zusammenfassung dieser Objekte zu einer Klasse nahelegt. Analog lassen sich so die Klassen Manager und Büro herauskristallisieren. Die Firma selber ist ein einzelnes Objekt und lässt sich so trivialerweise als Instanz einer Klasse betrachten.
Alle gefundenen Objekte sollen über die Lebensdauer der Anwendungen hinaus erhalten bleiben, was deren Speicherung in der Datenbasis nahelegt. Aus diesem Grund müssen sie als Instanzen von persistenzfähigen Klassen (Datenbankklassen) modelliert werden.
Den einzelnen Klassen werden, soweit dies zu dieser Stufe der Abstraktion möglich ist, bereits Attribute und Methoden zugeordnet.

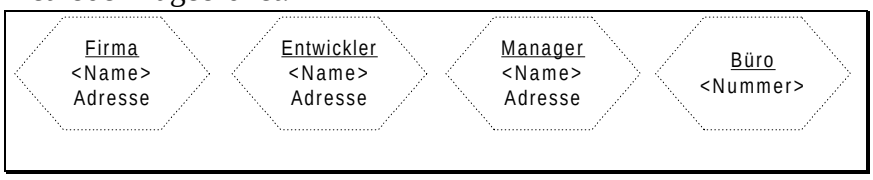


Abbildung 4-1: Datenbankklassen im Schemadiagramm

4.2.4 Hilfsklassen

transiente Objekte

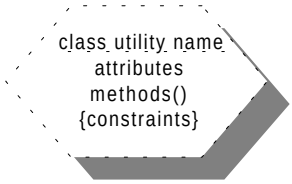
Nicht alle Objekte, die während der Bearbeitung der Datenbasis erzeugt werden, sollen nach Beendigung der Anwendung in der Datenbank gespeichert werden. Der Entwickler möchte für die Implementation von Programmen gleichsam transiente Objekte anlegen dürfen, ohne Gefahr zu laufen, dass diese unbeabsichtigterweise persistent werden.

Persistenz versus Transienz

Während Datenbankklassen dadurch gekennzeichnet sind, dass sie persistente Instanzen erzeugen können, muss für die Allokation von flüchtigen Objekten (Hilfsobjekten) ein weiteres Konstrukt eingeführt werden. Tatsächlich liessen sich sämtliche transiente Objekte auch als Instanzen von Datenbankklassen modellieren. Dies trüge aber den Nachteil in sich, dass vom System her keinerlei Schutzmechanismen greifen könnten, die den Entwerfer vor den angesprochenen Nebeneffekten bewahren würden.

Wird dem Entwerfer jedoch die Möglichkeit gegeben, bereits während des Entwurfes zwischen persistenten und transienten Objekten durch die Verwendung von unterschiedlichen Konstrukten zu differenzieren, kann durch systemunterstützte Validierungen verhindert werden, dass sich die Persistenz auf Hilfsobjekte ausdehnt.

4.2.4.1 Konstrukt



In Booch's Methode findet sich ein Konstrukt für „Class-Utilities". Dieses wird direkt übernommen. Die Hilfsklassen können also als Spezialisierung der vorgängig erläuterten Klassenkonstrukte aufgefasst werden.

4.2.4.2 Beispiel FIS

Aus der Anforderung, dass die Liste der Büros mit den jeweilig zugeteilten Mitarbeitern stets verfügbar sein muss, kann das Bedürfnis entstehen, eine Hilfsklasse zu bilden, welche die gewünschten Dienste wie Drucken, Anzeigen, usw. kapselt. Es wird also dem Diagramm eine weitere Klasse „Büroliste" als Hilfsklasse hizugefügt.
Denkbar wären weitere Klassen als reine Datenstrukturen, die z.B. die Adressinformationen aufnehmen könnten. Derartige Klassen sind aber innerhalb der Datenbankumgebung O_2 als tuple formulierbar und werden daher nicht als Klassen modelliert, falls sie nicht noch weitere Funktionalitäten aufweisen müssen.

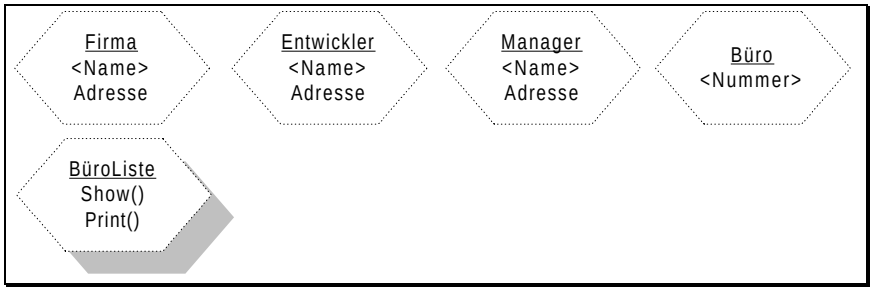


Abbildung 4-2: Hilfsklassen im Schemadiagramm

4.2.5 Vererbungsbeziehung

Generalisierung und Spezialisierung

Die für die Instantiierung der notwendigen Objekte benötigten Klassen werden in einem ersten Überarbeitungsschritt auf Gemeinsamkeiten untersucht. Häufig werden dabei Klassen gefunden, die als Spezialisierungen von anderen bereits im Entwurf gezeichneten Klassen aufgefasst werden können. Somit macht es aus der Sicht der Objektorientierung keinen Sinn, die Gemeinsamkeiten redundant zu implementieren. Vielmehr werden durch die Verwendung von Vererbungsbeziehungen die Eigenschaften der Superklasse auf die Unterklasse übertragen.

Werden Klassen gefunden, die einen gewissen Teil ihres jeweiligen Verhaltens gemeinsam haben, kann der Entwerfer diese Gemeinsamkeiten in eine neue Klasse auslagern, die anschliessend gegenüber der speziellen Klassen durch die Definition von neuen Vererbungsbeziehungen als Superklasse deklariert wird.

4.2.5.1 Konstrukt

Die Vererbungsbeziehung wird sowohl in der Darstellung als auch in der Semantik direkt von der Booch-Methode übernommen. Sie dient zur Darstellung von Generalisierungs- bzw. Spezialisierungsbeziehungen („is-a“). Die Vererbung wirkt zwischen zwei Klassen, wobei der Pfeil bei der Unterklasse beginnt und bei der Superklasse in die Spitze ausläuft.

4.2.5.2 Einschränkungen

Damit die Semantik der Vererbungsbeziehung nicht verletzt wird, müssen für dessen Verwendung einige einschränkende Regeln beachtet werden:

- Es dürfen keine Vererbungszyklen entstehen (eine Klasse darf also insbesondere keine Vererbungsbeziehung zu sich selber unterhalten).
- Eine abstrakte Klasse muss mindestens eine Unterklasse haben.
- Eine Klasse kann an mehreren solchen Beziehungen teilnehmen, sei es als Superklasse von mehreren Unterklassen, oder als Unterklasse mehrerer Superklassen (Mehrfachvererbung).

4.2.5.3 Beispiel FIS

Nachdem die Klassen identifiziert worden sind, soll sich der Entwerfer auf die Suche nach Klassen mit Gemeinsamkeiten machen. Diese Übereinstimmungen lassen sich in eine Superklasse auslagern, von der die entsprechenden Unterklassen die gemeinsamen Eigenschaften vererbt bekommen.

Das Ziel dieses Schrittes ist es zudem, Superklassen zu finden, deren Eigenschaften so allgemein wie möglich und so speziell wie nötig sind, damit sie in späteren Anwendungen als Komponenten wiederverwendet werden können.

Wenn für die bereits gefundenen Klassen Gemeinsamkeiten zu finden sind, drängen sich die beiden Klassen Entwickler und Manager nachgerade auf. Beide Klassen beschreiben eine Person und einen Mitarbeiter. Eigenschaften dieser gemeinsamen Basis können also in eine Superklasse ausgelagert werden. Der Entwickler und der Manager sind in der Folge als Spezialisierung dieser neuen Klasse „Mitarbeiter“ zu betrachten.

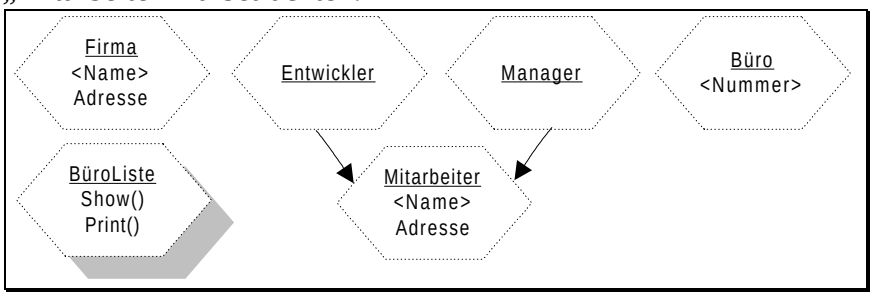
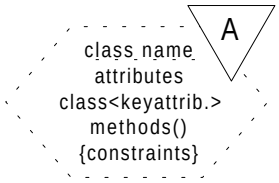


Abbildung 4-3: Schemadiagramm mit Vererbung

4.2.6 Abstrakte Klassen

Abstrakte Klassen sind dadurch gekennzeichnet, dass sie keine Instanzen besitzen dürfen. Üblicherweise werden derartige Klassen dort verwendet, wo für mehrere Klassen Übereinstimmungen gefunden wurden, die gemeinsame Basisklasse jedoch semantisch nicht sinnvoll instantiiert werden kann.

4.2.6.1 Konstrukt



Das Konstrukt der abstrakten Klasse ist in Booch’s Diagrammen gleichfalls anzutreffen und wird aus diesem Grund direkt übernommen. Die Ikone trägt im Bereich der rechten oberen Ecke ein nach unten zeigendes Dreieck, was sie von der Datenbankklasse abhebt, ansonsten ist die Darstellung analog.

4.2.6.2 Beispiel FIS

Von den bereits identifizierten Klassen ist die Instanzenbildung im Falle der Klasse „Mitarbeiter“ semantisch nicht sinnvoll. Damit wird diese zu einer abstrakten Klasse.

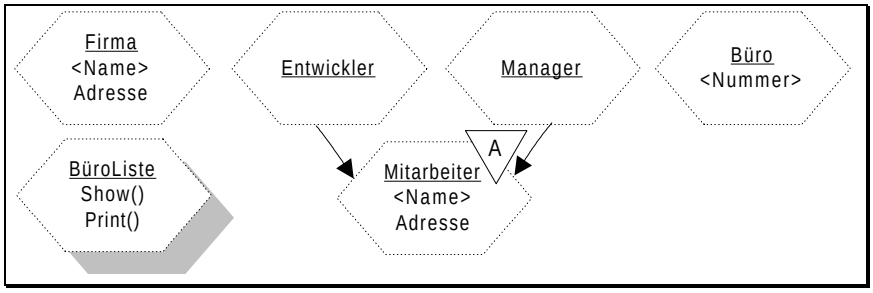


Abbildung 4-4: abstrakte Klassen im Schemadiagramm

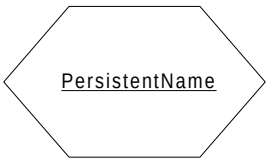
4.2.7 Objekte

Persistenzmechanismen

Für den Entwurf von Datenbankschemata ist es unerlässlich, sich mit den Persistenzmechanismen des Zielsystems auseinanderzusetzen. In Systemen, die „Persistenz durch Erreichbarkeit“ implementiert haben, muss für mindestens ein Objekt die Persistenz explizit erzwungen werden. Dieses Objekt wird so für das System der Ausgangspunkt der Persistenzfortpflanzung.

Aus diesem Grund muss der Notation ein Konstrukt zur Verfügung stehen um derartige Verankerungsobjekte bereits im Entwurf zu berücksichtigen.

4.2.7.1 Konstrukt



In Booch’s Methode werden in den Klassendiagrammen nur die Klassen und deren Abhängigkeiten betrachtet. Der Entwurf von Objekten als Instanzen von Klassen wird also streng von den Klassendiagrammen getrennt, sofern sie nicht selber als Klassen aufgefasst werden können, wie dies bei Metaklassen oder instantiierten Klassen von parametrisierbaren Klassen der Fall ist.

Im Gegensatz dazu lässt sich das Definieren von Objekten nicht aus dem Schemadiagramm verbannen, da es fester Bestandteil des Schemaentwurfes ist. Derartige Objekte werden mit Hilfe der von Booch vorgeschlagenen Objektikonen dargestellt.

Diese Objekte müssen einen systemweit eindeutigen Namen tragen. Auf die Angabe der Attribute und Methoden kann verzichtet werden, da die Objekte immer mit einem Instantiierungspfeil an eine bereits definierte Klasse angebunden sind.

4.2.8 Instantiierungsbeziehung

explizite Persistenz

Objekte sind Klasseninstanzen, deren Persistenz explizit erzwungen wird. Die Fähigkeit, persistente Instanzen ausbilden zu können, ist aber gemäss Definition einzig den Datenbankklassen vorbehalten. Aus diesem Grund muss ein Objekt mit einer Datenbankklasse in Relation stehen. Zudem muss der Entwerfer definieren können, von welcher Klassenzugehörigkeit das dauerhaft in der Datenbasis gespeicherte Objekt sein soll.

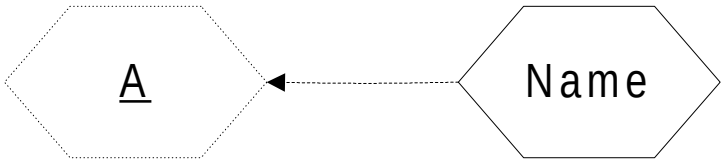
Durch die Erweiterung der Notation um das Konstrukt der Instantiierungsbeziehung können die gewünschten Eigenschaften der explizit persistenten Objekte modelliert werden.

4.2.8.1 Konstrukt

Diese Beziehung zwischen dem Objekt und der Klasse wird durch den Instantiierungspfeil dargestellt. Dieser Pfeil ist auch in der Methode von Booch enthalten, unterliegt jedoch dort einer völlig anderen Semantik, da er dort im Zusammenhang mit der Instantiierung von Metaklassen verwendet wird.

Ein Objekt muss als eine Instanz von genau einer Klasse aufgefasst werden. Es können durch dieses Konstrukt keine Hilfsklassen instantiiert werden.

4.2.8.2 Objekte und Instantiierung



Einstiegspunkt

Durch den gestrichelten Pfeil wird eine Instanz der Datenbankklasse A gebildet. Das Objekt wird Einstiegspunkt genannt und durch seinen Namen identifiziert.

Ein Einstiegspunkt wird als persistente Instanz einer Klasse aufgefasst. Die Klasse selbst wird dadurch persistenzfähig und kann Komponenten enthalten.

In den betrachteten Methoden wurde beanstandet, dass nicht zwischen persistenten und transienten Klassen unterschieden werden konnte. Mit der neuen Methode aber werden dem Entwerfer Konstrukte in die Hand gelegt, mit denen er diesen Unterschied während des Entwurfes mitberücksichtigen kann.

4.2.8.3 Einstiegspunkt

Persistenzfortpflanzung

Datenbankumgebungen, welche der „Persistenz durch Erreichbarkeit“ Folge leisten, verlangen für eine Schemadefinition mindestens einen persistenten Einstiegspunkt. Dieser muss eine Instanz einer persistenzfähigen Klasse sein und kann demnach als globales Objekt betrachtet werden. Die globale Instanz wird über den sogenannten persistenten Namen identifiziert.

Persistenzgraph

Ist ein solcher Einstiegspunkt einmal definiert, werden sämtliche von ihr referenzierten Instanzen gleichfalls persistent. Der Einstiegspunkt kann also bezüglich der Persistenzfortpflanzung als Wurzel des Persistenzgraphen betrachtet werden.

4.2.8.4 Einschränkungen

Damit die Semantik der Instantiierungsbeziehung nicht verletzt wird, müssen für dessen Verwendung einige einschränkende Regeln beachtet werden:

- Es dürfen nur Datenbankklassen instantiiert werden.
- Ein Einstiegspunkt muss durch genau einen Instantiierungspfeil an eine Datenbankklasse gebunden werden (ein Objekt kann nicht Instanz mehrerer Klassen sein).
- An eine Datenbankklasse dürfen mehrere Einstiegspunkte gebunden sein.
- Ein Schemadiagramm muss mindestens einen solchen Einstiegspunkt enthalten.
- Einstiegspunkte können nicht an anderen Beziehungen teilnehmen (die Beziehungseigenschaften werden von der instantiierten Klasse übernommen).
- Der Name muss für ein Diagramm eindeutig gewählt sein.

4.2.8.5 Beispiel FIS

Ein Schemadiagramm muss einen persistenten Einstiegspunkt besitzen. Im Beispiel FIS wird genau eine Firma namens „Macrohard“ betrachtet und es existiert eine entsprechende Datenbankklasse „Firma“. Somit wird sinnvollerweise ein explizit persistentes Objekt mit dem Namen „Macrohard“ ins Diagramm eingefügt und dieses mit einem Instantiierungspfeil an die Datenbankklasse „Firma“ gebunden.

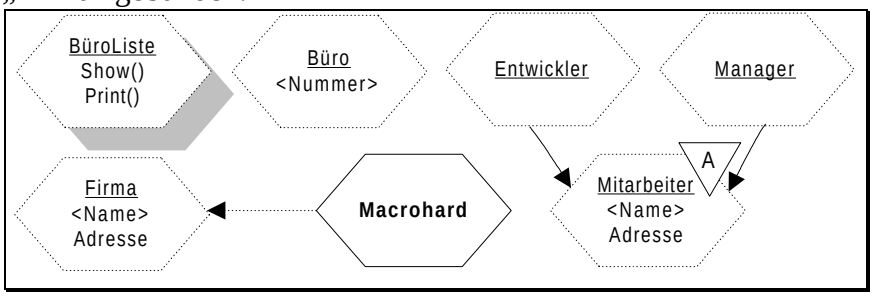


Abbildung 4-5: Schemadiagramm mit persistentem Einstiegspunkt

4.2.9 Komponentenbeziehung

Existenzabhängigkeit

Objektorientierte Datenbanksysteme unterstützen keine Aggregationsbeziehungen und können so nicht den Anforderungen der vollständigen Objektorientierung genügen. Für viele Anwendungen sind aber genau die Eigenschaften der Existenzabhängigkeit wünschenswert, da sie die Konsistenzsicherung bei der Erzeugung und Löschung von Objekten gewährleisten helfen.

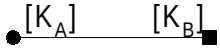
Komponentenbeziehung versus Referenz

Für den Entwerfer sind die implementationsspezifischen Einschränkungen der Zielsysteme irrelevant, solange er auf der konzeptuellen Ebene die Eigenschaften von Aggregationen abbilden kann. Dieser Sichtweise wird Rechnung getragen, indem während des Entwurfes zwischen der Komponentenbeziehung und der allgemeinen Referenz unterschieden werden kann.

Propagation

Zudem sollen im Entwurf bereits die Aspekte der Persistenzfortpflanzung und der Lösch-, bzw. Einfügepropagation mitberücksichtigt werden können. Im weiteren sollen zu jedem Zeitpunkt die Objektsammlungen durch die Definition von Extensionen zugänglich sein, um bestimmte Objekte auffinden oder um Einfügeeinschränkungen (Schlüssel) formulieren zu können.

4.2.9.1 Konstrukt



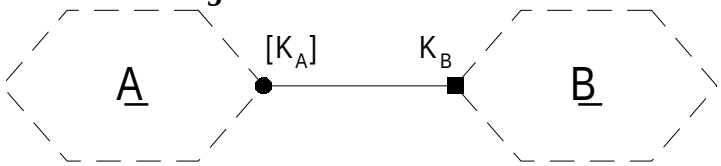
In Booch’s Methode ist die Relation „has-a by value“ vorgeschlagen. Diese wird mit einer kleinen Erweiterung übernommen.

Mit dieser Beziehung wird das physische Enthaltensein in einer anderen Klasse dargestellt (existenzabhängige Aggregation). Eine Klasse kann nicht in sich selber enthalten sein (keine Rekursion). Das Konstrukt beginnt bei der Aggregatsklasse mit einem ausgefüllten schwarzen Kreis und endet bei der Komponentenklasse mit einem ausgefüllten schwarzen Quadrat.

exklusive und nicht-exklusive Aggregation

Die Beziehung kann auf der Seite der Komponente eine Kardinalität tragen, welche die Mengeneinschränkung angibt, während auf der gegenüberliegenden Seite die Kardinalität zu interpretieren ist, als exklusive Aggregation, im Falle 0, 1, C, oder nicht-exklusive Aggregation, wenn sie grösser als 1 ist. Wird die Kardinalitätsangabe auf der Seite des Kreises gänzlich weggelassen, wird damit eine exklusive Aggregation ausgedrückt. Das Weglassen der Kardinalität K_B hingegen wird mit der Angabe 0+ gleichgestellt.

4.2.9.2 Verwendung



Eine Komponentenbeziehung wirkt zwischen einer Klasse A, die als Besitzer der Beziehung definiert wird, und der Klasse B, die als Teilnehmer bezeichnet wird. Auf der Seite des Besitzers steht der ausgefüllte schwarze Kreis und auf der Seite des Teilnehmers das ausgefüllte schwarze Quadrat.
Die Komponentenkardinalität K_B definiert die Anzahl der Komponenten, die eine Instanz der Klasse A vom Typ B haben kann oder muss. Diese wird durch eine Minimalangabe, welche beziffert werden muss, und eine Maximalangabe, welche offen gelassen werden darf, definiert.
Der Besitzer der Komponentenbeziehung wird Aggregationsklasse genannt (A), während der Teilnehmer Komponente heisst (B).

Komponentengraph

Die Komponentenbeziehung kann als eine gerichtete Kante im Komponentengraphen definiert werden. In diesem Graphen werden als Knoten nur die Klassen gemäss obiger Definition und die Komponentenbeziehung als gerichtete Kanten betrachtet. Aus der Forderung, dass eine Instanz einer Klasse nicht in mehreren Aggregaten vorkommen darf, lässt sich ableiten, dass es sich bei diesem Graphen sogar um einen allgemeinen Baum handeln muss. Werden die persistenten Einstiegspunkte als Komponenten einer virtuellen Wurzel definiert und die Instantiierungsbeziehungen als Kanten in den Baum übernommen, muss er sogar zusammenhängend sein.

4.2.9.3 Semantik

Fortpflanzung der Persistenz entlang der Komponenten-beziehungen

Ist die Datenbankklasse A persistenzfähig, so überträgt sie diese Eigenschaft auf die Datenbankklasse B. Da alle Datenbankklassen als Komponenten von bereits persistenzfähigen Klassen definiert sind, kann jede Klasse als Bestandteil eines Einstiegspunktes angesehen werden. Auf diese Weise entstehen entweder ein Komponentenbaum oder mehrere disjunkte Komponentenbäume. Der im vorigen Abschnitt erwähnte Persistenzgraph kann auf diese Komponentenbäume abgebildet werden, woraus sich für das Schema die Eigenschaft ergibt, dass sich die Persistenz entlang der Komponentenbeziehungen fortpflanzt.

Ist die Beziehung invers definiert, so kennt die Komponentenklasse ihre Aggregationsklasse, d.h. mit einer Referenz auf die Klasse B kann die Referenz auf die Klasse A beschafft werden.

Extension und Schlüsselbeziehungen

Hat die Beziehung bei B eine Kardinalität grösser als 1, so bedeutet das, dass die Klasse A mehr als eine Komponente vom Typ B enthält. Die Instanz der Klasse A enthält dann entsprechend die Extension der Komponenten vom Typ B, über welche z.B. ein bestimmtes Exemplar ausfindig gemacht werden kann. Zudem können beim Einfügen die Schlüsselbeziehungen überprüft werden.

Existenzabhängigkeit

Mit der Komponentenbeziehung wird eine Existenzabhängigkeit dargestellt. Soll also eine Beziehung zwischen einer Instanz von A zu einer Instanz von B aufgebaut (abgebaut) werden, muss die Instanz neu kreiert (gelöscht) werden. Die Instanzen der Aggregationsklasse üben also die Kontrolle über ihre Komponenteninstanzen aus, d.h. dass Komponenten nur vom Besitzer erzeugt oder gelöscht werden können.

Daraus lässt sich ableiten, dass, wenn der Entwerfer wünscht, eine Instanz B_m der Klasse B direkt zu löschen, etwa mit der Anweisung
delete Bm
so muss er dies über einen Umweg erledigen. Zunächst muss die Komponentenbeziehung zwischen A und B invers definiert sein, damit von B aus seine Kontrollklasse aufgespürt werden kann. Dann kann er über diese Referenz der Instanz von A die Nachricht
Bm->refA->RemoveB(Bm)
schicken. Somit wird die Aggregationsklasse, die bezüglich der Löschung von B-Instanzen monopolistisch agiert, angewiesen, die Instanz B_m aus der Datenbasis zu entfernen.

4.2.9.4 Extensionen

lokale Extension

Eine Extension ist gemäss Definition die Menge aller zu einem gewissen Zeitpunkt existenten persistenten Instanzen einer Klasse C. Der Begriff wird für die Verwendung innerhalb der Methode auf den Begriff der globalen Extension ausgeweitet. Zusätzlich muss der Begriff der lokalen Extension eingeführt werden. Wird nur der Begriff „Extension“ verwendet, gilt das Ausgesagte sowohl für den einen wie auch für den andern Typ.

Stelle man sich eine Klasse A, bestehend aus 1+ Komponenten der Klasse B, und jedes B wiederum als Aggregation von 1+ Komponenten der Klasse C, vor. Die globale Extension der Klasse C, also die Gesamtsammlung der Instanzen C_i. liegt in keiner der Klassen explizit vor. Sie kann aber aus der Vereinigung der Teilextensionen oder der lokalen Extensionen, die in den Instanzen B_j enthalten sind, gewonnen werden.

Aggregationsklassen als Extensionen

Es muss also kein spezielles Konstrukt für die Modellierung von Extensionen in die Notation aufgenommen werden. Vielmehr übernehmen die Aggregationsklassen die Rollen der Instanzsammlungen. Dank der Forderung nach Komponentenbäumen ist jede Instanz einer Aggregationsklasse also eine lokale Extension ihrer Komponenteninstanzen und in der umgekehrten Richtung ist jede Extension, global oder lokal, trivialerweise die Aggregation der gesammelten Objekte. Aus diesem Umstand lassen sich für die Extensionen verschiedene Eigenschaften ablesen:

- Eine Instanz gehört zu genau einer Extension.
- Der Durchschnitt aller lokalen Extensionen ist leer. Die Vereinigung aller lokalen Extensionen entspricht der globalen Extension. Die Vereinigung aller globalen Extensionen deckt sich bijektiv mit allen in der Datenbasis enthaltenen Objekten.
- Die Extension unterliegt denjenigen mengenwertigen Einschränkungen, die durch die Kardinalitätsbedingung der Komponentenbeziehung definiert sind.
- Eine Instanz ist immer eine (direkte oder indirekte) Komponente eines Einstiegspunktes. Deshalb lässt sich über ihn immer die globale Extension zusammenstellen.

Eine Extension nimmt also keine spezielle Rolle innerhalb des Entwurfes oder der Implementation wahr, die Aggregationstypen werden vielmehr um die Eigenschaften der Extensionen angereichert. Eine Extension muss die Fähigkeit besitzen, ein Objekt in sich aufzunehmen, eines zu entfernen und ein sich entweder durch bestimmte Attributsausprägungen oder durch seine Identität auszeichnendes Objekt aufzufinden. Die ersten beiden Fähigkeiten sind für alle Datenbankklassen durch die Eigenschaften, die sie aus der Notwendigkeit der Fortpflanzungunterstützung geerbt haben, bereits enthalten, während die dritte den derartigen Klassen noch hinzuzufügen ist.

4.2.9.5 Schlüssel

Die Methode bietet dank der Erweiterung des Klassenkonstruktes die Definitionsmöglichkeit von Schlüsselattributen, aus denen der Schlüssel gewonnen

werden kann. Schlüssel sind Tupel von Attributen einer Klasse und dienen den folgenden Zwecken:
1. Sie sollen die Eindeutigkeit innerhalb einer Extension bezüglich der Schlüsselattributwerte gewährleisten.
2. Sie sollen Kriterien für das wertbasierte Auffinden von bestimmten Objekten definieren helfen.
3. Sie sollen Sortierungen und Gruppierungen ermöglichen.
4. Sie sollen Kriterien für leistungssteigernde Massnahmen wie Indexierung liefern.

Ein Schlüssel ist eine Eigenschaft der Aggregationsbeziehung. Sind die Instanzen der Klasse B Komponenten der Klasse A und sollen diese bezüglich eines bestimmten Attributes eindeutig sein, wird diese Einschränkung der Klasse A in der Form
B<Key>
einbeschrieben. Die Definition von Schlüsseltupeln hat also lediglich Einfluss auf die Instanzen der Extensionsklasse. Insbesondere wirkt sie auf die Methoden für das Einfügen eines neuen Objektes und auf das Auffinden eines bestimmten Objektes, das durch seine Schlüsselattributsausprägungen somit identifizierbar gemacht wurde.

Da wie im vorigen Abschnitt ausgeführt, zwischen globalen und lokalen Extensionen unterschieden werden muss, wird beim Einfügen entsprechend nur auf lokale Eindeutigkeit geprüft. Somit bleibt es unter der Aufsicht des Entwerfers, ob die Schlüssel global oder lokal eindeutig sind.

4.2.9.6 Einschränkungen

Damit die Semantik der Komponentenbeziehung nicht verletzt wird, müssen für dessen Verwendung einige einschränkende Regeln beachtet werden:

- Die Klasse B kann nur in genau einer Klasse enthalten sein.
- Alle Klassen müssen entweder als Komponenten von anderen Klassen definiert werden oder durch Objekte instantiiert sein.

4.2.9.7 Beispiel FIS

Die Klasse „Firma“ ist bereits durch ihre Instantiierungsbeziehung vom Objekt „Macrohard“ aus persistent gemacht worden. Damit sich die Persistenz auch auf die anderen Instanzen der Klassen „Entwickler“, „Manager“ und „Büro“ auswirkt, müssen sie als Komponenten des Einstiegspunktes definiert werden. Die Instanzen der Klasse „Büroliste“ dürfen aber nicht persistent werden, weshalb keine Komponentenbeziehung zu ihr aufgebaut werden darf.

Die Entwickler, die Manager und die Büros sollen als Bestandteil der Firma aufgefasst werden. Damit lassen sich die Komponentenbeziehungen formulieren.
Dies sei im Detail am Beispiel des Entwicklers ausgeführt. Die Firma beschäftigt Entwickler, also muss zwischen der Klasse „Firma“ als Besitzer und der Klasse „Entwickler“ als Teilnehmer eine Komponentenbeziehung existieren. In der Firma können beliebig viele Entwickler arbeiten, insbesondere auch keiner, was durch die Kardinalität 0+ auf der Seite des Entwicklers verdeutlicht wird. Durch die Auffassung des Entwicklers als Komponente der Firma zieht die Erweiterung der Eigenschaften der Klasse „Firma“ um die Fähigkeit Entwickler anzustellen (Beziehungsaufbau) und zu entlassen (Beziehungsabbau) nach sich. Zudem ist die Firma - oder genauer das Objekt „Macrohard“ - als einzige in der Lage, die Menge aller angestellten Entwickler zu kennen.
Auf analoge Art werden die entsprechenden Komponentenbeziehungen zwischen der Klasse „Firma“ und den Klassen „Manager“ und „Büro“ eingezeichnet.

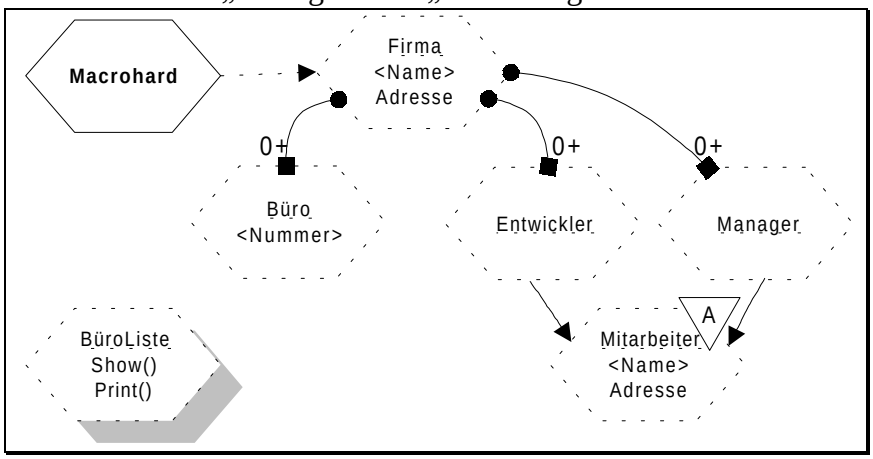


Abbildung 4-6: Schemadiagramm mit Komponentenbeziehungen

Die Instanzen der Klassen „Entwickler“ und „Manager“ werden gleichsam als Komponenten des Objektes „Macrohard“ aufgefasst. Dabei handelt es sich um eine gemeinsame Eigenschaft dieser beiden Klassen. Ohne Änderung der Semantik hätte deshalb die Komponentenbeziehung auch zu der Klasse „Mitarbeiter“ definiert werden können.

4.2.10 Inverse Beziehung

Während mit den Komponentenbeziehungen Existenzabhängigkeiten modelliert werden, muss dem Entwerfer ein Konstrukt für die Formulierung von allgemeinen Beziehungen zur Verfügung gestellt werden. Trotzdem soll die bereits erreichte Sicherheit bei der Erzeugung und Löschung von Datenobjekten nicht aufgeweicht werden, weshalb in der Notation keine einseitigen Beziehungen zugelassen sind.

ungewollte Persistenz durch einseitige Beziehungen

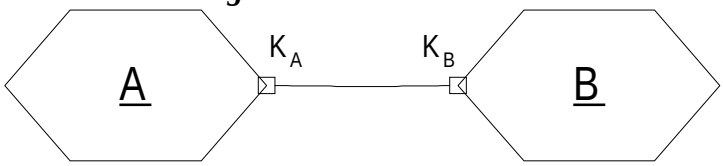
Durch die Verwendung von einseitigen Beziehungen könnten nämlich Objekte in der Datenbasis durch andere referenziert sein, ohne dass sie etwas davon wissen. Bei der Löschung eines derartigen Objektes zum Beispiel wäre der Entwickler gefordert, auf umständliche Weise die nun nicht mehr gültige Referenz zu löschen. Dies liesse den Implementationsaufwand unverhältnismässig stark steigen und wäre darüber hinaus sehr fehleranfällig. Zudem werden Datenbanksysteme, welchen die Philosophie der „Persistenz durch Erreichbarkeit“ zu Grunde liegt, durch „hängende“ Beziehungen referenzierter Objekte nicht aus ihrer Datenbasis entfernt, was leicht zu gefährlichen Inkonsistenzen führen könnte.

4.2.10.1 Konstrukt

Bei Booch sind innerhalb der Klassendiagramme keine echten inversen Beziehungen vorgesehen. Sind Kardinalitäten auf beiden Seiten einer „has-a by reference“ Beziehung angegeben, obliegt die Interpretation dieses Sachverhalts dem Ermessensspielraum des Lesers.

Aus diesem Grund wird ein neues Konstrukt für die Modellierung von inversen Beziehung eingeführt. Dieses trägt an beiden Enden ein nicht ausgefülltes Quadrat. Auf beiden Seiten müssen entsprechend Kardinalitäten angegeben werden.

4.2.10.2 Verwendung



Diese inverse Beziehung kann zwischen zwei beliebigen Klassen gleichen Klassentyps wirken. Heterogene Verbindungen sind nur dann zulässig, wenn es sich bei der einen Klasse um eine abstrakte handelt. Sie ist symmetrisch und unterscheidet somit semantisch nicht zwischen den beiden teilnehmenden Klassen. Insbesondere kann sie von beiden beteiligten Klassen gleichermassen auf- und abgebaut werden.

Die Kardinalitäten können beliebig gewählt werden. Dabei unterliegen die beiden Kardinalitäten K_A und K_B keinen Abhängigkeiten oder Einschränkungen.

Selbstreferenzen

Die inverse Beziehung dient der Modellierung von allgemeinen Relationen und wird daher nur dann verwendet, wenn sich semantische Beziehungen nicht mit den anderen Verbindungstypen darstellen lassen. Mit diesem Konstrukt können vor allem auch Selbstreferenzen dargestellt werden.

4.2.10.3 Einschränkungen

Damit die Semantik der inversen Beziehung nicht verletzt wird, müssen für deren Verwendung einige einschränkende Regeln beachtet werden:

- Eine Klasse kann an beliebig vielen solchen Beziehungen teilnehmen.
- Inverse Beziehungen können nicht zwischen Datenbank- und Hilfsklassen wirken.

4.2.10.4 Beispiel FIS

Gemäss Anforderungskatalog wird einem Mitarbeiter jeweils ein Büro zugeordnet. Aus diesem Grund muss zwischen den beiden Klassen eine Beziehung existieren. Ein Mitarbeiter darf aber nicht als Komponente eines Büros aufgefasst werden, da sonst ein Mitarbeiter als Bestandteil zweier verschiedener Objekte auftreten würde. Also muss zwischen diesen beiden Klassen eine inverse Beziehung eingerichtet werden. Die Beziehung unterliegt der Bedingung, dass ein Mitarbeiter immer ein Büro haben muss und dass ein Büro von beliebig vielen Mitarbeitern genutzt werden kann, insbesondere auch von keinem. Daraus lassen sich die Kardinalitäten N* auf der Seite des Mitarbeiters und 1 auf der gegenüberliegenden Seite der Beziehung ableiten.
Analog muss eine Beziehung zwischen dem Manager und dem Entwickler erstellt werden, da sie, wie aus der Liste der Anforderungen zu entnehmen ist, zusammen und in Teams organisiert an einem Projekt arbeiten. Die Teambeziehung ist dabei genau dann konsistent, wenn genau ein Manager die Koordinationsrolle übernimmt und mindestens ein Entwickler mit der Bearbeitung beschäftigt ist.

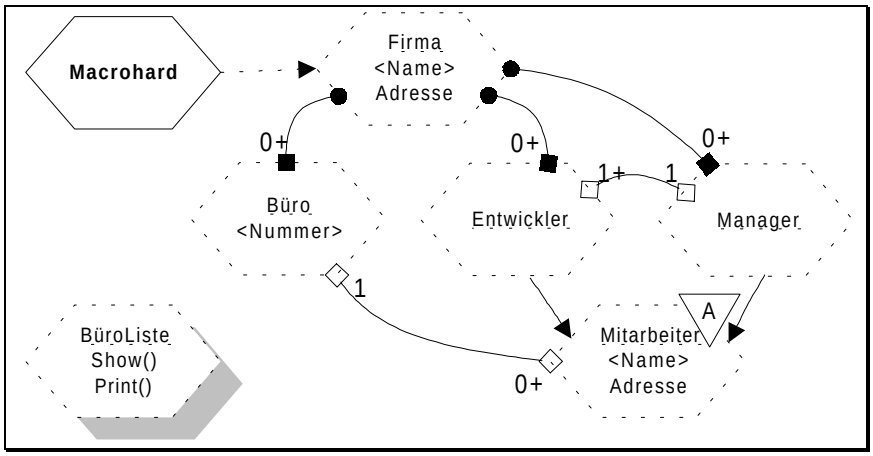


Abbildung 4-7: Schemadiagramm mit inversen Beziehungen

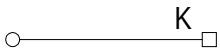
4.2.11 Beobachtungsbeziehung

Verbindung von transienten und persistenten Objekten

Komponentenbeziehungen und inverse Beziehungen verbinden die persistenten bzw. transienten Objekte untereinander. Naheliegenderweise muss in der Notation auch ein Konstrukt angeboten werden, welches die Lücke zwischen diesen beiden Welten zu überbrücken weiss. Derartige Relationen dürfen sich aber nicht nachteilig auf die Sicherheit bezüglich der Persistenzfortpflanzung auswirken, weshalb sie nur von transienten Objekten aus auf Instanzen der Datenbankklasse zeigen dürfen.

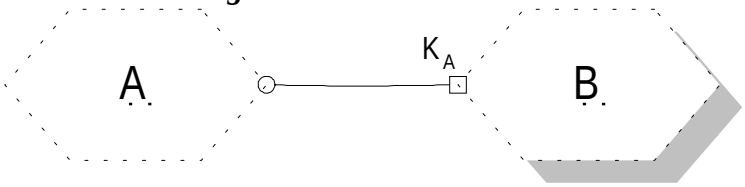
Der Entwerfer muss sich einzig vergegenwärtigen, dass die Propagation die Referenzsicherheit entlang den Beobachtungsbeziehungen nicht gewährleisten kann. Die Algorithmen müssen dieser Problematik also durch die Implementation von speziellen Behandlungen oder Formulierung von expliziten Konsistenzbedingungen Herr werden.

4.2.11.1 Konstrukt



Der in Booch's Methode eingeführte Beziehungenstyp „has-a by reference" beschreibt eine einseitige beobachtende Relation von einer Klasse zu einer anderen. Diese Idee soll für die Beobachtungsbeziehung übernommen werden. Auch die von ihm vorgeschlagene „using"-Beziehung hat gewisse Aspekte der neuen Beziehung gemein. Das Symbol wird daher als Kombination dieser beiden Verbinder gezeichnet und in die Notation aufgenommen.

4.2.11.2 Verwendung



Diese Beobachtungsbeziehung verbindet die Hilfsklassen mit den Datenbankklassen. Der Besitzer der Beziehung muss die Hilfsklasse sein. Entsprechend muss der Teilnehmer eine Datenbankklasse sein. Das Besitzen der Beziehung wird durch einen nicht ausgefüllten Kreis verdeutlicht, die Teilnahme entsprechend mit einem nicht ausgefüllten Quadrat.
Mit diesem Konstrukt wird einer Hilfsklasse die Fähigkeit gegeben, eine Datenbankklasse zu „beobachten". Somit kann sie als eine Art von Sicht innerhalb des Systems arbeiten. Durch die Angabe einer Kardinalität auf der Seite des Teilnehmers kann die Anzahl der zu beobachtenden Instanzen definiert werden. Diese Kardinalität kann beliebig gewählt werden.

Keine Fortplanzung der Persistenz entlang der Beobachtungs-beziehungen

Die Persistenz kann sich für Datenbanksysteme, die sich dem Grundsatz der Persistenz durch Erreichbarkeit verschrieben haben, entlang jeder Beziehung fortpflanzen. Dieser Effekt darf sich aber auf keinen Fall auf die Hilfsklassen auswirken, weshalb immer die Hilfsklasse der Besitzer einer derartigen Beziehung sein muss. Daraus folgt im weiteren, dass die Beziehung einseitig sein muss und nicht invers erweitert werden kann.

Sicherheit der Referenz

Da die Beobachtungsbeziehung einseitig ist, kann bei einer Veränderung der beobachteten Instanzenmenge die Sicherheit der Referenz nicht gewährleistet werden. Wird beispielsweise ein beobachtetes Objekt gelöscht, hat dieses keine Möglichkeit, dem Beobachter seine Löschung mitzuteilen. Analog kann bei der Erzeugung einer neuen Instanz nicht ohne weiteres geprüft werden, ob das neue Objekt das Beobachtungskriterium erfüllt und deshalb in die Referenzmenge der Hilfsklasseninstanz aufgenommen werden müsste.

Validierung der Referenzmenge

Aus diesem Grund muss bei jeder Verwendung der Referenzmenge gewährleistet sein, dass sich die beobachtete Menge nicht verändert hat. Da dies aber mit einigem, gefährlich komplexen Programmieraufwand verbunden ist, kann auch die Strategie gewählt werden, welche die Menge vor jedem Verwenden validiert oder neu aufbaut. Das Konstrukt wird darum Beobachtungsbeziehung genannt.

4.2.11.3 Einschränkungen

Damit die Semantik der Beobachtungsbeziehung nicht verletzt wird, müssen für dessen Verwendung einige einschränkende Regeln beachtet werden:

- Die Beziehung muss von einer Hilfsklasse aus zu einer Klasse als Teilnehmer führen. Sie kann nicht zwischen gleichartigen Klassen wirken und auch nicht von einer Klasse aus auf sich selber.
- Eine Datenbankklasse kann an beliebig vielen solchen Beziehungen teilnehmen.
- Eine Hilfsklasse kann beliebig viele solcher Beziehungen besitzen.

4.2.11.4 Beispiel FIS

Nachdem die Datenbankklassen allesamt persistent gemacht worden sind, darf sich dieser Effekt nicht auf die Instanzen der Klasse „Büroliste“ ausweiten. Aus diesem Grund dürfen Bürolistenobjekte nicht über eine inverse Beziehung mit den zu beobachtenden Instanzen der Klasse „Büro“ verbunden werden. Auch die Verwendung der Komponentenbeziehung ist an dieser Stelle nicht zulässig, da die überwachte Klasse bereits als Komponente einer anderen Klasse deklariert worden ist. Vielmehr ist hier die Beobachtungsbeziehung zu verwenden.
Die Büroliste beobachtet eine beliebige Anzahl von Büros, weshalb auf der Seite der Büroklasse die Kardinalität 0+ hizuzufügen ist. Die Angabe einer Kardinalität auf der anderen Seite ist sinnlos, da diese Beziehung immer einseitig ist.

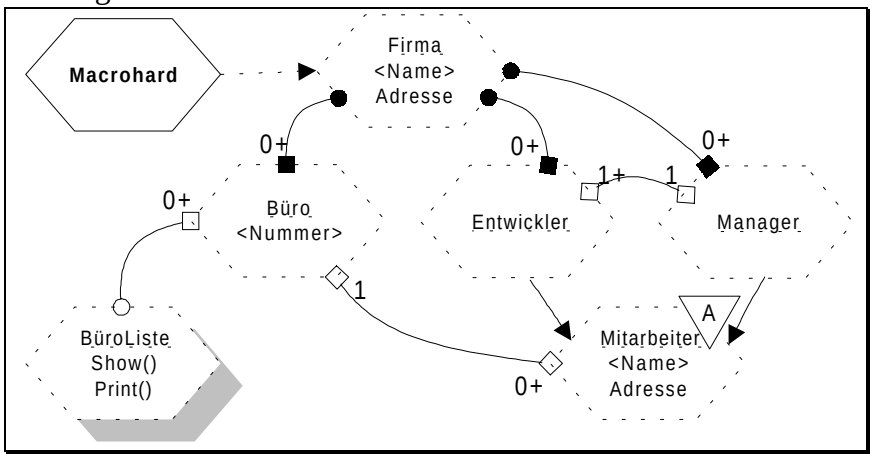


Abbildung 4-8: Schemadiagramm mit Beobachtungsbeziehung

4.2.12 Objektevolution

Objektevolution als Lösch- und Erzeugungsprozess

Typischerweise werden in Datenbanken Informationen gespeichert, die während einer sehr langen Lebensdauer zugreifbar sind. Ebenso typisch treten während dieser Lebenszeit Ereignisse ein, die bei einem betrachteten Objekt semantisch eine Evolution initiieren. Objektorientierte Datenbanksysteme sind aber im Allgemeinen nicht auf derartige Mutationsprozesse vorbereitet, weshalb dieses unverzichtbare Verhalten durch einen Lösch- und Erzeugungsprozess nachgebildet werden muss.

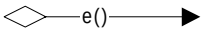
Erhaltung der Daten und Beziehungen

Die Evolution unterscheidet sich dabei von einer gemeinen Löschung, bzw. Erzeugung dadurch, dass gewisse bereits gespeicherte Daten erhalten bleiben sollen. Zudem sollen etwaige Beziehungen, die semantisch auch nach der Mutation noch Sinn ergeben, diesen Übergang sicher überleben.

statische Eigenschaft versus dynamisches Verhalten

Mit der Objektevolution wird ein dynamisches Verhalten von Objekten beschrieben. Das Ereignis, welches eine derartige Mutation provoziert, die einschränkenden Regeln, usw. müssen dementsprechend in den dynamischen Diagrammen wie z.B. Zustandsübergangsdiagramme spezifiziert werden. Die Fähigkeit zur Objektevolution hingegen ist eine statische Eigenschaft der Datenbankklassen und hat aus diesem Grund hier durchaus seine Berechtigung.

4.2.12.1 Konstrukt

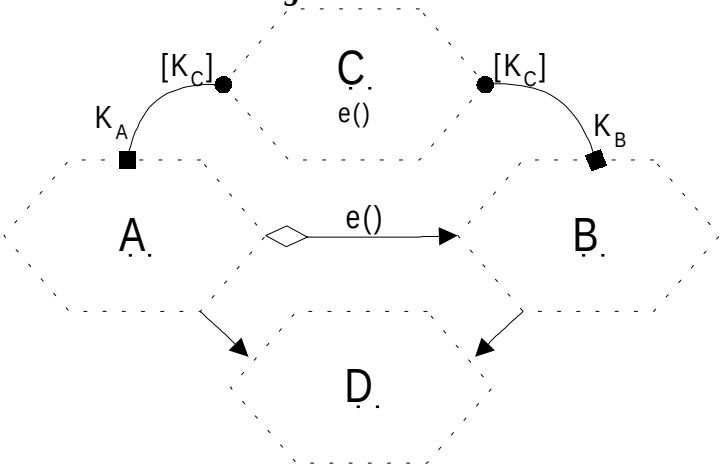


In der Methode von Booch ist das Wechseln der Klassenzugehörigkeit von Instanzen nicht spezifizierbar. Also muss ein neues Konstrukt eingeführt werden, um den möglichen Wechsel einer Instanz von einer Klasse in eine andere zu beschreiben.

Das Konstrukt wird als Pfeil dargestellt, der auf der Seite der Ausgangsklasse eine Raute als Symbol trägt und mit seiner Spitze auf die Zielklasse zeigt. Die Evolution muss einen Namen tragen.

Mit diesem Konstrukt wird die Eigenschaft der Instanzen einer Klasse modelliert, ihre Klassenzugehörigkeit zu ändern. Zudem wird angegeben, zu welcher Klasse die Instanz mutieren kann, wer diesen Prozess kontrolliert und welche Daten bei der Mutation erhalten bleiben.

4.2.12.2 Die Verwendung



Die Objektevolution definiert die Eigenschaft der Instanzen der Klasse A, ihre Klassenzugehörigkeit nach B wechseln zu können. Die Beziehung ist gerichtet, d.h. durch einen Pfeil von A nach B wird lediglich die Fähigkeit ausgedrückt, von A nach B migrieren zu können. Soll dies auch in der anderen Richtung möglich sein, muss dies durch einen zweiten Pfeil verdeutlicht werden.

Besitzer der Objektevolution

Die Objektevolution kann zwischen zwei beliebigen Komponenten (unterschiedlicher Klassen) eines Objektes wirken. Dieses wird der Besitzer der Objektevolution genannt.
Der Evolutionspfeil muss einen Namen tragen. Die Evolutionsfunktion wird innerhalb der Besitzerklasse als Methode implementiert und trägt den Namen des Evolutionspfeils. Aus diesem Grund muss der Name der Evolution innerhalb eines Besitzers eindeutig sein.

Ursprungsklasse und Zielklasse

Die Klasse, deren Instanzen evolutionsfähig sind, wird „Ursprungsklasse“ genannt, während die Klasse, welche Instanzen durch Objektevolution hinzugewinnen kann, den Namen „Zielklasse“ trägt. Zudem heisse die Superklasse der Ursprungs- und Zielklasse im Folgenden Basisklasse.

Basisklasse

Objektevolution macht vor allem dann Sinn, wenn gewisse Daten der Ursprungsklasse in die Zielklasse übernommen werden können. Sind die beiden Klassen Derivate derselben Klasse oder ist die eine Klasse eine Spezialisierung, bzw. Generalisierung der anderen Klasse, so sollen natürlich die Daten der gemeinsamen Basis erhalten bleiben. Sämtliche Eigenschaften, die lediglich die Ursprungsklasse aufweist, werden verworfen, während die Eigenschaften, welche nur die Zielklasse besitzt, neu erstellt werden.

4.2.12.3 Semantik

Die Migration einer Instanz A_i zu einer Instanz B_j bedeutet, dass zuerst ein neues Objekt B_j erzeugt wird, die gemeinsamen Daten von der alten auf die neue Instanz kopiert werden und zuletzt die Instanz A_i gelöscht wird. Aus dieser Auffassung von Objektevolution lässt sich ableiten, dass diese Migration nur vom Besitzer der beiden Instanzen A_i und B_j, nennen wir ihn C_k durchgeführt werden kann, da dies das einzige Objekt ist, welches die Fähigkeit des Erzeugens von B_j und die des Löschens von A_i auf sich vereinen kann. Daraus folgt, dass die Durchführung der Migration von C_k aus vorgenommen werden muss, somit im Allgemeinen eine Eigenschaft der Klasse C wird und entsprechend als Klassenmethode implementiert wird. Zudem lässt sich daraus ableiten, dass die Objektevolution die Komponentenbeziehung nicht verändert, d.h. das Objekt B_j kann nicht zu einer Komponente von C_l (mit $l \neq k$) mutieren.

Abbildungsvorschrift

Sollen bei diesem Verwandlungsprozess Daten aus der zu löschenden Instanz auf die neu generierte Instanz übertragen werden, stellt sich die Frage nach der Abbildung der Daten vom Original- auf das Zielobjekt. Sind die beiden Objekte Instanzen von Klassen, die beide ein Derivat (beliebiger Tiefe) einer Vaterklasse D sind, so lässt sich diese Abbildungsvorschrift intuitiv definieren als die Übernahme der vom Vater her stammenden gemeinsamen Attribute und der Verwerfung, bzw. Neukreation der differierenden Eigenschaften. Ist diese Klasseneigenschaft jedoch nicht gegeben, so muss sich der Entwickler selber um die passende Abbildung kümmern. Sollen beim Wechsel keine Daten übernommen werden, so stellt sich berechtigterweise die Frage nach dem Sinn der Objektevolution in diesem Kontext.

Die Funktionalität der Datenübernahme kann sowohl als Eigenschaft der Klasse A als auch der Klassen B, C oder D betrachtet werden und wird gleichfalls als Klassenmethode implementiert.

4.2.12.4 Einschränkungen

Damit die Semantik der Objektevolution nicht verletzt wird, müssen für deren Verwendung einige einschränkende Regeln beachtet werden:

- Eine Klasse kann an beliebig vielen solchen Beziehungen teilnehmen. Zwischen der Klassen A und B kann aber höchstens eine Beziehung mit A als Ursprungs- und B als Zielklasse definiert werden.
- Objektevolutionen können nur zwischen verschiedenen Komponenten einer Besitzerklasse gezeichnet werden. Daraus folgt, dass keine Evolutionsbeziehungen zwischen Hilfs- und Datenbankklassen unterhalten werden können.
- Objektevolution kann nicht zwischen zwei Objekten gleicher Klasse wirken.
- Es empfiehlt sich, die Objektevolution vor allem zwischen Ursprungs- und Zielklassen wirken zu lassen, die eine gemeinsame Basisklasse besitzen. Dies ist erfüllt, wenn die beiden Klassen von derselben Klasse abgeleitet sind oder die eine Klasse eine Spezialisierung der anderen Klasse ist.

4.2.12.5 Beispiel FIS

Nur im Fall der Beförderung wird von den Anforderungen verlangt, dass ein Objekt seine Klassenzugehörigkeit ändert. Instanzen der Klasse „Entwickler“ sollen die Eigenschaften erhalten, zu Instanzen der Klasse „Manager“ migrieren zu können.

Als Voraussetzungen zur Verwendung der Objektevolution wird verlangt, dass die Ursprungs- und die Zielklasse Komponenten derselben Klasse sind. Dies ist dank der geeigneten Wahl der Komponentenbeziehungen erfüllt. Im Weiteren ist durch die gemeinsame Basisklasse „Mitarbeiter“ eine Erhaltung der Mitarbeiterdaten,

insbesondere aber auch die Bürozuordnung, beim Übergang gewährleistet.
Damit lässt sich die Evolutionsbeziehung vom Entwickler zum Manager einrichten. Als Name der Beziehung wird „befördern“ gewählt, was für die Basisklasse bedeutet, dass sie eine neue Methode desselben Namens erhalten muss.

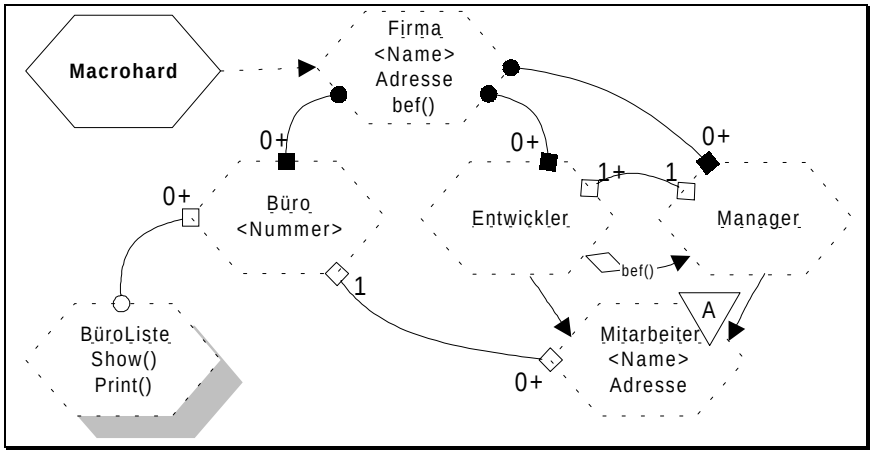
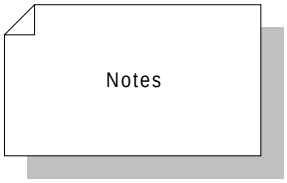


Abbildung 4-9: Schemadiagramm mit Objektevolution

4.2.13 Notizen



Analog zu Booch sind erweiterte Informationen, die zum Verständnis des Schemaentwurfes beitragen in Notizfelder formulierbar. Dieses Konstrukt wird direkt von Booch übernommen.

4.2.13.1 Notizbezug

Durch die Verbindung eines Kommentarelementes mit einem beliebigen anderen Element des Diagramms mittels einer gestrichelten Linie kann der Bezug auf einen entworfenen Sachverhalt hergestellt werden.
Notizen haben semantisch keine Bedeutung, sie tragen lediglich zum besseren Verständnis des Diagramms bei. Aus diesem Grund sind für dessen Verwendung keine Einschränkungen zu berücksichtigen.

4.2.13.2 Beispiel FIS

Das Schemadiagramm für das Beispiel FIS kann nun an dieser Stelle durch Notizen bereichert werden.

4.2.14 Zusammenfassung Beispiel FIS

Das Schemadiagramm wurde in den vorherigen Abschnitten Schritt für Schritt aufgebaut. Nach diesem Raster vorzugehen, ist empfehlenswert aber nicht zwingend. Die Entwurfsabfolge kann auch variieren, da eventuell nach einer „top-down“ oder „bottom-up“ Strategie vorgegangen wird und zu einem gewissen Zeitpunkt nicht alle Abstraktionsebenen vollumfänglich überschaut werden können. Häufig werden erst während des Designs zusätzliche Informationsbedürfnisse erkannt.

Zudem unterliegt der Entwurf immer einer gewissen Willkür. Für die Modellierung könnten auch andere Klassen und Beziehungen identifiziert worden sein, die derselben Semantik genügen.
Gemäss dem Anforderungskatalog fehlen den Klassen im Diagramm noch einige Eigenschaften und Fähigkeiten. Diese sind, falls sie nicht durch die impliziten Klassenmethoden abgedeckt sind, der Klassendefinition hinzuzufügen. Implizite Methoden sind Fähigkeiten, die eine Klasse erhält, indem für sie Abhängigkeiten zu anderen Klassen eingerichtet worden sind.
Beispielsweise soll die Klasse „Firma“ Entwickler anstellen und entlassen können. Diese Fähigkeiten können im Sinne des Schemas als Auf- bzw. Abbau der Komponentenbeziehung zu der Klasse „Entwickler“ interpretiert werden. Derartige Methoden gelangen implizit in die Klasse „Firma“, weshalb sie nicht durch die Definition von dedizierten Funktionen beigefügt werden müssen.
Genauso verhält es sich beispielsweise für die Bürozuordnung der Mitarbeiter, welche vom Auftraggeber als Anforderung formuliert worden ist. Sie kann analog als Auf- und Abbaufunktion der Beziehung zwischen den Klassen „Büro“ und „Mitarbeiter“ betrachtet werden und ist somit bereits implizit vorhanden.
Demnach fehlen zu der Erfüllung der geforderten Funktionalität einzig die lesenden Zugriffsfunktionen für den Aufbau der Mitarbeiterlisten und Projektteammitgliederlisten. Es obliegt nun dem Entwerfer zu entscheiden, an welcher Stelle diese Funktionen eingebaut werden sollen. Auf der einen Seite können sie als Klassenmethoden implementiert werden, auf er anderen Seite können sie auch als Programme oder Funktionen ihren Platz auf der Seite der Applikationen erhalten. Da es sich um rein lesende Zugriffe handelt, müssen sie nicht in Transaktionsblöcke verpackt und somit auch nicht zwingend aus dem Methodenteil herausgehalten werden. Also bleibt als Beurteilungskriterium nur noch der Aspekt der Wiederverwendbarkeit der Klassen.

4.3 Das Transaktionsdiagramm

In der Methode von Booch wird das Moduldiagramm für den physischen Entwurf der Module verwendet. Betrachtet werden die Abhängigkeiten der verschiedenen Module auf der Ebene der Definitionen und der Implementationen. Während bei den Klassendiagrammen die statische Struktur der Klassen und deren Beziehungen dargestellt wurden, werden diese Klassen nun verschiedenen Implementationsmodulen zugeordnet. Haben zwei Klassen im Klassendiagramm Abgängigkeiten, müssen sie entweder in demselben Modul zu liegen kommen oder in unterschiedlichen Modulen, welche ihrerseits durch eine Modulabhängigkeit verbunden sein müssen.

Compiler

Im Zentrum der Betrachtung steht bei dieser Modellierung vor allem das fehlerfreie Terminieren des Compilers. Unterhalten zwei Klassen eine Beziehung und liegen sie nicht im selben Modul, muss - für C++ z.B. über "include“-Direktiven - der einen Klasse die Definition der anderen zugänglich gemacht werden. Diese Abhängigkeiten werden durch Verknüpfungen der entsprechenden Module modelliert.

Implementation der Schnittstellen

Nicht im Betrachtungsfeld befindet sich die Implementation der Schnittstellen zwischen den verschiedenen Modulen, es wird vielmehr angenommen, dass diese aus dem Klassendiagramm abgelesen werden können.

Für Datenbankapplikationen aber sind die Anwendungen selber nicht als Klassen implementiert und bilden somit keinen Bestandteil der eigentlichen Datenbasis, sind aber dennoch im Datenbankschema enthalten. Zudem müssen Transaktionen als zentraler Bestandteil von Datenbankprogrammen entworfen werden können.

Grobentwurf

Aus diesem Grund wird das Moduldiagramm von Booch von der Idee her übernommen und vor allem an seiner Stelle des Entwurfprozesses belassen. Die Konstrukte aber dienen nur mangelhaft den erweiterten Anforderungen und können höchstens in einer frühen Phase des Grobentwurfes, namentlich zur Identifikation der Subsysteme, verwendet werden.

Moduldiagramm - Transaktionsdiagramm

Zudem wird der zentralen Aufgabe dieses Entwurfschrittes, die Definition des Transaktionskonzeptes durch Umbenennung in „Transaktionsdiagramm“ Rechnung getragen.

4.3.1 Subsystemdiagramm

„divide-and-conquer“ durch Subsysteme

Im Subsystemdiagramm werden während des Grobentwurfes die verschiedenen Subsysteme einer Applikation identifiziert. In Subsystemen werden logisch zusammengehörige Funktionalitäten unter der Berücksichtigung der „divide-and-conquer“-Strategie verpackt.

Ein derartiges Diagramm besitzt eine Applikationsikone als Wurzelkonstrukt, welches die Kontrolle über die verschiedenen Subsysteme ausübt. Existieren weitere Beziehungen zwischen den Subsystemen, so wird damit ausgedrückt, dass mindestens ein Element des einen Subsystem mindestens ein Element des anderen Subsystem aufruft.

4.3.2 Transaktionsdiagramm

Die im vorgelagerten Grobentwurf gefundenen Subsysteme werden in den folgenden Detaillierungsschritten zu Transaktionsdiagrammen expandiert.

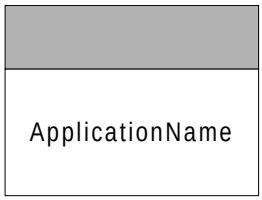
Applikationen, Programme, Funktionen und Transaktionen

Das Transaktionsdiagramm besteht aus vier aufeinandergeschichteten Ebenen, welche auch als Abstraktionsstufen oder Detaillierungsgrade angesehen werden können. Die Ebenen beinhalten von oben nach unten aufgezählt die Applikationen, die Programme, die Funktionen und die Transaktionen.

4.3.3 Konstrukte

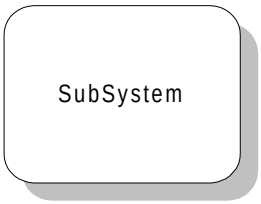
Die nachfolgend beschriebenen Konstrukte Applikation und Subsystem werden für das Subsystemdiagramm verwendet, während die restlichen Ikonen zum Entwurf der Transaktionen innerhalb des Transaktionsdiagrammes zur Anwendung gelangen.

4.3.3.1 Applikationen



Die Applikationen bilden die oberste Ebene des Moduldiagramms. Eine Applikation ist eine Sammlung von Programmen und bildet somit die Wurzel des Aufrufbaumes. Eine Applikation kann nur Programme aufrufen und entsprechend auch nur mit diesen Ikonen verknüpft sein.

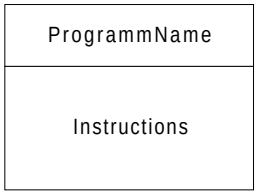
4.3.3.2 Subsysteme



In Subsysteme werden logisch zusammengehörige Programme, Funktionen und Transaktionen zusammengefasst. Ein Subsystem kann auch als Rahmen einer Untermenge von Subsystemen stehen. Ein Subsystem muss entweder weitere Subsysteme oder mindestens ein Programm enthalten, damit die Aufrufverbindung semantisch sinnvoll ist. Somit können von Applikationen aus auch Subsysteme aufgerufen werden.

Das Subsystemkonstrukt darf nur im Subsystemdiagramm, welches auf diese Weise die Rolle eines Übersichts- oder Kontextdiagramms einnimmt, verwendet werden.

4.3.3.3 Programme



Programme sind Sammlungen von Instruktionen. Das Symbol definiert einen eindeutigen Namen im Titel und einer Sequenz von Instruktionen im Rumpf. Instruktionen können in Form von Anweisungen und Steuerblocks auftreten. Anweisungen sind Programmzeilen, die eine Deklarationen, eine Zuweisungen oder einen Aufruf enthalten. Steuerblocks kontrollieren den Programmausführungsfluss und sind entweder Selektion, Iteration oder Wiederholschleifen.

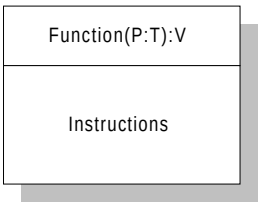
Mit diesen Konstrukten lassen sich Programmabläufe pseudocodeartig entwerfen. Insbesondere können die Aufrufe der Funktionen, der Transaktionen und der nur lesenden Klassenmethoden dargestellt werden. Aufrufe von Methoden, die schreibend auf die Objekte zugreifen, sind in Transaktionen zu schachteln und von dieser Ebene aus nicht aufrufbar. Auch dürfen keine anderen Programme aufgerufen werden.

Klassenmethoden werden in der Form *ClassName***::***MethodName***(**[*Param*(,*Param*)*]**)** aufgeschrieben, wobei sowohl die Klasse als auch deren Methode im Schemadiagramm definiert worden sein muss.

Programme gehören zu genau einer Applikation. Es ist zwar durchaus möglich, ein Programm zeichnerisch in zwei verschiedenen Applikationen unterzubringen, tatsächlich

werden aber physisch zwei verschiedene Kopien ein und desselben Programmkörpers im Schema abgelegt.

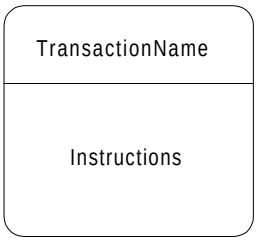
4.3.3.4 Funktionen



Analog zu den Programmikonen tragen die Funktionsdarstellungen den eindeutigen Namen der Funktion im Titelteil und die Instruktionen im Körperteil. Während es bei Programmen weder Übergabeparameter noch einen Rückgabewert gibt, sind diese bei der Funktiondefinition im Titelteil zu spezifizieren. P stehe dabei als Name des Parameters von Typ T und V definiere den Typ des Rückgabewertes.

Die pseudocodeartig beschriebenen Funktionsabläufe können Aufrufe von Transaktionen oder anderen Funktionen enthalten. Klassenmethoden, die schreibend auf die Objekte zugreifen, und Programmaufrufe sind nicht erlaubt.

4.3.3.5 Transaktionen



Transaktionen sollen die schreibenden Zugriffe auf die Datenbasis kapseln. Der Ausführungszeitraum einer Transaktion ist das einzige Zeitfenster, in welchem die Konsistenzbedingungen verletzt sein dürfen. Aus diesem Grund ist in jedem Fall darauf zu achten, nach der Ausführung der Schreibzugriffe die Konsistenz zu prüfen. Sind die formulierten Bedingungen auf irgend einer Ebene nicht erfüllt, müssen die Manipulationen durch ein Abort (Rollback) der Transaktion rückgängig gemacht werden.

Transaktionen bilden die atomaren Ausführungsschritte, welche nicht unterbrochen werden dürfen. Nicht zuletzt aus diesem Grund müssen sie so kurz wie möglich gehalten werden. Die physischen Gegebenheiten wie Mehrbenutzerbetrieb, welche später im physischen Entwurf genauer spezifiziert werden, sollen dabei nicht ausser Acht gelassen werden.

Transaktionen - logisch zusammenhängende Arbeitsschritte

Eine Transaktion sollte also möglichst aus den Methodenaufrufen, die im Schemadiagramm als schreibend deklariert worden sind, und den Konsistenzprüfungen bestehen. Häufig wird aber vielmehr ein logisch zusammenhängender Arbeitsschritt, welcher aus einem lesenden und einem schreibenden Teil besteht, in eine Transaktion gefasst.

geschachtelte Transaktionen

Transaktionen können von Programmen und Funktionen aufgerufen werden, dürfen aber ihrerseits nicht mehr in Untertransaktionenen verzweigen, solange das darunterliegende Datenbanksystem keine geschachtelten Transaktionen unterstützt.

Zudem ist die Zusammenfassung von verschiedenen Schreibzugriffen in einer Transaktion nur dann zu empfehlen, wenn zwischen den einzelnen Manipulationen die Konsistenz nicht gewährleistet werden kann.

4.3.3.6 Aufrufe

→ Durch die Aufrufpfeile werden die verschiedenen Ikonen miteinander verbunden, wobei der Pfeil beim Aufrufenden beginnt und beim Aufgerufenen in einer Spitze endet. Für die Verwendung sind die oben ausgeführten Einschränkungen zu beachten.

Die von einer Ikone auslaufenden Pfeile verstehen sich als synchrone Aufrufe und können, falls die Struktur aus dem Programm-, bzw. Funktionsbeschrieb nicht hervorgeht, durch Numerierungen sequenziert werden.

4.3.3.7 Beispiel FIS

Sämtliche Funktionalität des Firmeninformationssystems soll in eine einzige Applikation gepackt werden. Die verschiedenen Teile der Anwendung sollen sich um die Belange der Mitarbeiterverwaltung, der Büroverwaltung und der Projektverwaltung kümmern. Ein zusätzliches Modul soll Bürolisten, Mitarbeiterlisten und Projektteamlisten erstellen, anzeigen und drucken können. Die Grobstruktur lässt sich wie folgt in einem Subsystemdiagramm veranschaulichen.

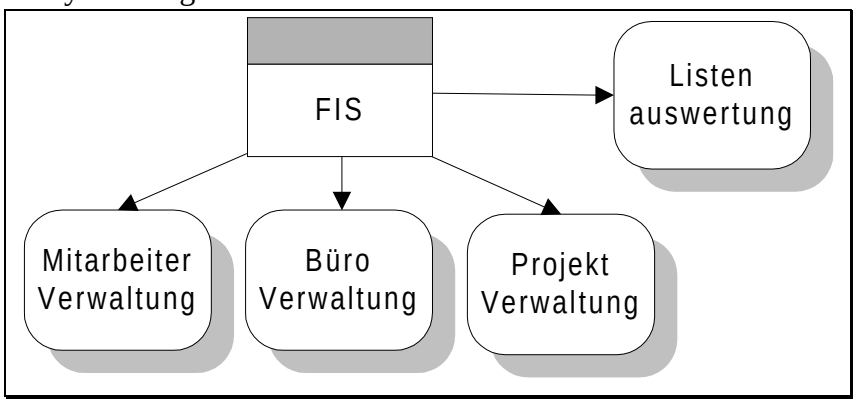


Abbildung 4-10: Subsystemdiagramm für die Anwendung FIS

Die identifizierten Subsysteme lassen sich nun in einer weiteren Abstraktionsstufe expandieren, um die entsprechenden Transaktionsdiagramme zu zeichnen. Dies sei am Subsystem „Mitarbeiterverwaltung“ aufgezeigt. Die Mitarbeiterverwaltung muss Entwickler und Manager anstellen und entlassen und Entwickler zu Manager befördern können.

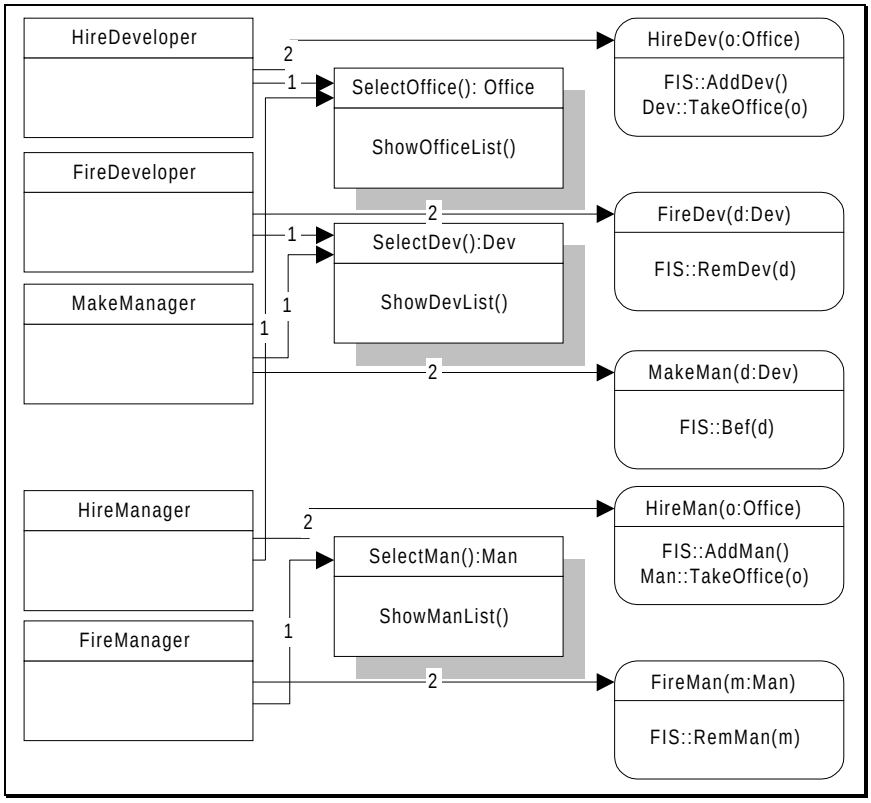


Abbildung 4-11: Transaktionsdiagramm für das Subsystem „Mitarbeiterverwaltung“

Die fünf Hauptfunktionen der Mitarbeiterverwaltung sind je in einem Programm implementiert. Die Programme sind nicht näher spezifiziert, da die Reihenfolge der Aufrufe zu den Funktionen und Transaktionen mit Abfolgenummern versehen sind.
Die Funktion SelectDev() ruft eine Funktion ShowDevList() auf. Bei dieser Funktion handelt es sich nicht um eine Klassenmethode, sondern um eine Funktion, die sich in einem anderen Subsystem befindet. Wären die beiden Funktionen im selben Subsystem enthalten, so würde anstelle des Funktionsaufrufes eine Aufrufbeziehung zwischen den beiden bestehen.

4.4 Verwendungsrichtlinien

In den vorigen Abschnitten wurden die verschiedenen Diagramme und deren Konstrukte definiert. Die Definition bezog sich aber stets auf die syntaktische Verwendung der Elemente, während dabei die Semantik nur als Begründung für die syntaktischen Entscheide beschrieben wurde.
Für die Anwendung der Methode auf ein bestimmtes Entwurfsproblem bleiben die Fragen

- wie entwerfe ich in bestimmten Situationen und
- wann soll ich welches Konstrukt verwenden?

noch offen. Diese Fragestellungen sollen in den folgenden Abschnitten durch verschiedene Vorschläge für Vorgehensrichtlinien beantwortet werden.

Vorgehensrichtlinien versus Freiheit des Entwerfers

Die Formulierung von Vorgehensrichtlinien ist immer ein Balanceakt. Auf der einen Seite soll die Freiheit des Entwerfers möglichst nicht beschnitten werden und auf der andreren Seite verbessern strenge Regeln die spätere Umsetzung der entworfenen Systeme in Anwendungsprogramme. Die nachfolgenden Richtlinien sind eher bezüglich der Sicherung der Datenintegrität optimiert und deshalb streng formuliert.

Zuerst werden verschiedene Aspekte der Verwendung von Konstrukten der Schema- und Transaktionsdiagramme beleuchtet, anschliessend soll dem Transaktionsentwurf und der Konsistenzsicherung besondere Aufmerksamkeit geschenkt werden.

4.4.1 Verwendung der Konstrukte eines Schemadiagramms

4.4.1.1 Persistente und transiente Objekte

Für die Modellierung von persistenten Objekten sollen die Datenbankklassen verwendet werden, welche als einzige persistenzfähig sind. Transiente Objekte hingegen sollen als Instanzen von Hilfsklassen aufgefasst werden. Abstrakte Klassen können keine Instanzen ausbilden, weshalb diese lediglich für die Beschreibung zusammenfassender Klasseneigenschaften in den strukturellen Entwurf einbezogen werden sollen. Die Definition von explizit persistenten Objekten kann mit Hilfe der Einstiegspunkte erreicht werden.

4.4.1.2 Exklusive existenzabhängige Aggregation

Die Modellierung von exklusiver existenzabhängiger Aggregation wird gut unterstützt. Die Verwendung der Komponentenbeziehung bildet exakt dieses Verhältnis zweier Klassen ab, wenn auf der rückbezüglichen Seite der Beziehung die Kardinalität gleich eins gesetzt wird. Die Erzeugungs- und Löschfortpflanzung ist gewährleistet.

4.4.1.3 Nicht-exklusive existenzabhängige Aggregation

Etwas problematischer verhält sich die Methode DEIMOS bei der Modellierung von nicht-exklusiven existenzabhängigen Aggregationsbeziehungen. Dies sei am Beispiel der Konferenzadministration (PCSS) veranschaulicht.

Ein Programmkomitee (PC) besteht aus Mitgliedern. Die Mitglieder können in mehreren Programmkomitees als Experten auftreten. Experten überprüfen eingereichte Papiere. Papiere können an mehrere Konferenzen eingereicht werden.

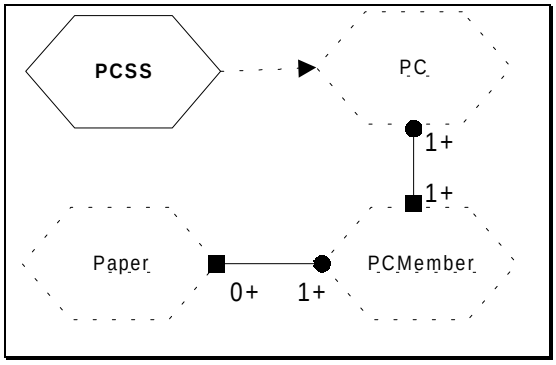


Abbildung 4-12: Beispiel Konferenzadministration: erster Entwurf

Der erste Entwurf betrachtet ein Papier als eine Komponente des Mitgliedes. Diese Modellierung ist aber künstlich und bildet intuitiv nicht den Sachverhalt der Realwelt ab. Zudem ist bei dieser Art der Struktur die Löschung nicht mehr eindeutig. Dies sei am folgenden Objektdiagramm veranschaulicht:

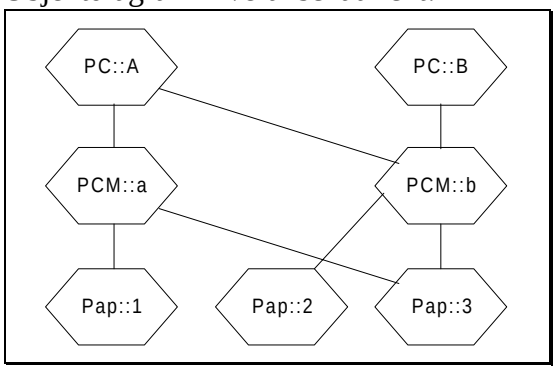


Abbildung 4-13: Objektdiagramm einer bestimmten Situation

Im Objektdiagramm sind die Komponentenbeziehungen zu einem bestimmten Zeitpunkt dargestellt. Soll nun die Konferenz „B“ gelöscht werden, entstehen einige Probleme:

- Soll das Papier „3“ gelöscht werden? Die Beziehung zum Mitglied „a“ deutet zwar darauf hin, dass dieses Papier noch in einer anderen Konferenz verwendet wird, aber was ist, wenn dieses Mitglied zusätzlich zu den obigen Annahmen noch im Komitee „B“ als Experte figuriert?
- Soll das Papier „2“ gelöscht werden? Es kann nicht mehr entschieden werden, zu welcher Konferenz es eingereicht wurde.

Die Beantwortung dieser Fragen, bzw. das korrekte Verhalten des Systems in diesen Situationen muss durch zusätzlichen Programmieraufwand erzwungen werden. Würde man das Papier hingegen als Komponente der Konferenz modellieren, könnten derartige Situationen nicht auftreten.

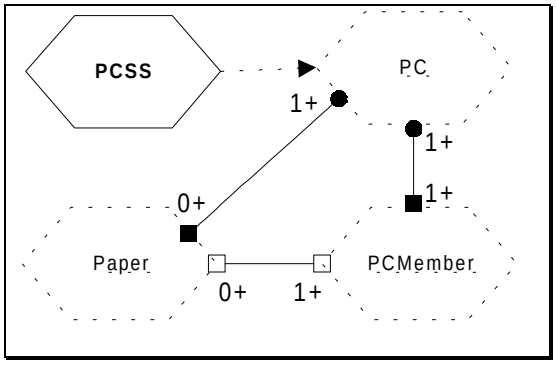


Abbildung 4-14: Schemadiagramm mit Papier als Komponente der Konferenz

Die Löschung der Papiere ist somit geregelt, da eine direkte Beziehung zwischen den Papieren und der Konferenz besteht. Es kann also die folgende Verwendungsrichtlinie für die nicht exklusive extistenzabhängige Aggregation formuliert werden:

📖 *Objekte, die als Komponenten von mehreren Objekten auftreten, dürfen ihrerseits nur Beziehungen zu Objekten unterhalten, die ebenfalls Komponenten ihres Besitzers sind.*

4.4.1.4 Nicht-existenzabhängige Aggregation

Für den Entwurf von Objekten, die eine nicht-existenzabhängige Aggregationsbeziehung unterhalten (die Löschung des Aggregats weitet sich nicht auf die Komponenten aus und umgekehrt), kann folgede Regel festgehalten werden:

📖 *Nicht-existenzabhängige Aggregation wird durch das Konstrukt der inversen Beziehung abgedeckt.*

4.4.1.5 Rekursive Aggregation

In der Realwelt trifft man in gewissen Situationen rekursive Aggregation an. Als Beispiel soll eine Firma betrachtet werden, deren Abteilungen ihrerseits Abteilungen enthalten können. In DEIMOS kann für die Abbildung der beschriebenen Struktur nicht die Komponentenbeziehung verwendet werden, da die Eindeutigkeit des Komponentenbaumes gefordert worden ist.

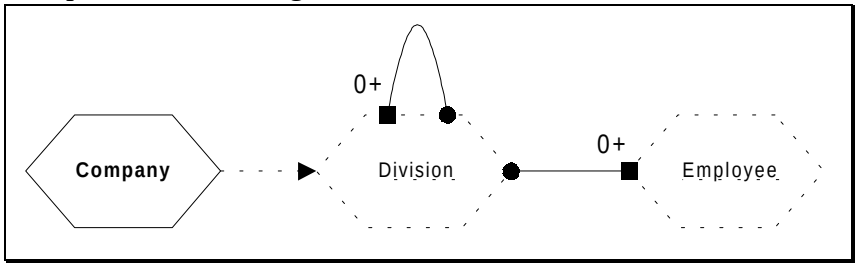


Abbildung 4-15: Schemadiagramm mit komponentenbeziehungsbasierter rekursiver Aggregation

Eine Abteilung wäre, falls diese Forderung aufgeweicht würde, einerseits eine Komponente der Firma oder andererseits eine Komponente einer Abteilung. Somit würde man die Fähigkeiten der Firmeninstanz, nämlich die

Existenzkontrolle (Erzeugung und Löschung) und die Extensionseigenschaft (Schutz der Eindeutigkeit für die Schlüsselbeziehungen), stark einschränken. Die Löschfortpflanzung und die Konsistenzsicherung wären bei der Umsetzung in ein physisches Schema nicht mehr ohne zusätzlichen Programmieraufwand möglich.
Aus diesem Grund sind rekursive Aggregationen mit Hilfe der inversen Beziehungen darzustellen. Die Existenzabhängigkeit ist durch die Angabe der Kardinalität 1 am Ausgangsort der Beziehung zu erreichen.

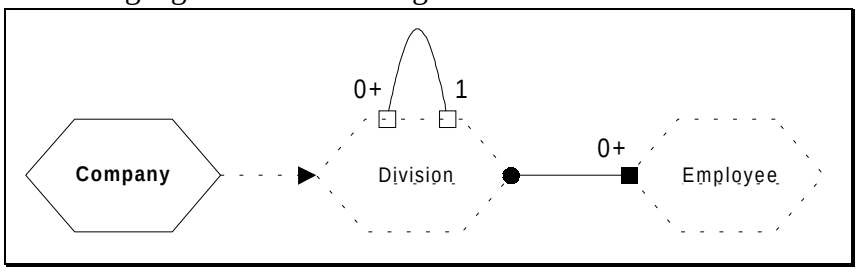


Abbildung 4-16: Schemadiagramm mit rekursiver Aggregation, basierend auf inverser Beziehung

Als Regel lässt sich folgendes formulieren:

📖 *Die rekursive Aggregation muss durch die Verwendung einer rückbezüglichen, inversen Beziehung modelliert werden.*

4.4.1.6 Extensionen und Schlüssel

Erneut soll das Beispiel der Konferenzadministration herangezogen werden. Der Entwerfer möchte erreichen, dass sämtliche PC-Mitglieder bezüglich ihres Vor- und Nachnamens eindeutig identifizierbar sind. Ausgehend vom obigen Entwurf betrachtet er die einzelnen Programmkomitees als lokale Extensionen der Mitglieder. Somit definiert er als Eigenschaft der Klasse PC einen Schlüssel
PCMember<Name,FirstName>
Damit hat er erreicht, dass er die Instanzen der Klasse PC für die Eindeutigkeit des Schlüsselkriteriums bei der Erzeugung von Mitgliedern überprüfen kann. Nun ist er aber mit dem Resultat noch nicht zufrieden, weil er eigentlich eine globale Eindeutigkeit erzielen wollte. Also muss er das Schemadiagramm geringfügig anpassen.

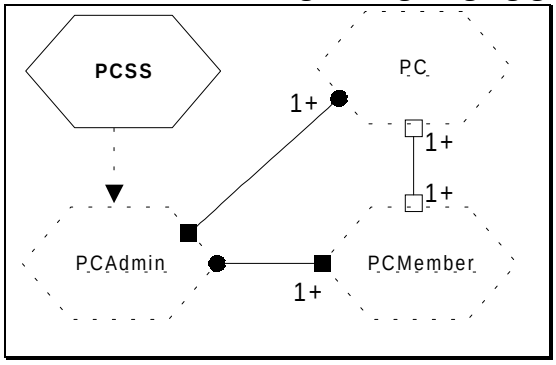


Abbildung 4-17: Schemadiagramm für die Konferenzadministration mit globalem Sammelobjekt

Die erste Möglichkeit besteht darin, die Mitglieder als Komponenten eines neuen globalen Verwaltungsobjektes PCAdmin zu betrachten. Damit ist die globale Eindeutigkeit erreicht, da nur eine Instanz dieser Klasse - durch die Instantiierung - erzeugt wird.
Als zweite Möglichkeit kann auch eine Extensionsklasse ins Schema eingeführt werden.

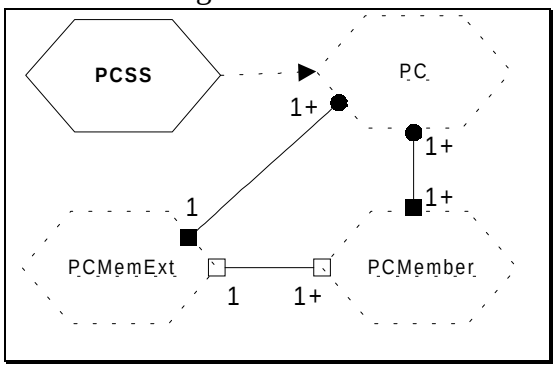


Abbildung 4-18: Schemadiagramm für die Konferenzadministration mit Extension

Diese Extensionsklasse ist beispielsweise als geteilte Komponente auffassbar (jede Instanz der Klasse PC verweist auf dieselbe Instanz der PCMemExt) oder als Komponente einer zusätzlichen Administrationsklasse (siehe oben). Die Eindeutigkeit des Schlüssels kann nun beim Aufbau der zwingenden Beziehung zwischen PCMember und PCMemExt, welche bei der Erzeugung einer Instanz der Klasse PCMember installiert werden muss, geprüft werden.
Als Regel für die Modellierung von globaler Eindeutigkeit lässt sich also folgendes festhalten:

🕮 *Globale Eindeutigkeit lässt sich nur dann erreichen, wenn alle Instanzen für welche das Eindeutigkeitskriterium gelten soll, Komponente desselben Objektes sind, oder wenn ein einmaliges Objekt exisitert, welches zu allen fraglichen Objekten eine inverse Beziehung mit der rückbezüglichen Kardinalität 1 besitzt.*

4.4.2 Verwendung der Konstrukte eines Transaktionsdiagramms

4.4.2.1 Transaktionsdiagramm

Das Transaktionsdiagramm wird für den Entwurf und die Definition der Lese- und Schreibtransaktionen verwendet.
In vielen der heute verfügbaren Datenbankimplementierungen werden die Applikationen nicht im Datenbankschema definiert (vlg. “Aktuelle Implementierungen”, Seite 18), sondern ausserhalb in Form von beispielsweise C++-Programmen, welche über definierte Schnittstellen (vlg. “

sort by
group by with

Für den Umgang mit Mengen stehen noch weitere Ausdrücke zur Verfügung. So kann eine Menge durch **sort** ... **in** ... **by** ... sortiert oder durch **group** ... **in** ... **by** ... **with** ... gruppiert werden, wobei der **with-Teil** ähnlich einer Klausel **having** in SQL eine einschränkende Bedingung enthalten kann.

union
intersect
except

Verschiedene aber gleichartige Menge können durch **union** verbunden (Vereinigungsmenge), durch **intersect** geschnitten (Durchschnittsmenge) oder durch **except** voneinander abgezogen werden (Differenzmenge).

element
flatten

Will man die innere Struktur einer Menge verändern, so existieren **element,** zur Schaffung einer Menge aus einem Wert, **flatten** für das Ausflachen einer homogenen Multimenge zu einer Menge, oder **flatten** zur Überführung einer Liste zu einer Menge als Möglichkeiten.

Anbindung an Programmiersprachen", Seite 15) auf die persistenten Objekte der Datenbasis zugreifen.
In der gewählten Umgebung O_2 hingegen sind Applikationsklasse und -instanzen aber Teil des Schemas und sollen daher auch in den Schemaentwurf einbezogen werden können.

4.4.2.2 Konstrukte des Subsystemdiagramms

Das Subsystemdiagramm bietet Möglichkeiten zum Grobentwurf von Datenbankapplikationen an. Die Konstrukte sind die Applikation, welche die als Sammlung von Subsystemen und/oder Programmen aufgefasst werden darf und die Subsysteme, die weitere Subsysteme oder aber direkt Programme enthalten.

📖 *Applikationen stellen Anwendungen dar, während Subsysteme für die Aufteilung des Problembereiches verwendet werden.*

4.4.2.3 Konstrukte des Transaktionsdiagramms

Im Transaktionsdiagramm werden Programme, Funktionen und Transaktionen als Konstrukte angeboten. Diese sollen nun wie folgt unterschieden und entsprechend verwendet werden:
Ein Programm soll als Lesetransaktion aufgefasst werden. In einem Programm werden also alle Datenbankoperationen untergebracht werden, welche nur lesend auf die Datenbasis zugreifen. Werden schreibende Aktionen verwendet, müssen diese in Transaktionen, welche vom Programm aus aufgerufen werden können, gepackt werden.
Eine Funktion kann als freier Programmblock einer Lesetransaktion betrachtet werden und fasst analog zum allgemeinen Verständnis von Funktionen wiederkehrende Aufgaben in wiederverwendbare Codeteile zusammen.

Eine Transaktionen hingegen ist eine Schreibtransaktionen und soll für die Zusammenfassung von Datenbankoperationen in logischen Arbeitseinheiten der Realwelt verwendet werden.
Zusammenfassend gelten für die Konstrukte des Transaktionsdiagramms die folgenden Verwendungsrichtlinien:

📖 *Programme werden für die Darstellung von Lesetransaktionen,*
Funktionen für die Zusammenfassung von wiederkehrenden Anweisungsblöcken und
Transaktionen für die Gliederung der schreibenen Datenzugriffe in logische Arbeitseinheiten verwendet.

4.4.3 Transaktionsentwurf

Die Transaktion ist, wie im vorigen Kapitel ausgeführt, das zentrale Instrument, um Daten innerhalb von Datenbanksystemen zu erzeugen, zu löschen und zu manipulieren. Sie bilden in Datenbanksystemen die Einheit der Konsistenz, der Dauerhaftigkeit und der Atomarität.

4.4.3.1 Transaktionskomplexität

Datenmanipulationen und physischen Aspekte

Der Systemarchitekt muss sich beim Entwurf überlegen, welche Datenmanipulationen er in eine bestimmte Transaktion verpacken will. Insbesondere muss er sich auch die physischen Aspekte, die er in einem späteren Entwurfsschritt noch genauer spezifizieren kann, vergegenwärtigen, da während der Ausführung Sperrungen auf die Daten propagiert werden.

Ausführungsdauer

Für die Ausführungsdauer einer Transaktion kann man sich zwei Extremfälle vorstellen:

Maximalfall: Eine Sitzung - Eine Transaktion

Im Maximalfall beginnt die Transaktion am Anfang der Sitzung (Start der Anwendung) und endet, wenn die Applikation geschlossen wird. Dieser Ansatz ist darum sehr gefährlich, weil zum einen keine anderen Benutzer die angelegten Datenbestände lesen oder beschreiben können, was den Mehrbenutzerbetrieb völlig ausser Kraft setzt, und zum anderen keine Datensicherheit besteht, da das System nach Wiederanlauf sämtliche Änderungen verworfen haben wird.

Minimalfall: Jede Manipulation - Eine Transaktion

Im Minimalfall hingegen wird jede Anweisung, welche Attributwerte von persistenten Objekten verändert, in eine Transaktion verpackt. Mit dieser Strategie können Methoden mehrere Transaktionsblöcke enthalten und zudem ist der Methodenaufruf selber in einer Transaktion untergebracht. Die Leistungsfähigkeit derart implementierter Systeme wird mit solchen Ressourcenverschleuderungen unnötig vermindert. Ferner kann in einem Fehlerfall nur die fehlgeschlagene Transaktion rückgängig gemacht werden. Die logisch zu derselben Manipulation gehörenden Teile sind bereits in der Datenbank gespeichert und müssen durch aufwendige Fehlerbehandlungsroutinen (die selber auch wieder Fehler verursachen können) auf einen konsistenten Zustand zurückgeführt werden. Diese Strategie setzt also die gewünschten Effekte der Transaktion - Übergang von einem konsistenten Zustand in den nächsten oder im Fehlerfall auf denselben zurück - gänzlich ausser Kraft.

Die angemessene Dauer der Transaktion muss also zwischen diesen beiden Extremvarianten gewählt werden. Insbesondere müssen zur genaueren Spezifikation des Transaktionsinhaltes Aspekte der Semantik der Realwelt herangezogen werden. Eine Transaktion soll beendet werden, wenn

- ein logischer Arbeitsschritt beendet ist,
- die Änderungen dauerhaft (für andere Benutzer sichtbar) gemacht werden sollen und
- die Datenintegrität erreicht werden kann.

Anstelle der physischen Aspekte, dass

- sich lange Transaktionen aufgrund der Sperrungen nachteilig auf den Mehrbenutzerbetrieb auswirken und
- kurze Transaktionen das System belasten (Commits sind teure Operationen),

werden also vielmehr Kriterien der Logik eines Systems für die Modellierung der Transaktionen angewendet:

- Definition der Transaktionen nach logischen Arbeitsschritten (Semantik der Realwelt).
- Im Fehlerfall soll auf einen logisch sinnvollen Zustand zurückgesetzt werden können (Abort). Zwischenresultate sollen also nicht gespeichert werden.

4.4.3.2 Transaktionsaufrufe

Plazierung der Transaktionen

Die zweite Überlegung, die der Entwerfer unbedingt anstellen muss, ist die Plazierung der Transaktionen innerhalb des Klassen- und Applikationsgefüges. Wiederum können bei diesen Erörterungen zwei verschiedene Extremstrategien verfolgt werden. Transaktionen können ausschliesslich in den Methoden der Datenbankklassen verpackt sein oder gänzlich aus den Methoden ausgelagert in den Transaktionsblöcken der Applikationen implementiert werden.

Transaktionen in Klassenmethoden

Werden die Transaktionen in den Klassenmethoden untergebracht, geht dieses Vorgehen zumeist zu Lasten der Wiederverwendbarkeit. Der Entwurf der Datenbankklassen berücksichtigt nur die Semantik der Datenhaltung, nicht aber die der Datenverwendung und Datenmanipulation. Letzere Aspekte werden durch die um die Datenbestände herumgebauten Programme spezifiziert. Datenbankklassen, welche eingekapselte Transaktionen in ihren Methoden verbergen, sind damit zweckgebunden.

Transaktionen in Applikationen

Aus diesem Grund sollten Transaktionen nicht in Klassenmethoden untergebracht werden. Auch Mischformen (z.B. Implementation der Transaktionen ausschliesslich in Hilfsklassen) unterliegen diesen Gefahren. Es empfiehlt sich also hier die Extremvariante zu wählen, die sämtliche Transaktionen von den Klassen der Datenbasis fernhält und in die Programme auslagert, welche die Datenbestände benutzen und verändern.

Ein weiterer Vorteil, der sich aus dieser Wahl ergibt, kann in der Systemunterstützung bei der Formulierung von Transaktionen durch das Sprachkonstrukt „transaction“ gesehen werden. Nur so deklarierte Anweisungsblöcke sind vom System O_2 für die Unterbringung der Transaktionen vorgesehen, was sich darin äussert, dass Zuwiderhandlungen vom Datenbanksystem entdeckt werden und von den Entwicklern durch Meldungen wahrgenommen werden können.

4.4.3.3 Transaktionsentwurf

lesende und schreibende Methoden

Mit der Methode wurde nun der Diagrammtyp „Moduldiagramm“ dergestalt erweitert, dass der Entwurf der Transaktionen überhaupt ermöglicht wird. Zudem ist durch die Typisierung der öffentlichen Methoden der Datenbankklassen in lesende und schreibende eine weitere Sicherheit eingebaut worden. Der Entwerfer muss sich zwangsläufig klar werden, ob eine Methode die Attributwerte verändert. Im Transaktionensentwurfsschritt muss er darauf achten, dass er schreibende Methoden immer in Transaktionen verpackt und lesende möglichst aus diesen Blöcken auslagert.

Zusammenfassend lassen sich also die folgenden Regeln formulieren:

📖 *Transaktionen sollen so gestaltet werden, dass sie logische Arbeitseinheiten abbilden und dass im Fehlerfall auf ein logisch sinnvollen Zustand zurückgekehrt werden kann.*

Transaktionen sollen nicht in Klassenmethoden, sondern ausschliesslich in den dafür vorgesehenen Konstrukten der Datenmanipulationssprache formuliert werden.

Dem Entwerfer wird also mit dieser Methode ein Werkzeug in die Hand gelegt, welches ihn hinsichtlich der Problematik der Transaktionen sensibilisiert und ihn in der sicheren Manipulation der Daten entsprechend unterstützt.

4.4.4 Fortpflanzung und Konsistenzsicherung

Die Wahrung der Konsistenz bei Manipulationen ist aufgrund der Forderung nach Komponentenbäumen und der Einschränkung, dass ausschliesslich inverse Beziehungen zwischen den Datenbankklassen herrschen dürfen, ermöglicht. Im Folgenden soll für die Objekterzeugung und -löschung ein Vorgehen aufgezeigt werden, welches die Datenintegrität bezüglich Fortpflanzung sichert. Anschliessend sind die Implikationen für eine Umsetzung eines Entwurfs in die Implementation ausgeführt.

4.4.4.1 Erzeugung

Die Konsistenzprüfung bei der Kreation neuer Klasseninstanzen muss innerhalb der folgenden Transaktion ablaufen:

1. Beginn der Transaktion.
2. Kreation der Instanz von ihrer Aggregationsinstanz aus.
3. Initialisierung der neuen Instanz.
4. Prüfung der Invariante für die neue Instanz am Ende der Initialisierungsmethode.
5. Aufbau der Komponentenbeziehung.
6. Prüfung der expliziten Konsistenzbedingungen.
7. Abschluss der Transaktion mit Erfolg oder Misserfolg.

Erzeugung der Subkomponenten

Besitzt die zu kreierende Instanz ihrerseits zwingende Komponentenbeziehungen, also Verbindungen, die Kardinalitäten grösser als 1 vorsehen, werden innerhalb des oben beschriebenen Schrittes 2 die entsprechenden Subkomponenten gleichfalls erstellt.

rekursive Erzeugung

Man darf sich somit die beschriebene Transaktion rekursiv vorstellen. Trotz dieser rekursiven Definition ist die Terminierung gewährleistet, da Komponenten und Subkomponenten nur entlang des Komponentenbaumes kreiert werden und dieser gemäss Definition keine Zyklen enthalten kann. Zudem müssen für die Minima bei der Kardinalitätsdefinition der Komponentenbeziehungen endliche Werte angegeben werden.

Initialisierung und Konsistenzprüfung

Im dritten Schritt werden die Attribute initialisiert. Sind bei der Definition der Klasse für die Attribute Standardwerte angegeben worden, werden diese den entsprechenden Attributen zugewiesen. Am Ende dieser Initialisierung wird die Invariante überprüft. Der Entwerfer ist also gefordert, das Zusammenspiel der Invariante und der Standardwerte aufeinander abzustimmen.

Aufbau der inversen Relationen

Inverse Relationen, an denen die neu entstehenden Komponenten teilnehmen, müssen gleichfalls während der Gesamttransaktion aufgebaut werden, damit die Kardinalitätsbedingungen nicht verletzt werden. Dies kann innerhalb der Abarbeitung der Initialisierungsroutine mittels Übergabe der entsprechenden Referenzen geschehen, oder, da diese Strategie nicht immer zur Konsistenz führt, explizit als dedizierter Transaktionsschritt.

Falls es sich bei der anzulegenden Instanz um einen Einstiegspunkt handelt, ist es naheliegend, dass der Schritt

2 nicht von der Aggregationsklasse aus durchgeführt werden kann und dementsprechend der Schritt 4 gänzlich überflüssig wird.

4.4.4.2 Löschung

Die Konsistenzprüfung bei der Löschung von Objekten muss innerhalb der folgenden Transaktion ablaufen:

1. Beginn der Transaktion.
2. Vorbereitung der Instanz auf die Löschung.
3. Löschung der Instanz von ihrer Aggregationsinstanz aus.
4. Abbau der Komponentenbeziehung.
5. Prüfung der expliziten Konsistenzbedingungen.
6. Abschluss der Transaktion mit Erfolg oder Misserfolg.

Löschvorbereitung für Subkomponenten

Unterhält die zu löschende Instanz ihrerseits Beziehungen zu anderen Objekten, so müssen diese während der Löschvorbereitung abgebaut werden. Dies gilt sowohl für die Komponentenbeziehungen als auch für die inversen Beziehungen. Wiederum dehnt sich also die beschriebene Transaktion auf die etwaigen Subkomponenten aus. Analog zu obiger Begründung ist die Terminierung dieser rekursiven Transaktion nicht gefährdet.

Ist von dieser Löschung ein Einstiegspunkt betroffen, fällt Schritt 4 weg und Schritt 3 kann unabhängig von einer Aggregationsklasse durchgeführt werden.

Konsistenzprüfung

Die Prüfung der Invariante für das Objekt am Ende der Löschvorbereitungsmethode ist überflüssig, da dieses entweder aus der Datenbasis entfernt wird oder bei nicht erfolgreicher Terminierung der Transaktion auf den „status quo ante“ zurückgesetzt wird.

Garbage Collection

Die Löschung von Subkomponenten wäre eigentlich nicht nötig, da innerhalb der Datenbankumgebung O_2 ein spezieller Prozess für das Aufräumen nicht mehr referenzierter Objekte verantwortlich ist (Garbage collector). Unterhalten die Objekte aber Beziehungen anderer Art, kann die Situation auftreten, dass Referenzen auf bereits gelöschte Objekte bestehen bleiben oder Objekte durch derartige Referenzen aufgrund der Persistenzfortpflanzungsstrategie unbeabsichtigt in der Basis erhalten bleiben.

4.4.4.3 Implikationen für die Umsetzung

Damit die Konsistenz des Schemas gesichert ist und die beschriebenen Fortpflanzungseffekte realisiert werden können, müssen die Datenbankklassen folgende Eigenschaften besitzen:

Sie müssen die Methoden

- für die Erzeugung aller notwendigen Subkomponenten,
- für Initialisierung der Attribute,
- für die Überprüfung der Invarianten,
- für die Überprüfung der impliziten

Konsistenzbedingungen und

- für die Löschvorbereitung

implementieren.

Löschvorbereitung bedeutet, dass sämtliche Beziehungen zu anderen Objekten abgebaut werden. Für alle Subkomponenten werden zuerst die entsprechenden Destroy-Methoden aufgerufen, damit diese sich ihrerseits auf die Löschung vorbereiten können. Danach werden die Objekte in Abhängigkeit der aktuellen Implementation des DBS explizit oder durch einen Aufräumdienst (Garbage Collection) implizit zerstört.

Verbindungsauf- bzw. abbau

Datenbankklassen müssen zudem für jede Beziehung je eine Methode besitzen, um die Verbindung auf- bzw. abzubauen. Für die inversen Beziehungen muss zudem zwischen dem aktiven und dem passiven Auf- bzw. Abbau unterschieden werden, was bei Komponentenbeziehungen, welche nur seitens des Besitzers kontrolliert werden, nicht gewährleistet sein muss.

In der letzten Eigenschaft liegt die Begründung, warum die Datenbankklassen nur Beziehungen vom Typ „Komponente“ und „invers“ unterhalten dürfen. Wären nämlich noch andere Typen zugelassen, wie etwa die einseitige „has-a by reference“-Beziehung, so wäre es einer beliebigen Instanz dieser Klasse nicht möglich, ohne Hilfe von anderen Instanzen eine konsistente Löschvorbereitung vorzunehmen, was die Propagierung der Löschung verunmöglichen würde.

Fortpflanzungseffekte

Die Konstrukte der Methode wurden also so gewählt, dass die Fortpflanzungseffekte so gut wie möglich unterstützt werden und der Entwickler der Datenbankapplikation so wenig wie möglich mit der Sicherung der Konsistenz zu tun hat.

Dennoch kann in manchen Fällen die Konsistenz gefährdet sein, da durch eine einzelne Manipulation die implizite Konsistenzbedingung, welche durch die Kardianlitätsangabe bei den inversen Beziehungen entstanden ist, verletzt werden könnte. Die Löschung wie auch das Einfügen pflanzen sich also nicht über derartige Beziehungstypen fort.

Man stelle sich ein Schema mit zwei Einstiegspunkten vor, die nicht Instanzen derselben Klasse sind. Damit hat man zwei disjunkte Komponentenbäume in seinem Entwurfsdiagramm. Seien nun die beiden Bäume durch eine inverse Beziehung miteinander verbunden und trage diese auf beiden Seiten die Kardinalität 1, so kann der Fall eintreten, dass eine Löschung nicht nur innerhalb des eigenen Baumes ihre Auswirkungen hat, sondern gleichermassen die Existenz von Instanzen im anderen Teilbaum semantisch verunmöglicht. Im Extremfall kann sich die Löschung einer Instanz semantisch sogar soweit fortpflanzen, dass sämtliche Objekte aus der Datenbasis entfernt werden müssten.

Die Eigenschaften der Datenbankklassen müssten also um die Fähigkeit, sich selber löschen zu können, erweitert werden. Diese Möglichkeit kann aber der Klasse selbst nicht gegeben werden, weil sonst neue Inkonsistenzen bei der Besitzer-Komponenten-Verbindung längs der Komponentenbeziehungen auftreten könnten.
Daraus folgt, dass die Funktionen für die Manipulationen im Allgemeinen - die oben beschreibenen Effekte treten ja nicht nur bei der Löschung allein auf - jeweils in zwei Ausprägungen verfügbar sein müssen. In der ersten wird die Manipulation abgebrochen, falls die Konsistenz nicht erreicht werden kann, während in der zweiten davon ausgegangen werden darf, dass es sich bei dieser Erzeugung nur um eine Teiltransaktion handelt und somit die übergeordnete Transaktion die Konsistenz am Ende der Sequenz prüft.

je zwei Methoden für Erzeugung und Löschung

Eine Datenbankklasse muss also für die Erzeugung wie auch für die Löschung mit je zwei Methoden aufwarten. Die eine ruft die Konsistenzsicherungs-Methode am Ausführungsende auf und sichert somit selber die Konsistenz, während die andere auf diese Prüfung verzichtet. Diese Funktion wird also neu in die Klassenschnittstelle aufgenommen, während für die Variante der inkonsistenten Erzeugung eine weitere Funktion zur Verfügung steht.

4.5 Bewertung der Methode

Nachfolgend soll überprüft werden, inwiefern sich die im vorigen Kapitel aufgedeckten Schwachstellen mit der neuen Methode ausräumen lassen.

4.5.1 Persistenz

In den betrachteten Methoden wurde beanstandet, dass nicht zwischen persistenten und transienten Klassen unterschieden werden kann, kein Konstrukt für die Darstellung eines persistenten Einstiegspunktes zur Verfügung steht und die Fortpflanzung der Persistenz nicht im Entwurf zu kontrollieren ist. Diese Konstrukte sind aber für den Schemaentwurf von zentraler Bedeutung.
Durch die Einführung eines Elementes für die Darstellung der Einstiegspunkte und eines Beziehungstyps (Komponentenbeziehung), der die Persistenz auf andere Klassen überträgt, sind dem Entwerfer Konstrukte in die Hand gelegt worden, mit denen er die Aspekte der Persistenz während des Entwurfes mitberücksichtigen kann. Die Schwachstelle konnte so vollständig beseitigt werden.
Die Erweiterung der Notation um Einstiegspunkte verursachte aber eine Aufweichung der in der Originalmethode strikte formulierten Trennung von Objekten und Klassen. In DEIMOS sind nun im Diagramm zur Modellierung der strukturellen Eigenschaften und Abhängigkeiten von Klassen auch einzelne Objekte zu finden. Zudem wurde dem Entwerfer durch die strenge Forderung nach eindeutigen Komponentenbäumen viel Entscheidungsfreiheit

entzogen. In vielen Fällen wird daher die intuitive Modellierung durch die Befolgung der starren Regeln eingeschränkt sein.
Man hätte auch auf die Definition des Einstiegspunktes im Rahmen des Schemadiagramms verzichten und diesen im Objektdiagramm spezifizieren können. Dies hätte aber sicherlich nicht zur besseren Lesbarkeit des Schemadiagramms beigetragen.
Ebenso hätte vo der Fortpflanzungskontrolle abgesehen werden können. Die Unterscheidung zwischen transienten und persistenzfähigen Klassen hätte schon viel für die Sicherheit der Datenbasis getan. Die Konsequenzen der Weglassung des expliziten Komponentenbeziehungstyps hätte sich aber nachteilig auf die Löschpropagation ausgewirkt (vgl. folgende Abschnitte).

4.5.2 Fortpflanzung und Konsistenzsicherung

Die Konzepte für die Fortpflanzung und die Sicherung der Datenintegrität sind sowohl im konzeptuellen Entwurf als auch in der Implementation des Datenbanksystems ungenügend berücksichtigt. Die Entwurfssprache enthält lediglich die Möglichkeit zur Formulierung von invarianten und das DBS prüft einzig inhärente Konsistenzbedingungen. Diese Konzepte sind aber für das Arbeiten mit Datenbanken von eminenter Wichtigkeit.
Dieser Schwachstelle wurde durch die strengen Forderungen der Notation und durch die Formulierung eines Transaktions- und Konsistenzsicherungs-Konzeptes zu Leibe gerückt. Damit konnte sie unter der Berücksichtigung der bereits diskutierten Einschränkungen weitgehend behoben werden.
Dafür musste aber sowohl seitens des Entwerfers als auch seitens des Entwicklers ein erheblicher Preis gezahlt werden: dem Entwerfer werden durch die strengen Verwendungsrichtlinien viele Freiheiten entzogen, er muss die definierten Vorgehensweisen strikte und ausnahmslos befolgen. Für den Entwickler ergeben sich viele Konsequenzen, welche er bei der Umsetzung des Entwurfes in die Implementation berücksichtigen muss. Lässt er sich aber bei diesem Prozess von Werkzeugen helfen oder kann er auf bestehende Rahmenwerke zurückgreifen, spart er viel Zeit und aufwendige Programmierarbeit.
Die Schwachstelle hätte auch bekämpft werden können, indem erhöhter Druck auf die Hersteller von objektorientierten Datenbanksystemen ausgeübt worden wäre, damit sie endlich die für relationale Systeme selbstverständlichen Konzepte in ihre Produkte hätten einfliessen lassen. Dies ist aber eine andere Problematik und soll entsprechend auch an einem anderen Ort diskutiert werden.

4.5.3 Extensionen und Schlüssel

Die ODMG-93 hält in ihrem Standardisierungsvorschlag die Definition von Schlüsseln für jede Klasse fest. Die Umsetzung dieses Konzeptes ist aber in der Implementierung der Datenbank nicht zu finden. Zudem stellt sie implizit keine Extensionen zur Verfügung. Die beiden Konzepte sind aber von grosser Bedeutung

für das Auffinden von Instanzen für die Formulierung von Eindeutigkeitskriterien und für die Auswertung von summarischen Abfragen.
Durch die Umsetzung der Forderung nach Komponentenbäumen konnte erreicht werden, dass alle persistenten Instanzen direkte oder indirekte Komponenten des Einstiegspunktes sein müssen. Es können also bezüglich der Komponentenbeziehungen keine losen oder ungebundenen Instanzen mehr in der Datenbasis enthalten sein. Diese Massnahme impliziert nun zum einen die Existenz von Extensionen (lokal oder global) und erlaubt zusätzlich die Formulierung von Schlüsselkriterien für die Komponentenbeziehungen, welche anschliessend auf einfache Weise in beim Einfügen oder Entfernen von Instanzen in die bzw. aus der Extension überprüft werden können. Die Schwachstelle wurde dadurch gänzlich behoben.
Eine Aggregationsklasse ist - sofern sie nicht durch den Einstiegspunkt instantiiert wird - immer eine lokale Extension der Instanzen ihrer Komponentenklasse. Auf der einen Seite ist diese Situation - wenn man globale Eindeutigkeit haben will - unbefriedigend, auf der anderen Seite lässt sie sich natürlich immer durch eine Umgestaltung der Komponentenbeziehungen modellieren. Es gilt also im Entwurfsprozess abzuwägen, ob lokale Eindeutigkeit ausreichend oder gar gewünscht ist, und wenn nicht, ob die Komponentenbeziehungen umgebaut werden sollen oder zusätzliche Hilfen (z.B. explizite Extensionsklasse mit einer inversen Beziehung zu sämtlichen Instanzen einer bestimmten Klasse) für die Erreichung der globalen Eindeutigkeit herangezogen werden sollen.
Mit einem anderen Ansatz, die Einführung von impliziten Extensionen, hätten weitgehend dieselben Resultate erzielt werden können. Zu jeder Datenbankklasse im Diagramm hätte man eine entsprechende Extensionsklasse generieren können, welche die Aufgaben des Einfügens, Löschens und Überprüfens der Schlüsselkriterien hätte übernehmen können. Dies hätte aber die Lesbarkeit der Diagramme erheblich beeinträchtigt, da derartige Klassen erst nach der Umsetzung sichtbar geworden wären. Zudem hätte eine lokale Eindeutigkeit, welche in vielen Anwendungen erwünscht ist, nur durch Einwirkung des Programmierers erreicht werden können.

4.5.4 Objektevolution

Die Fähigkeit zur Objektevolution ist in vielen Anwendungen eine wichtige Eigenschaft der persistenten Instanzen. Trotzdem werden derartige Migrationen von den betrachteten Datenbanksystemen nicht unterstützt. Da sie sich auch nicht erweitern lassen, muss der gewünschte Wechsel durch eine Erzeugung und eine Löschung simuliert werden. Dabei geht aber die Objektidentität verloren, welche für die Objektbeziehungen von zentraler Bedeutung ist, was der Kern der Problematik ausmacht.

Mit der Einführung des neuen Konstruktes für die Darstellung der Objektevolution wurden die Fähigkeiten des darunterliegenden Datenbanksystems nicht erweitert. Die in der Schwachstellenanalyse festgestellte Problematik, die entsteht, wenn ein Objekt seine eindeutige Kennung aufgeben muss, obschon es nicht zerstört wird und entsprechend die Verbindungen zu ihm erhalten bleiben sollen, wurde nicht gelöst. Vielmehr wurde eine Strategie entworfen, die derartige Manipulationen kontrollierter ablaufen lässt.
Somit wurde die Schwachstelle nicht behoben, sondern es wurde durch die Definition eines Konzeptes, welche den zur Objektevolution äquivalenten Effekt zum Ziel hat, eine gute Umgehung erreicht. Die Instanzmigration läuft in einem kontrollierten Prozess ab.
Der Übergang des Objektes A_i in das Objekt B_j unterliegt denselben Problemen wie die Löschungen, bzw. Erzeugungen von Datenbankklassen im Allgemeinen (vgl. “Fortpflanzung und Konsistenzsicherung”, Seite 112). Darüber hinaus unterliegt die Verwendung des Konstruktes sehr strengen Voraussetzungen, was die Entwurfsfreiheit erheblich beeinträchtigt.
Die Objektevolution besteht aus einem statischen Teil, welcher für gewisse Klasseninstanzen Übergänge als Eigenschaften definiert und einem dynamischen Teil, der die Umstände und Voraussetzungen für einen derartigen Wechsel beschreibt. Somit muss die Modellierung von Instanzmigration über mindestens zwei Diagramme hinweg angegeben werden.
Die Schwachstelle hätte auch behoben werden können, indem die Migration von Instanzen nur im dynamischen Teil beschrieben wäre. Zum einen hätte dies in komplizierten und schwer verständlichen Objektdiagrammen gegipfelt und zum anderen hätte man den beteiligten Klassen im Schemadiagramm diese Fähigkeiten nicht mehr angesehen. Das Schemadiagramm würde damit nicht mehr alle Klasseneigenschaften widerspiegeln.

4.5.5 Inverse Beziehungen

Trotz der Forderung der ODMG-93 nach inversen Beziehungen, finden sich diese nicht in der betrachteten Implementation des Datenbanksystems. Auch in der Notation von Booch existiert kein Konstrukt für die Definition von inversen Beziehungen. Dieser Beziehungstyp ist jedoch für beispielsweise die Löschfortplanzung von eminenter Wichtigkeit.
Durch die Einführung des Konstruktes für die Beschreibung der rückbezüglichen Beziehungen konnte die Schwachstelle ganzheitlich behoben werden.
Das gleichzeitige Weglassen anderer referenzieller Beziehungstypen (Assoziation, etc.) führte aber dazu, dass sämtliche Beziehungen invers definiert sein müssen, was in vielen Fällen zu einem gewissen Overhead führt.
Man hätte die Booch’sche Philosophie der beidseitigen Kardinalitätsangabe ohne Definition der Inversität beibehalten

können, hätte aber in diesem Fall dem Entwickler ein weiteres Problem überlassen, das er mit viel Programmieraufwand hätte lösen müssen, und gänzlich auf die Lösch- und Erzeugungssicherheit verzichten müssen.

4.5.6 Transaktionen

Die betrachtete Entwurfsmethode verfügt weder über entsprechende Konstrukte zur Definition von Transaktionsaufrufen noch über Vorgehensrichtlinien für die Formulierung von Konsistenzbedingungen zur Sicherung der semantischen Datenintegrität. Dies ist natürlich für die Modellierung von transaktionsorientierten Systemen eine grosse Schwachstelle. Durch die Widmung eines ganzen Diagrammtyps dieser Problematik und die entsprechende Einführung eines Transaktionskonstruktes auf der einen Seite und durch die Benennung eines Vorgehens für den Transaktionsentwurf ist der Schwachstelle in angemessener Form Rechnung getragen worden. Zudem können Klassenmethoden dank der Definition einer weiteren Eigenschaft in lesende und schreibende unterschieden werden. Mit Hilfe dieser Differenzierung kann ein Transaktionskonzept in einem Applikationsentwurf auf einfache Weise validiert werden.
Dass die Mechanismen ihre Wirkung erst bei einem sehr genauen Entwurf zeigen können, darf als Nachteil dieser Erweiterung gesehen werden. Häufig werden aber Systeme nur grob entworfen. Die Klassen sind freisprachlich spezifiziert und die Applikationen lediglich schematisch dargestellt. In diesen Fällen verfehlen natürlich die eingeführten Konstrukte ihre Aufgabe. Auf der anderen Seite kann der Zwang für den Entwerfer, sein System genau zu beschreiben, auch als Vorteil gesehen werden, da bei der Umsetzung durch den Entwickler keine grossen Ermessensspielräume mehr bestehen und das Ergebnis entsprechend auch besser auf die entworfene Lösung passt.
Man hätte auch gänzlich auf den Entwurf von Transaktionen verzichten können, indem man definiert, dass schreibende Methoden selbst als Transaktionen aufzufassen sind. Die Zusammenfassung derartiger Aufrufe in Transaktionsblöcke hätte man aber entsprechend dem Programmierer überlassen und die Voraussetzung verletzen müssen, dass der Entwerfer für die Wahrung der Datenintegrität verantwortlich ist.

4.5.7 Applikationen

Applikationen sind zwar in den objektorientierten Datenbankumgebungen Teil des Schemas, sie sind aber nicht als Instanzen von Applikationsklassen implementiert. Auch sind die Programme und Funktionen nicht als Methoden von Klassen auffassbar. Dennoch können sie Klassen des Schemas instanzieren, was eine Unterscheidung in transiente und persistenzfähige Klassen fordert.

Durch die Einführung des Transaktionsdiagramms konnte eine strenge Trennung der Applikationen von den Klassen des Schemas erreicht werden. Darüber hinaus erlaubt das neue Diagramm eine genaue Spezifikation der Schnittstellen zwischen der Applikation und der Datenbasis und legt die Verantwortlichkeiten der Applikation und der Objekte fest.
Der Transaktionsentwurf schützt das System aber nicht vor ungewollter Persistenzfortpflanzung. Vielmehr wird der Entwerfer aufgrund der strengen Regeln des Transaktionsdiagramms auf diese Problematik sensibilisiert. Dennoch können in den Ausführungsteilen der Anweisungsblöcke Klassen instantiiert und deren Methoden aufgerufen werden. Diese Objekte werden aber ausserhalb von Transaktionen erzeugt und daher nicht in der Datenbasis abgelegt. Auf diese Weise können sowohl Hilfs- wie auch Datenbankklassen temporär verwendet werden. In Datenbankumgebungen, in denen der Grundsatz der „Persistenz durch Erreichbarkeit“ gilt, muss trotzdem darauf geachtet werden, dass derartige Objekte nicht durch die Referenzierung von persistenten Objekten aus selber persistent werden.
Durch die von der Methode geforderte Unterscheidung zwischen Datenbankklassen, deren Instanzen persistenzfähig sind, und Hilfsklassen, deren Instanzen nicht persistent gemacht werden dürfen, können derartige unerwünschte Nebeneffekte bereits zur Entwurfszeit verhindert werden, da beim Aufbau der Beziehungen darauf geachtet werden kann, dass Datenbankklassen nur Verbindungen zu anderen persistenzfähigen Klassen unterhalten. Sollen Hilfsklassen mit Datenbankklassen verkettet werden, so ist dies nur zulässig, wenn die Relation der Hilfsklasse gehört und nur von ihr sicht-, bzw. auf- und abbaubar ist.
Applikationen können nicht mit den Klassendiagrammen entworfen werden. Vielmehr muss dafür eine zweite Ansicht, das Transaktionsdiagramm, herangezogen werden. Damit leidet die einfache Nachvollziehbarkeit, da die beiden sehr nahe beieinander liegenden Diagramme, welche beide statische Aspekte des Systems darstellen, physisch getrennt sind.
Die Schwachstelle konnte also nicht vollständig behoben werden. Dennoch wurde die Zielsetzung in einem hohen Masse erreicht, da durch Validierungen von anderen Entwerfern oder den Einsatz von Werkzeugen unerwünschte Effekte frühzeitig erkannt werden können.
Man hätte auch das Konstrukt der Hilfsklassen für die Modellierung von Applikationen, Programmen, Funktionen und Transaktionen erweitern können. Man hätte aber im Falle von O_2 virtuelle Klassen geschaffen, welche bei der Übertragung ins Datenbankschema in andere Strukturen umgesetzt worden wären. Dieser Ansatz wäre eher für Systeme geeignet, welche Applikationen nicht im Schema enthalten, sondern ausserhalb (z.B. C++) implementieren. In diesen Fällen könnte man sich aber berechtigterweise fragen, ob Applikationen nicht mit bereits

bestehenden Methoden (Booch, OMT, etc.) entworfen werden sollten.

4.5.8 Zusammenfassung

Mit der Definition der Methode wurden zwei Ziele verfolgt. Zum einen sollen die im vorherigen Kapitel beschriebenen und ausgewählten Schwachpunkte möglichst ausgeräumt werden. Zum anderen wurde durch das konsequente Streben nach Sicherheit sehr viel Wert auf die Integrität der Datenbasis gelegt.
Diese Zielverfolgung gipfelte in wenigen Konstrukten und strengen Anwendungsregeln, was sicherlich vor allem zu Lasten der Freiheit des Entwerfers und der Flexibilität des Entwurfes ging. Trotzdem ist in der Umgebung von grossen und komplexen Systemen der Sicherheit sicherlich die grösste Beachtung zu schenken.
Als grosser Vorteil der Methode kann der Umstand angesehen werden, dass sich Instanzen dank den Komponentenbeziehungen sicher erzeugen und löschen lassen. Bei der Umsetzung des Entwurfes erlaubt sie die Implementation von sicheren Konstruktoren, Destruktoren und Validierungsfunktionen. Sicherheit bedeute in diesem Zusammenhang, dass bei einer erfolgreichen Ausführung des Konstruktors, bzw. Destruktors die Datenbasis konsistent ist. In Fällen, in denen die selbstprüfende Variante der Erzeugungs- bzw. Löschfunktion nicht gewählt werden kann, stellen die veränderten Objekte Prüfungsmethoden in ihrer öffentlichen Schnittstelle zur Verfügung, die am Ende der Transaktion aufgerufen werden können, was die Sicherheit der Transaktionen und dementsprechend das Vertrauen in die Datenbasis erheblich steigert.
Dass man keine speziellen Konstrukte für die Modellierung der Extensionen verwenden muss, kann als ein weiterer Vorteil der Methode ins Feld geführt werden. Zudem widerspiegelt die lokale Extension häufig den tatsächlich betrachteten Sachverhalt und schränkt die zugrunde liegende Idee nur geringfügig ein, da über den Einstiegspunkt stets die globale Extension beschafft werden kann.
Ein Objekt kann aber nicht als Komponente von verschiedenen Objekten modelliert werden, was vordergründig als ein Nachteil erscheinen mag. Bei genauerer Betrachtung aber kann man sich durchaus daran gewöhnen, dass alle persistenten Instanzen, die zu einem gewissen Zeitpunkt in der Datenbasis zu finden sind, als Komponenten der Einstiegspunkte betrachtet werden müssen. Selbst wenn sich partout keine semantisch sinnvolle Komponentenbeziehung zwischen den verschiedenen beteiligten Klassen ausdenken lässt, kann immer die Strategie verfolgt werden, sämtliche Objekte als Komponenten der Klasse „Universum“ oder „System“ zu betrachten.
Auch wenn sich der Anwender der Methode DEIMOS über die starken Einschränkungen wundert, wird auf die Problematiken des Datenbankentwurfes sensibilisiert, was die Qualität seines Resultates sicherlich erhöhen wird.

TEIL II

OSWOOD

OSWOOD - **O_2 S**chema design **W**izard
based on Booch's **O**bject-**O**riented **D**esign

An expressive notation makes it possible to eliminate much of the tedium of checking the consistency and correctness of the decisions by using automated tolls. As a report by the Defense Science Board states, „Software development is and always will be a labor-intensive technology... Although our machines can do the dog-work and can help us keep track of our edifices, concept development is the quintessentially human activity... The part of software development that will not go away is the crafting of conceptual structures: the part that can go away is the labor of expressing them“ [Booch 94, pp. 171-172]

Auswahl des Zeichenwerkzeuges

Als Zeichenwerkzeug wurde das Programm Hardy der University of Edinburgh gewählt. Dieser Entscheid stützt sich auf die folgenden Vorteile von Hardy:

- Hardy bietet eine benutzerfreundliche Oberfläche für das Zeichnen der Diagramme. Die gängigen Operationen der Alineation von Elementen, der Verschiebung von Elementen unter Beibehaltung der Verbindungen, etc. werden allesamt unterstützt.
- Die Zeichnung von Diagrammen mit Hardy basiert auf Symbolbibliotheken und Diagrammdefinitionen. In den Symbolbibliotheken werden die Konstrukte definiert, während in den Diagrammdefinitionen deren Zusammenspiel, insbesondere die Einschränkungen in der Verwendung festgelegt sind. Vom Benutzer können sowohl Symbolbibliotheken als auch Diagrammdefinitionen erstellt werden. Hardy ist also erweiterbar.
- Die von Hardy erzeugten Dateien, seien dies Indexdateien, Diagrammdateien, Symbolbibliotheken oder Diagrammdefinitionen, sind allesamt ASCII-Dateien, die einem einfachen Syntax folgen, und eignen sich demnach für eine weitere Verarbeitung.
- Hardy ist auf den Plattformen Windows und UNIX verfügbar.
- Die Lizenzkosten sind insbesondere für die Universitäten valabel. Eine Demoversion mit geringen Einschränkungen gegenüber der Vollversion wird im Internet angeboten.
- Hardy wird an der Universität Zürich eingesetzt und von der Universität Bern evaluiert.

Auswahl der Entwicklungsumgebung

Die Entwicklung des Werkzeuges steht nicht im Zentrum der Arbeit. Aus diesem Grund wurde für die Implementierung eine Umgebung gewählt, welche die Minimalkriterien erfüllt und keinen grossen Aufwand für die Einarbeitung in Anspruch nimmt.
Die Wahl fiel auf die C++-Entwicklungsumgebung von Microsoft, nicht zuletzt darum, weil C++ eine anerkannte Programmiersprache ist, sondern weil diese Entwicklungsumgebung mit einer Reihe von komfortablen Hilfsmittel ausgestattet ist und über eine reichhaltige und sichere Klassenbibliothek verfügt (vgl. “Bewertung der Implementation” auf Seite 261).
Die Plattformunabhängigkeit stand also beim Entscheid nicht im Vordergrund (der gewählte Compiler kann nur Code für Win32 oder MacOS erzeugen). Vielmehr sollte der Weg der Portierung, sei es durch eine Neukompilation mit einem adäquaten Compiler auf UNIX oder sei es durch Transformation in die Programmiersprache Java, nicht verbaut sein.

5 OSWOOD

5.1 Einleitung

Das Werkzeug OSWOOD ist die Umsetzung der im vorigen Kapitel eingeführten Methode DEIMOS. Es soll den Entwerfer während des Analyse- und Entwurfsprozesses begleiten und dem Entwickler verschiedene Arbeiten durch die Generierung von Klassen- und Applikationsgerüsten abnehmen.

5.1.1 Die Aufgabe von OSWOOD

Die Aufgabe von OSWOOD ist also die Begleitung des Entstehungsprozesses einer Datenbankanwendung. Der Prozess beginnt dabei mit dem Start des Werkzeuges OSWOOD und dem anschliessenden Verzweigen in den Diagrammentwurfsmodus.
Mit Hilfe des Zeichenprogrammes Hardy [Hardy User 96] kann der Entwerfer die Diagramme gemäss der Methodenbeschreibung DEIMOS zeichnen. Nachdem die verschiedenen Ansichten des konzeptuellen Entwurfes der Anwendung einem gewissen Reifegrad entsprechen und gesichert sind, kehrt der Designer in die Hauptmaske von OSWOOD zurück.
In OSWOOD öffnet er die erstellte Indexdatei, welche das Hauptresultat seiner vorgelagerten Entwurfsanstrengung darstellt, und erhält eine Übersicht über die erstellten Diagramme, mit deren Hilfe er in die verschiedenen Ansichten navigieren kann. Die einzelnen wählbaren Diagrammansichten zeigen eine strukturierte und interpretierte Darstellung der entworfenen Sachverhalte und bieten eine Voransicht des aus den Entwurfsentscheidungen resultierenden Schemadefinitionscodes.
Werden bei der Interpretation der Hardydateien Verstösse gegen die in der Methode DEIMOS festgesetzten Verwendungsregeln der Konstrukte festgestellt, oder müssen Entwurfsentscheidungen aufgrund der resultierenden Implikationen überarbeitet werden, kann der Entwerfer in die Diagrammentwurfsansicht zurückkehren und entsprechende Anpassungen an den Diagrammen vornehmen. Die Änderungen werden in der Folge bei einer Neueröffnung der Index- oder Diagrammdateien in OSWOOD sichtbar.
Ist der Entwerfer schliesslich mit seiner Schema- und Applikationsdefinition zufrieden, kann er von OSWOOD die entsprechenden Schemadefinitionsdateien in O_2C generieren lassen. Damit hat er seine Arbeit mit OSWOOD beendet.
Die Schemadefinitionsdateien können nun in das Datenbanksystem O_2 importiert, validiert und nach Belieben angepasst werden.

5.1.2 Starten von OSWOOD

Die Installation von OSWOOD kann dem Anhang entnommen werden.

Nach der erfolgten Installation (vgl. “Installation von OSWOOD”, Seite 266) wird OSWOOD durch das Klicken auf die Programmikone gestartet.
Es erscheint das folgende Hauptfenster:

Abbildung 5-1: Hauptfenster von OSWOOD

5.1.2.1 OSWOOD Bildschirmbereiche

Das Hauptfenster von OSWOOD ist folgendermassen aufgeteilt (von oben nach unten):

Titelzeile

enthält die Standardelemente von Windows sowie den Namen des Programmes. Ist ein Diagrammfenster geöffnet und maximiert, trägt die Titelzeile - wie dies in allen Applikationen, welche die MDI-Architektur implementieren, der Fall ist - zusätzlich die Informationen über das Dokumentenfenster.

Menüleiste

zeigt das für einen bestimmten Kontext (aktives Fenster) gültige Menü an.

Werzeugleiste

bietet einen schnellen Zugriff mit der Maus auf häufig verwendete Aktionen. Sämtliche Kommandos sind auch im Menü enthalten.

Dokumentenbereich

In diesem Bereich werden die verschiedenen Diagrammfenster angezeigt.

Ausgabebereich

informiert während einer aufwendigen Aktion (z.B. Interpretation einer Diagrammdatei über den Fortschritt der Ausführung und etwaige Unzulänglichkeiten oder Fehler. Die ausgegebenen Texte bleiben bis zu der nächsten vom Benutzer ausgelösten längeren Verarbeitung im Fenster erhalten und können allesamt eingesehen werden.

Statuszeile

Die Statuszeile liefert Kurzinformationen über den Verarbeitungsstatus oder zeigt gegebenenfalls Hinweise zu Menüpunkten an.

Alle diese Bereiche sind während der ganzen OSWOOD-Sitzung sichtbar und werden insbesondere von der aktuell geöffneten Datei nicht beeinflusst.

5.1.2.2 OSWOOD Benutzeraktionen

OSWOOD kann über Menüs und die Werkzeugleiste gesteuert werden. Die Menüs im Hauptfenster lösen die folgenden Aktionen aus

Menü **File**

Nach dem Starten von OSWOOD zeigt sich kein offenes Fenster.

In diesem Menü wird nun die Möglichkeit geboten, Indexdateien zu öffnen. Der Benutzer kann auch eine der vier letzten Dateien, welche in OSWOOD geöffnet worden sind, aus dem Menü auswählen.

Durch Selektion der Auswahl Exit kann die Sitzung beendet werden.

Menü **View**

Mit Hilfe der Aktion Hardy kann in die Diagrammentwurfumgebung gewechselt werden.

Das Ein- und Ausschalten der Anzeige der Werkzeugleiste und der Statuszeile wird ebenfalls in diesem Menü behandelt.

Menü **Help**

Durch die Wahl dieser Aktion wird ein Informationsfenster mit Angaben zur Version und zu den Autoren angezeigt.

5.1.2.3 Starten von Hardy

Die Installationsanleitung von Hardy findet sich im Anhang “Installation”, Seite 261.

Hardy wird durch ein Klicken in der Werkzeugleiste auf das Hardysymbol oder Wahl der Menüaktion **View | Hardy** gestartet.

5.2 Schemaentwurf mit Hardy

Hardy ist ein Zeichnungswerkzeug, welches an der Universität Edinburgh entwickelt worden ist. Es zeichnet sich neben seiner hohen Bedienerfreundlichkeit vor allem durch seine Fähigkeit zum selbständigen Erstellen von Symbolbibliotheken aus.

Das Programm ist urheberrechtlich geschützt und lizenzpflichtig. Die Anbieter stellen aber eine eingeschränkte Version gratis und zur freien Verwendung zur Verfügung. Diese unterscheidet sich von der Vollversion durch die Limitierung der maximalen Anzahl Symbole in einem Diagramm und durch starke Einschränkungen in der Erstellung von benutzerdefinierten Symbolbibliotheken.

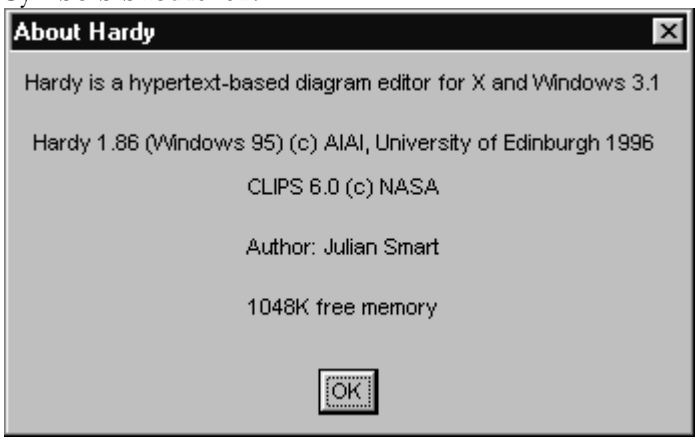


Abbildung 5-2: Copyright Informationen von Hardy

5.2.1 Arbeiten mit Hardy

5.2.1.1 Hardy Oberfläche

Nach dem Start von Hardy präsentiert sich das Hauptfenster von Hardy wie folgt:

Abbildung 5-3: Hauptfenster von Hardy

5.2.1.2 Hardy Benutzeraktionen

Hardy kann über Menüs und Toolbarbuttons gesteuert werden. Die Menüs im Hauptfenster lösen die folgenden Aktionen aus:

Menü **File**

Hardy bietet nach dem Starten bereits eine leere Indexdatei an, in welcher neue Diagramme aufgenommen werden können. Soll an einem bereits erstellten Entwurf weiter gearbeitet werden, kann dieser durch die Aktion **Open Index File** geöffnet werden.

Im Weiteren bietet dieses Menü Aktionen zum Speichern und Drucken der Arbeit.

Durch Selektion der Auswahl Exit kann die Sitzung beendet werden. Hardy erkundigt sich, ob die bearbeiteten Diagramme zu speichern sind, falls diese nicht schon gesichert worden sind.

Menü **Cards**

Dieses Menü bietet die Aktionen für die Manipulation von (Diagramm-) Karten an. Insbesondere erlaubt es durch **Create Top Card** die Erzeugung einer neuen Hauptkarte. Auch Funktionen zum Auffinden bestimmter Karten oder zum Neuzeichnen einer Ansicht sind in dieser Auswahl enthalten.

Menü **Tools**

Mit Hilfe dieses Menüteils werden die Diagrammtypen und Symbolbibliotheken verwaltet.

Zudem können durch die Wahl von **Preferences** gewisse Standardeinstellungen verändert werden, welche das Verhalten von Hardy im Allgemeinen beeinflussen.

Insbesondere kann die Registrierung vorgenommen werden (vgl. "Registrierung", Seite 263).

Menü **Window**

Bei diesem Menü handelt es sich um ein Standardmenü für die Navigation durch die verschiedenen Fenster und deren Anordnung innerhalb der Mehrfensteransicht, welches in nahezu allen Windows-Anwendungen finden lässt.

Menü **Help**

Mit dem Hilfemenü kann in die mitgelieferte Hilfedatei verzweigt werden, in welcher z.B. durch Suche im Index Information über einen bestimmten unklaren Begriff beschafft werden.

Zudem kann durch die Wahl von **About Hardy** der Herstellerdialog (vgl. Abbildung 5-2, Seite 128) eingesehen werden.

5.2.2 Schemaentwurf mit Hardy

5.2.2.1 Index und Karten

Ein Entwurf im Allgemeinen und ein Schemaentwurf mit DEIMOS im Speziellen besteht aus mehreren Diagrammen. Diese Diagrammsammlungen werden innerhalb von Hardy als Index verwaltet. Die einzelnen Diagramme hingegen werden als sogenannte Karten aufgefasst. Dabei gilt, dass jedes Diagramm durch mindestens eine Karte repräsentiert wird. Lässt man das Expandieren von Metakonstrukten wie z.B. Subdiagrammen zu, so kann sich ein Diagramm auch über mehrere Karten erstrecken.

Zudem fordert das Entwurfswerkzeug die hierarchische Strukturierung der einzelnen Ansichten. Dies bedeutet, dass man angehalten ist, eine Oberkarte zu erfassen (Topcard) und von dieser aus Verknüpfungen zu weiteren Unterkarten (Subcards) erstellt.

Somit muss man während der Entwurfsarbeit streng zwischen Index und Karten unterscheiden. Der Index enthält die Information, welche Diagramme in der Sammlung enthalten und wie diese untereinander verbunden sind, und stellt dem Benutzer an der Oberfläche einen Baum für die Navigation durch die Karten zur Verfügung. Die Karte hingegen ist eine Ansicht eines Diagramms und wird somit am Bildschirm in Form eines eigenen Fensters angezeigt.

5.2.2.2 Dateistruktur von Hardy

Hardy kennt also zwei Arten von Dateien: die Indexdatei und Diagrammdateien. Die Indexdatei speichert die Information über die enthaltenen Karten, insbesondere die entsprechenden Dateinamen, während in den Diagrammdateien sämtliche Informationen über die gezeichneten Elemente abgelegt sind. Elementinformationen sind neben den graphischen Angaben für deren Repräsentation am Bildschirm auch die diagrammspezifischen Daten über Name und Typ des Elementes und die vom Benutzer zusätzlich erfassten Eigenschaften.

5.2.3 Schemaentwurf mit DEIMOS

Als erstes muss eine Hauptkarte erzeugt werden. Dies wird durch die Wahl des Menüpfades **Card | Create Top Card** initiiert. Hardy bietet in einem Dialog sämtliche zur Verfügung stehenden Kartentypen an, welche als Hauptkarten in Frage kommen können.

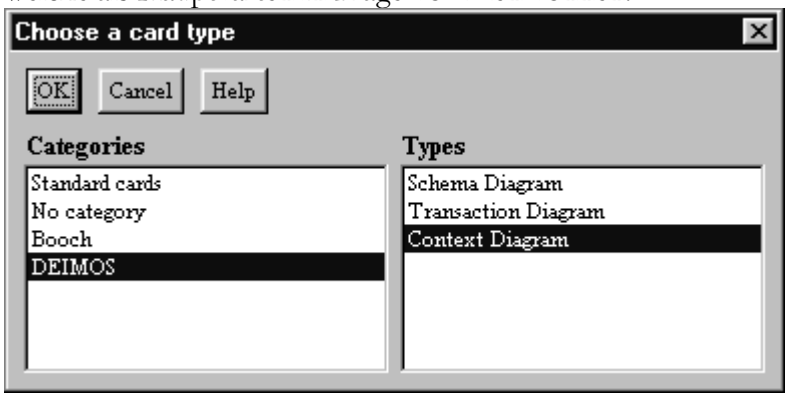


Abbildung 5-4: Die Diagrammtypen von DEIMOS

DEIMOS schlägt vor, als Hauptkarte das Kontextdiagramm zu wählen.

5.2.4 Erstellen eines Kontextdiagramms

Nach der Auswahl des Kontextdiagrammes als Hauptkarte wird der folgende Bildschirm dargestellt.

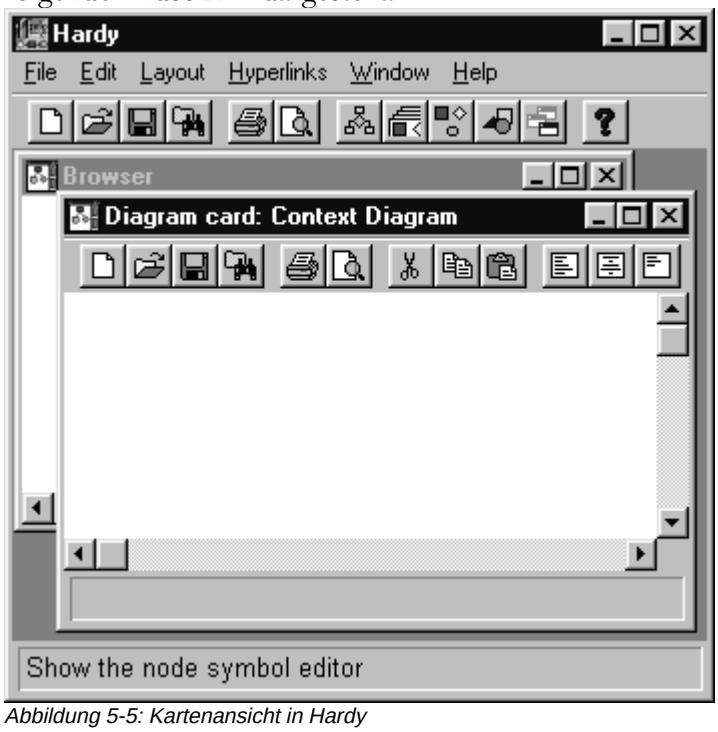


Abbildung 5-5: Kartenansicht in Hardy

Die Menüs werden in dieser Ansicht neu organisiert und lösen die folgenden Aktionen aus:

Menü **File**

bietet Aktionen für das Speichern, Öffnen und Drucken der Diagrammkarte an.

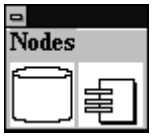
Die Diagrammansicht kann durch die Wahl von **Quit Card** verlassen werden.

Menü **Edit**

Beinhaltet Aktionen für das Selektieren, Deselektieren, Kopieren,

Ausschneiden und Einfügen von Diagrammelementen.
Elemente können über dieses Menü formatiert werden, indem beispielsweise die Beschriftungen, Eigenschaften, Schriftarten, usw. geändert werden.
Soll ein bestimmtes Element zu einer neuen Ansicht expandiert werden, kann der Punkte **New Expansion** gewählt werden.
Manü **Layout**
Erlaubt die Allinierung mehrerer selektierter Elemente nach bestimmten Kriterien. Zudem ist in diesem Menü die Vergrösserungs-, bzw. Verkleinerungsmöglichkeit (**Zoom**) implementiert.
Menü **Hyperlinks**
Sprünge zu anderen Elementen oder anderen Ansichten können mit Hilfe dieses Menüs konfiguriert werden. Darüber hinaus lassen sich verschiedene Kontrollelemente wie etwa die Palette oder die Werkzeugleiste ein-, bzw. ausblenden.
Die weiteren Menüs **Window** und **Help** unterscheiden sich nicht von denjenigen des Hauptfensters.

5.2.4.1 Zeichenelemente



Gleich nach dem Aufbau des leeren Diagrammfensters wird eine Palette angezeigt, welche die zur Auswahl stehenden grafischen Konstrukte für dieses Diagramm anbietet.

Ein gewünschtes Element wird in zwei Schritten auf das Zeichenblatt übertragen:
1. Selektion des Elementes mit der linken Maustaste auf der Palette.
2. Positionierung durch einfachen Mausklick mit der linken Maustaste auf der Diagrammkarte.
Elemente, die sich bereits auf dem Zeichenblatt befinden, können durch die üblichen Mausaktionen oder durch Menübefehle verschoben, in der Grösse angepasst oder gelöscht werden. Zudem bietet Hardy eine Reihe von Funktionen an, die in der Werkzeugliste oder aus dem Menü angesprochen werden können, um ein oder mehrere Elemente gegeneinander oder auf dem Blatt auszurichten.

5.2.4.2 Knoten

Für die Erstellung eines Kontextdiagramms stehen die folgenden Knoten zur Auswahl:

	Konstrukt	Eigenschaften
	Schemadiagramm	`Name` beliebige Beziechung
	Transaktionsdiagram	`Name` beliebige Bezeichung

Tabelle 5-1: Knoten in einem Kontextdiagramm

Das Kontextdiagramm steht ausschliesslich als Übersichtsdiagramm, da dessen Informationen die Definition des Schemas nicht beeinflusst. Es ist also nicht nötig, die einzelnen Knoten durch Kanten zu verbinden. Aus diesem

Grund werden auf dieser Diagrammpalette auch keine Kantenkonstrukte angeboten.

5.2.4.3 Beispiel FIS

Im konkreten Fall könnte ein Kontextdiagramm wie folgt aussehen:

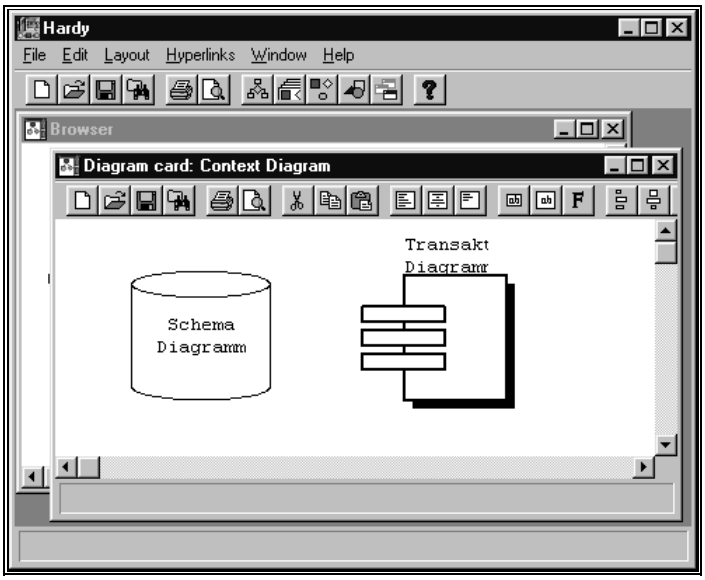
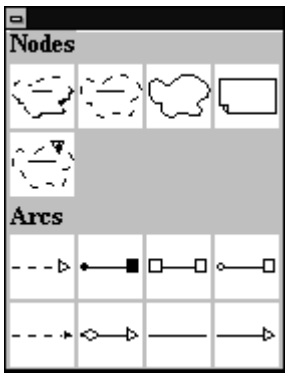


Abbildung 5-6: Beispiel eines Kontextdiagramms

5.2.5 Erstellen eines Schemadiagramms

Um ein Schemadiagramm zu erstellen, wird von DEIMOS vorgeschlagen, einen Knoten vom Typ Schemadiagramm zu expandieren. Dazu muss dem Menüpfad **Hyperlinks | Hyperlink New Card** gefolgt und im anschliessend angezeigten Dialog (vlg. Abbildung 5-4, Seite 130) den DEIMOS-Typ **Schema Diagram** gewählt werden.

5.2.5.1 Zeichenelemente



Die in einem Schemadiagramm zur Verfügung stehenden Zeichenelemente werden auf der nebenstehenden Palette angezeigt. Die Auswahl ist in Knoten (Elemente) und Kanten (Elementverbinder) unterteilt.

Das Positionieren der Konstrukte auf dem Zeichenfenster wurde bereits im vorigen Abschnitt erläutert, während die beiden notwendigen Schritte für das Einrichten einer Verbindung zwischen zwei Knoten kurz erklärt werden soll.

1. Selektion des gewünschten Kantentyps auf der Palette.
2. Verbindung von Knoten durch Ziehen der Maus, während die rechte Maustaste gedrückt bleibt (Drag&Drop).

5.2.5.2 Knoten

Für die Erstellung eines Schemadiagramms stehen die folgenden Knoten zur Auswahl:

	Konstrukt	Attribute
	Datenbankklasse	`class name` Name der Klasse `properties` Eigenschaften

	Hilfsklasse	`class name` `properties`	Name der Klasse Eigenschaften
	abstrakte Klasse	`class name` `properties`	Name der Klasse Eigenschaften
	Einstiegspunkt	`name`	Persistenter Name
	Notiz	`note`	Notiz

Tabelle 5-2: Knotenkonstrukte in einem Schemadiagramm

Die angegebenen Klassennamen müssen diagrammweit eindeutig sein. Die Eigenschaften (`properties`) einer Klasse beschreiben Methoden, Attribute, Schlüssel und Konsistenzbedingungen. Jede Definition muss auf einer neuen Zeile im Eingabefeld für das Knotenattribut gesetzt werden.

Damit OSWOOD so viel Information wie möglich aus diesen Angaben herauslesen und für die Umsetzung (vgl. "Umsetzung eines Schemadiagramms", Seite 153) verwenden kann, sollen die Klasseneigenschaften einem bestimmten Syntax folgen.

Methodendefinitionen sind in der Form

`[W|R] Name([ParamName:ParamType[, ParamName:ParamType]*])[:Type]`

zu erfassen. Wird auf die Angabe des Zugriffstyps (W=Write, R=Read) verzichtet, wird eine lesende Methode angenommen. Lässt man die Definition des Typs weg, nimmt OSWOOD an, diese Funktion retourniere einen bool'schen Wert.

Die explizit definierten Attribute sollen in der Form

`AttributeName[:AttributeType[=[Default]]]`

aufgeschrieben sein. Das Weglassen des Typs führt zur standardmässigen Definition als Zeichenkette. Gibt man einen Standardwert für das Attribut an, wird OSWOOD dafür sorgen, dass es bei der Erzeugung eines Objektes auf diesen Wert initialisiert wird. Wird der Wert weggelassen, aber das Gleichheitszeichen angegeben, so ist dieses Attribut als initialisierungsnotwendig gekennzeichnet. Ein Objekt kann also nur noch dann erzeugt werden, wenn gleichzeitig - in der impliziten Init-Methode - diesem Attribut ein Wert zugewiesen wird.

Sollen lokale Extensionen bezüglich eines Schlüsselkriteriums eindeutig definiert werden, kann dies durch die Angabe eines Schlüssels der Form

`ClassName<AttributeName [, AttributeName]*>`

definiert werden. Dabei muss sichergestellt sein, dass die Extension zu der angegebenen Klasse `ClassName` auch tatsächlich eine Komponentenbeziehung unterhält. Sind die im Schlüsselkriterium enthaltenen Attribute in der Komponente nicht enthalten, werden sie von OSWOOD implizit erzeugt.

Invariante Konsistenzbedingungen werden in der Form

`{ConstraintString}`

formuliert. OSWOOD verpackt bei der Umsetzung die angegebene Bedingung in eine `if`-Anweisung, ohne weitere Validierungen wie Typenprüfung, etc. durchzuführen.

5.2.5.3 Kanten

Für die Erstellung eines Schemadiagramms stehen die folgenden Kanten zur Auswahl:

	Konstrukt	Attribute	
	Vererbung	keine	
	Instantiierung	keine	

	Komponenten-beziehung	`name`
		`cardinality`
		`inverse`
			Bezeichnung
			Kardianlitätsangabe
			rückbezügliche Kard.angabe

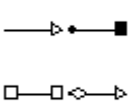
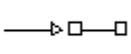
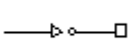
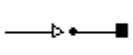
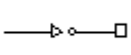
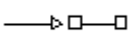
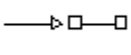
	inverse Beziehung	`name`
		`begin`
		`end`
			Bezeichnung
			Kardianlitätsangabe
			rückbezügliche Kard.angabe

	Beobachtungs-beziehung	`name` `cardinality`	Bezeichnung Kardianlitätsangabe
	Evolution	`function`	Name der Evolutionsmethode
	Notizbezug	keine	

Tabelle 5-3: Kantenkonstrukte in einem Schemadiagramm

5.2.5.4 Verbindungsmatrix

Das Zeichnungswerkzeug Hardy bietet bereits verschiedene vorgegebene und definierbare Einschränkungen für das Zeichnen von Kanten.
Zu den vorgegebenen Einschränkungen zählt, dass eine Verbindung ausschliesslich zwischen zwei Knoten eingerichtet werden und zudem zwischen zwei Knoten nur genau eine Verbindung eines gewissen Typs gezeichnet werden kann.
Bei der Definition von Diagrammtypen ist dem Benutzer die Möglichkeit gegeben, Einschränkungen für die Verwendung eines bestimmten Kantenkonstruktes zu erfassen. Für das Schemadiagramm seien diese Einsatzregeln in einer Matrix zusammengefasst:

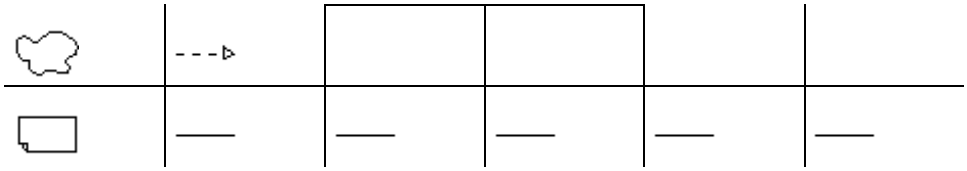


Tabelle 5-4: Matrix der möglichen Knotenverbindungen in einem Schemadiagramm

Aus dieser Darstellung lässt sich leicht erkennen, dass Vererbung zwischen allen Typen von Klassen zugelassen ist. Die Instantiierung ist einzig für die Verbindung eines Einstiegspunktes mit einer Datenbankklasse vorgesehen und darf nie als Ziel einer Verbindung auftreten. Die Notizknoten sind immer der Ausgangspunkt eines Notizbezuges, es sei denn, es würden Notizen kommentiert.

5.2.5.5 Beispiel FIS

Im konkreten Fall könnte ein Schemadiagramm wie folgt aussehen:

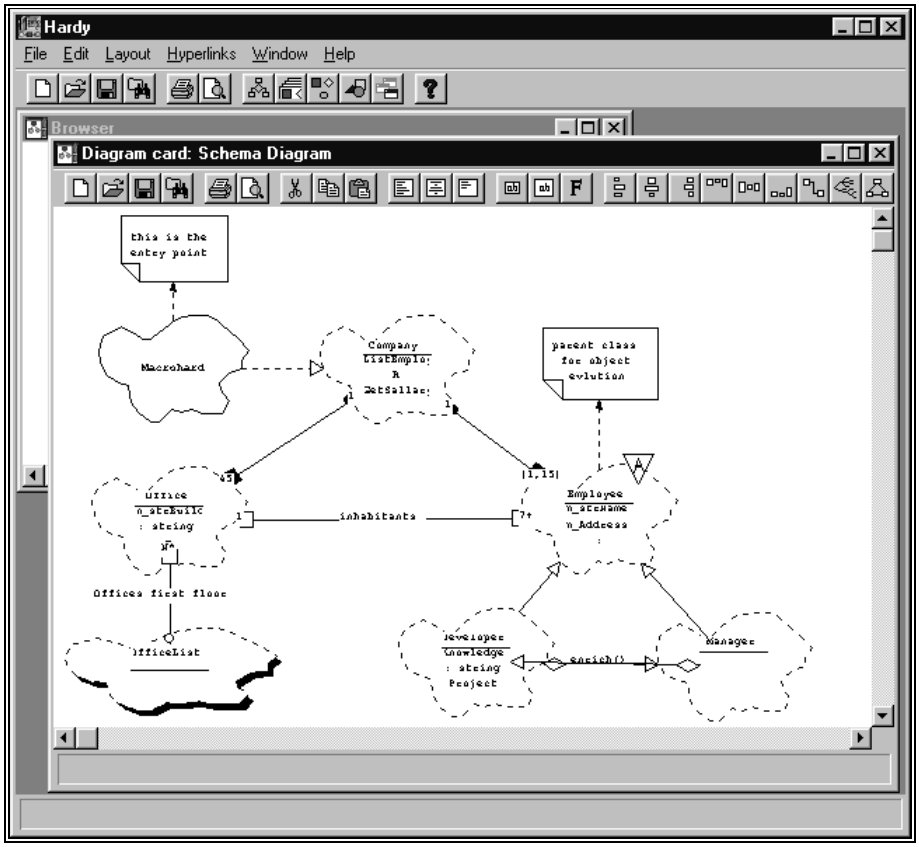
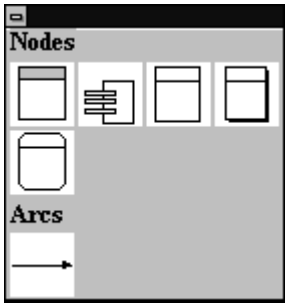


Abbildung 5-7: Beispiel eines Schemadiagramms

5.2.6 Erstellen eines Transaktionsdiagramms

Um ein Schemadiagramm zu erstellen, wird von DEIMOS vorgeschlagen, innerhalb eines Kontextdiagrammes einen Knoten vom Typ Transaktionsdiagramm zu expandieren. Dazu muss innerhalb des Kontextdiagramms ein entsprechendes Element selektiert, dem Menüpfad **Hyperlinks | Hyperlink New Card** gefolgt und im anschliessend angezeigten Dialog (vlg. Abbildung 5-4, Seite 130) den DEIMOS-Typ **Transaction Diagram** gewählt werden.

5.2.6.1 Zeichenelemente



Die in einem Schemadiagramm zur Verfügung stehenden Zeichenelemente werden auf der nebenstehenden Palette angezeigt. Die Auswahl ist in Knoten und Kanten unterteilt.

5.2.6.2 Knoten

Für die Erstellung eines Transaktionsdiagramms stehen die folgenden Knoten zur Auswahl:

	Konstrukt	Attribute	
	Hauptprogramm	`name`	Hauptprogrammname
	Subsystem	`name`	Name des Subsystems
	Programm	`name`	Name des Programmes
		`pseudo code`	Textuelle Ablaufbeschreib.
	Funktion	`name (params) return`	Funktionsname
		`pseudo code`	Textuelle Ablaufbeschreib.
	Transaktion	`name (params) return`	Transaktionsname
		`pseudo code`	Textuelle Ablaufbeschreib.

Tabelle 5-5: Knotenkonstrukte in einem Transaktionsdiagramm

Die angegebenen Namen der einzelnen Konstrukte müssen diagrammweit eindeutig sein. Bei Funktionen und Transaktionen werden in den Namen die Aufrufkonventionen mitangegeben. Diese müssen in der Form

Name([*ParamName*:*ParamType*[, *ParamName*:*ParamType*]*])[:*Type*]

notiert sein.

5.2.6.3 Kanten

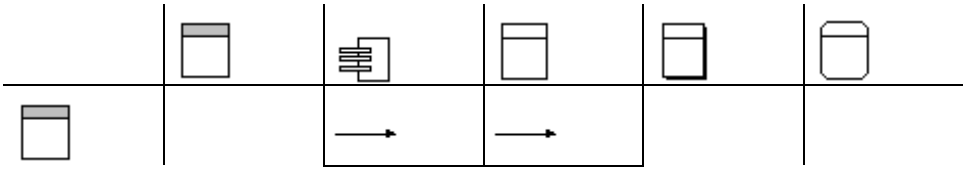
Für die Erstellung eines Transaktionsdiagramms steht lediglich eine Kante zur Auswahl:

	Konstrukt	Attribute	
	Aufruf	`number`	Sequenznummer für die Aufrufreihenfolge

Tabelle 5-6: Kantenkonstrukte in einem Transaktionsdiagramm

5.2.6.4 Verbindungsmatrix

Die Elemente eines Transaktionsdiagramms können unter Berücksichtigung der bereits erwähnten Einschränkungen für das Zeichnen von Verbindungslinien zwischen Knoten gemäss der folgenden Matrix miteinander verbunden werden:



			→		
				→	→
				→	→

Tabelle 5-7: Matrix der möglichen Knotenverbindungen in einem Transaktionsdiagramm

Bei der genaueren Betrachtung der Verbindungsmatrix fällt auf, dass keine Aufrufe zu Hauptprogrammen zugelassen sind. Möchte man trotzdem mehrere Hauptprogramme entwerfen, so muss man je eine Diagrammkarte als Expansion eines Transaktionsknotens im Kontextdiagramm zu Hilfe nehmen.

Zudem lässt sich aus der Matrix erkennen, dass von Transaktionen aus keine weiteren Verbindungen mehr erlaubt sind. Die Transaktion bildet also in dieser Ansicht das Blatt des Aufrufbaumes.

5.2.6.5 Beispiel FIS

Im konkreten Fall könnte ein Transaktionsdiagramm, welches verschiedene Subsysteme enthält, wie folgt aussehen:

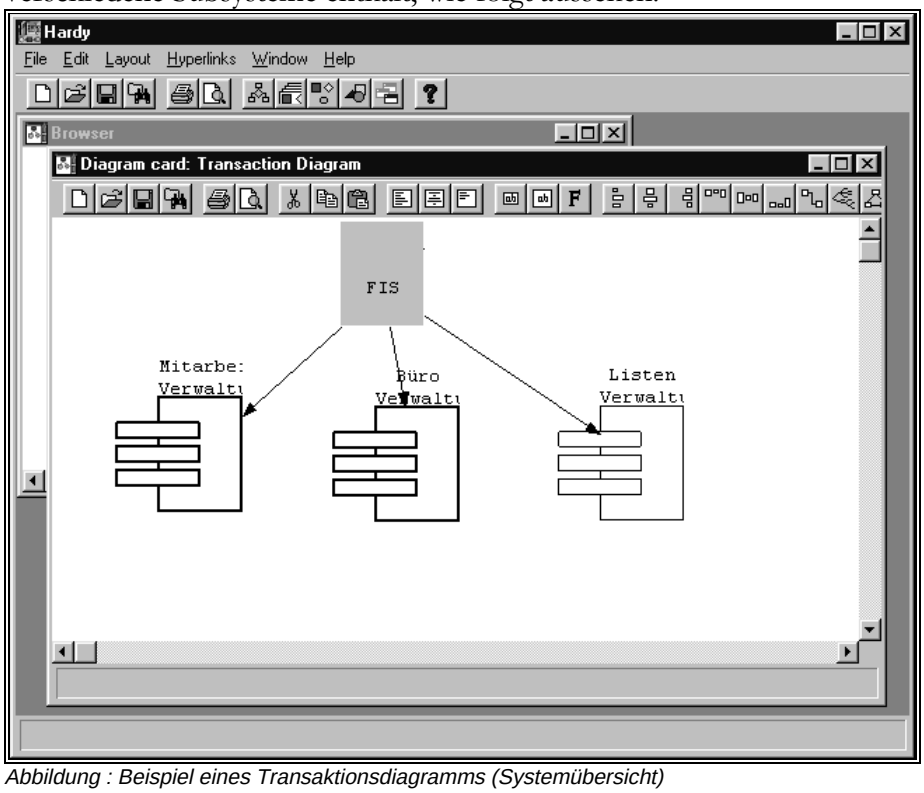


Abbildung : Beispiel eines Transaktionsdiagramms (Systemübersicht)

Die etwas stärkere Strichbreite, mit welcher ein Element gezeichnet ist, weist darauf hin, dass es sich um ein expandierbares Objekt handelt. Die Expansion für das Subsystem „Mitarbeiter Verwaltung“ wird in der nächsten Abbildung veranschaulicht:

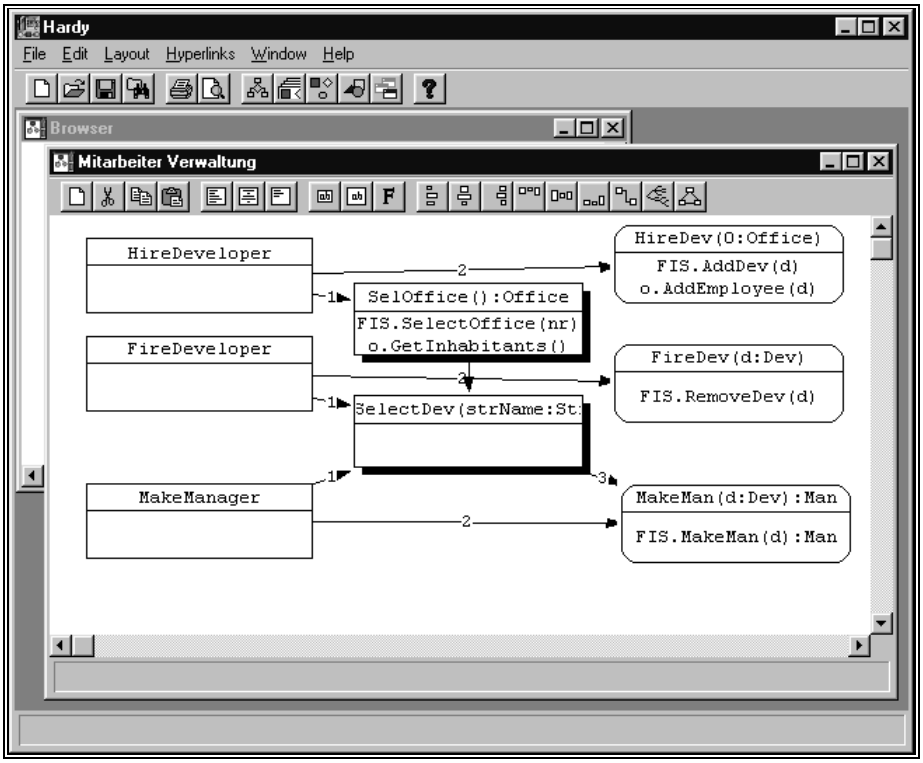


Abbildung 5-8: Beispiel eines Transaktionsdiagramms (Subsystemexpandsion)

5.3 Code generieren mit OSWOOD

Nachdem das Datenbankschema durch verschiedene Diagramme entworfen ist, sollen mit Hilfe von OSWOOD die entsprechenden Anweisungen in der Schemadefinitionssprache generiert werden.
Zu diesem Zweck müssen die erstellten Diagrammdateien in OSWOOD geöffnet werden, was implizit zu der Interpretation der gezeichneten Information führt, und durch Speicherung als ODL-Datei in Schemadefinitionssprache überführt werden. Geöffnet werden können zum einen direkt die von Hardy erzeugten Diagrammdateien (*.dia) oder zum anderen die Indexdateien, von denen aus durch die verschiedenen Diagrammtypen navigiert werden kann.
Nachfolgend sei der Ablauf beschreiben, welcher von der Annäherung mittels der Indexinformation ausgeht und seines Top-Down-Ansatzes wegens auch gegenüber den direkten Ansätzen vorgezogen werden sollte.

5.3.1 Anzeige der Indexdatei

Durch die Wahl der Schaltfläche  aus der Werkzeugleiste oder durch die Selektion des Menüpfades **File | Open** wird der folgende Dialog zur Auswahl der gewünschten Indexdatei angezeigt:

Abbildung 5-9: Auswahl einer Indexdatei in OSWOOD

Nach der Wahl einer bestehenden Indexdatei wird diese interpretiert und der folgende Indexbaum - ähnlich der Navigationsinformation in Hardy - angezeigt:

Abbildung 5-10: Darstellung einer Indexdatei von Hardy in OSWOOD

OSWOOD listet baumartig sämtliche Diagramme auf, welche in der gegebenen Indexdatei gefunden werden konnten. Es werden also auch Diagramme angezeigt, die nicht in der Methode DEIMOS vorgesehen sind.

Für die Darstellung werden die folgenden Symbole verwendet:

	Schemadiagramm
	Transaktionsdiagramm
	Kontextdiagramm oder nicht bekanntes Diagramm

Tabelle 5-8: Diagrammtypen in einer Indexdatei

5.3.1.1 Inhalt einer Indexdatei

Eine Indexdatei besteht in der Regel aus

- genau 1 Kontextdiagramm,
- genau 1 Schemadiagramm,
- genau 1 Transaktionsdiagramm. Dieses kann durch Subsystemexpansion aus beliebig vielen verschachtelten Transaktionsdiagrammen bestehen.

Das Kontextdiagramm ist der Vater der anderen Diagramme, die Verweise auf die Subdiagramme sind aber nicht in der Kontextdiagrammdatei gespeichert, sondern in der Indexdiagrammdatei.

5.3.1.2 Benutzeraktionen

Durch einen Doppelklick mit der linken Maustaste über einem Baumelement, lassen sich die entsprechenden Subdiagramme näher betrachten. OSWOOD implementiert nur die Ansichten für die Schema- und die Transaktionsdiagramme. Weitere Diagrammtypen - insbesondere auch das Kontextdiagramm - können nicht

interpretiert werden, weshalb sie in der Indexansicht durch eine Fragezeichenikone dargestellt sind.

5.3.2 Anzeige des Schemadiagramms

Ist innerhalb der Navigationsansicht der Indexdatei ein Schemadiagramm gewählt oder durch die **File | Open** Aktion geöffnet worden, wird die Information der Diagrammdatei untersucht und interpretiert.
Während der Interpretation werden verschiedene Validierungen durchgeführt. Zu unterscheiden sind dabei Konditionen, welche die methodische Korrektheit des Diagramms im Sinne von DEIMOS prüfen und Validierungen, welche auf die vom Entwerfer erfassten Attribute ausgeführt werden. Verletzt ein Diagramm eine zwingende Anforderung von DEIMOS - beispielsweise, wenn ein Zyklus in der Vererbungshierarchie entdeckt wird - wird die Verarbeitung unter Ausgabe einer entsprechenden Benutzerbenachrichtigung unverzüglich abgebrochen, während bei einem syntaktischen Fehler in beispielsweise einer Attributdefinition lediglich als Warnung im Ausgabebereich des Hauptfensters angezeigt wird. Die Sachverhalte, welche zu Fehlermeldungen oder Warnungen führen, werden im Implementationskapitel (vgl. “Implementation”, Seite 189) eingehend diskutiert.
Nach vollständiger und fehlerfreien Verarbeitung wird die Datei im Schemadiagrammbildschirm dargestellt.

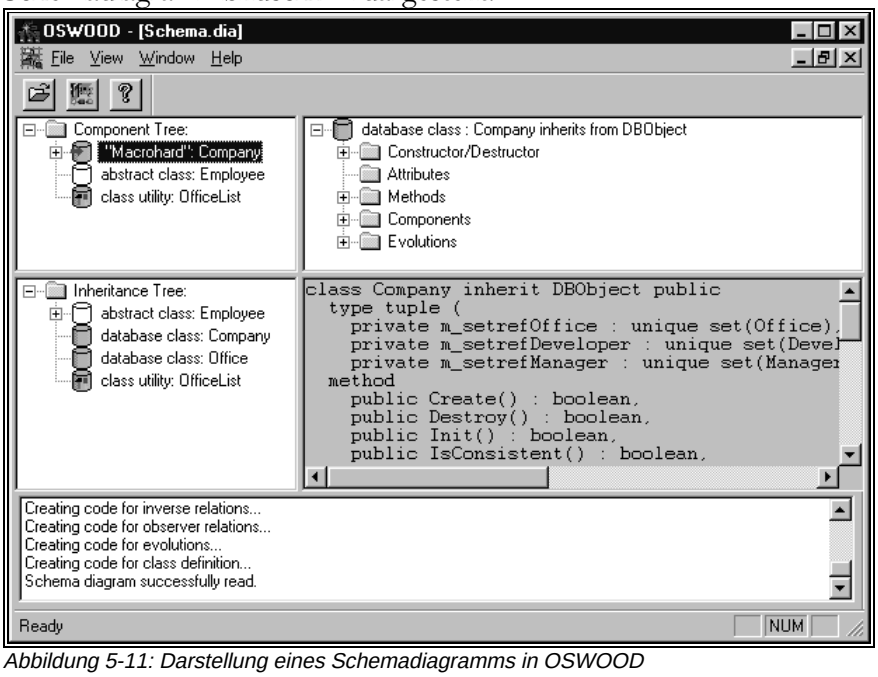


Abbildung 5-11: Darstellung eines Schemadiagramms in OSWOOD

Neben den für alle Ansichten gemeinsamen Bildschirmbereichen (vgl. “OSWOOD Bildschirmbereiche”, Seite 126) werden für die Anzeige einer Schemadiagrammdatei vier weitere eröffnet. Dies sind (von oben links nach unten rechts):

- Komponentenbaum,
- Vererbungshierarchie,
- Klassendarstellung,
- ODL-Ansicht.

Die einzelnen Ansichten lassen sich in der Grösse anpassen, während ihre Position auf dem Bildschirm nicht geändert werden kann.

5.3.2.1 Der Komponentenbaum

In der Komponentenansicht werden die im Schema definierten Klassen bezüglich ihrer Komponentenbeziehungen visualisiert. Diese Ansicht ist vor allem für die Untersuchung der Persistenzfortpflanzung von grosser Bedeutung.

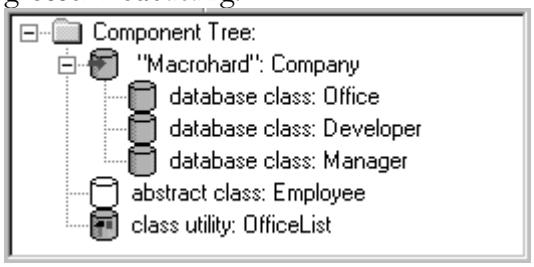


Abbildung 5-12: Komponentenbaum eines Schemadiagrammes

Jede im Schemadiagramm enthaltene Klasse wird in diese Ansicht aufgenommen und mit einem typenspezifischen Symbol charakterisiert. Die einzelnen Symbole sind wie folgt den Basistypen zugeordnet:

	Explizit definierter Einstiegspunkt, als Instanz einer Datenbankklasse
	Datenbankklasse
	Abstrakte Klasse
	Hilfsklasse

Tabelle 5-9: Die Klassensymbole im Komponentenbaum

Die Methode DEIMOS fordert, dass jede Klasse, die persistente Instanzen ausbilden kann, als Komponente des expliziten Einstiegspunktes zu definieren ist. Hilfsklassen oder abstrakte Klassen dürfen demnach nicht als Subelemente des Einstiegspunktes auftreten. Dies lässt sich dank der baumartigen Darstellung der Komponentenansicht leicht überprüfen.

5.3.2.2 Die Vererbungshierarchie

In der Vererbungshierarchie sind die Vererbungsbeziehungen zwischen den verschiedenen Klassen baumartig dargestellt.

Abbildung 5-13: Vererbungshierarchie in einem Schemadiagramm

Da in dem betrachteten Datenbanksystem Mehrfachvererbung zugelassen ist, muss die Vererbungshierarchie als Graph angesehen werden. Der Graph jedoch ist gerichtet und zyklenfrei, weshalb er trotzdem als Baum, in welchem ein bestimmtes an einer Mehrfachvererbung beteiligtes Element an mehreren Positionen auftritt, dargestellt werden kann.

5.3.2.3 Die Klassendarstellung

In der Klassenansicht werden die Eigenschaften einer bestimmten sich im Fokus der Betrachtung befindenden Klasse dargestellt.

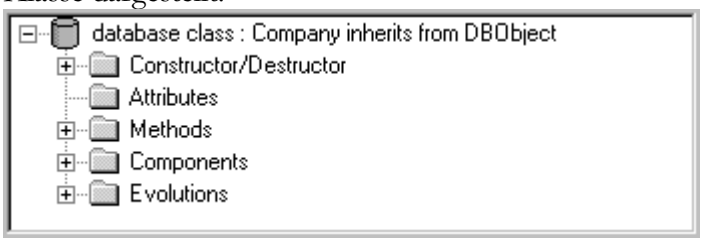


Abbildung 5-14: Klassenansicht in einem Schemadiagramm

Die Ansicht ist in verschiedene Interessensbereiche strukturiert und enthält die folgenden Abschnitte:

- Konstruktion/Destruktion,
- Attribute,
- Methoden,
- Komponentenbeziehungen,
- Inverse Beziehungen,
- Evolutionsbeziehungen,
- Beobachtungsbeziehungen.

Jede dieser Sektion enthält die in diesem Zusammenhang relevanten Methoden und Attribute. Für die Darstellung der Elemente werden die folgenden Symbole verwendet:

	Explizites öffentliches Attribut
	Explizites privates Attribut
	Explizites Schlüsselattribut
	Implizites privates Attribut
	Explizite öffentliche Methode
	Explizite private Methode
	Explizite öffentliche nur lesende Methode
	Schlüsselmethode
	Implizite öffentliche Methode
	Implizite private Methode
	Implizite öffentliche nur lesende Methode
	Virtuelle öffentliche Methode
	Virtuelle private Methode
	Virtuelle öffentliche nur lesende Methode
	Komponentenbeziehung
	Beobachtungsbeziehung
	Inverse Beziehung
	Evolutionsbeziehung

Tabelle 5-10: Symbole ein einer Klassenansicht

5.3.2.4 Konstruktion/Destruktion

Die drei in dieser Sektion aufgeführten Methoden werden von OSWOOD generiert und zählen zu den virtuellen öffentlichen Methoden.

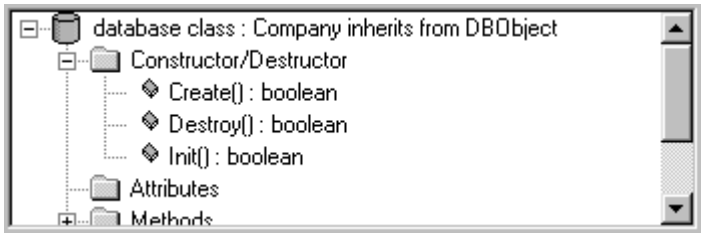


Abbildung 5-15: Darstellung der Konstruktoren und Destruktoren

5.3.2.5 Attribute

Der folgende Abschnitt zeigt die vom Entwerfer definierten und somit öffentlichen Attribute. Zum einen wurden diese direkt im Hardydiagramm als Eigenschaften der Klasse angegeben und zum anderen durch die Definition von Schlüsselkriterien in der bezüglich der Komponentenbeziehung als Besitzer deklarierten Klasse impliziert.

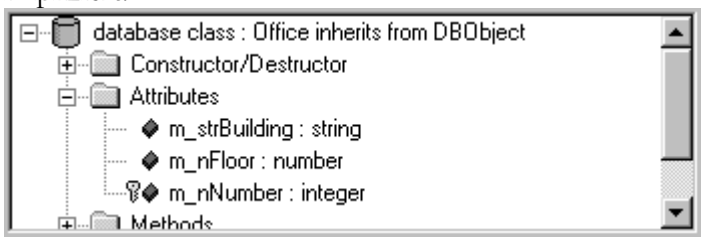


Abbildung 5-16: Darstellung der benutzerdefinierten Attribute

5.3.2.6 Methoden

Im Methodenblock sind die vom Entwerfer definierten Methoden sowie die von OSWOOD implizit generierten virtuellen Methoden für die Instanzenvalidierung dargestellt. Bei den Benutzererfassten wird anhand der Definitionszeile zwischen lesenden und schreibenden Exemplaren unterschieden.

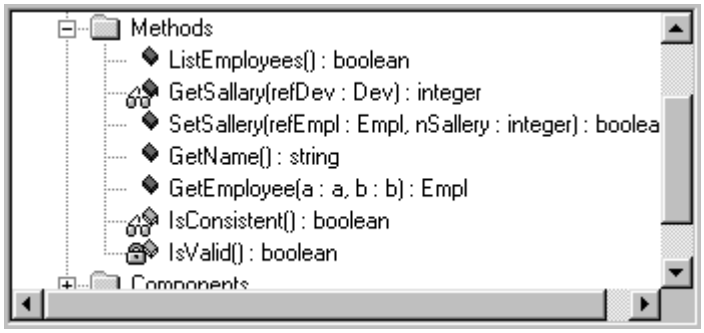


Abbildung 5-17: Darstellung der Methoden

5.3.2.7 Komponentenbeziehungen

In der Komponentensektion werden sämtliche einer bestimmten Klasse zugehörigen Komponenten aufgelistet. Je Beziehung lässt sich dabei die Kardinalitätseinschränkung, die Schlüsselkriterien, sowie die implizit generierten Attribute und Methoden herauslesen.

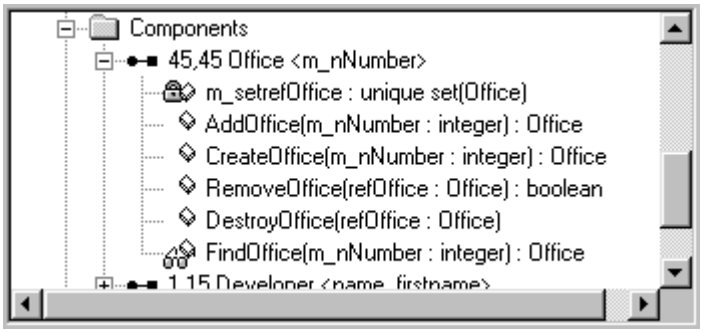


Abbildung 5-18: Darstellung der Komponentenbeziehungen

5.3.2.8 Inverse Beziehungen

Inverse Beziehungen werden gleichsam in einer eigenen Sektion zusammengefasst. Analog der Komponentenbeziehungen sind die impliziten Methoden und Attribute enthalten.

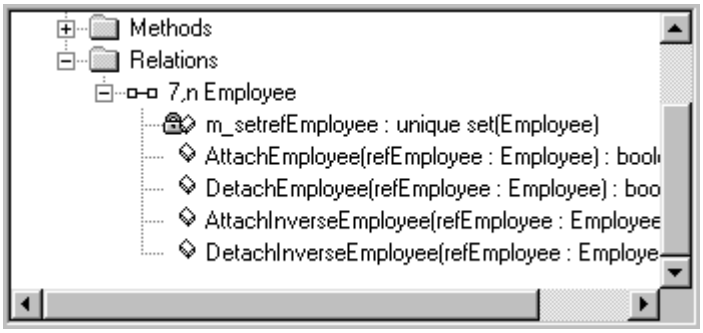


Abbildung 5-19: Darstellung der inversen Beziehungen

5.3.2.9 Evolutionsbeziehungen

Die implizit generierten Methoden für die Umsetzung einer Instanz im Rahmen der Objektevolution sind im Evolutionsteil der Klassendarstellung zu finden.

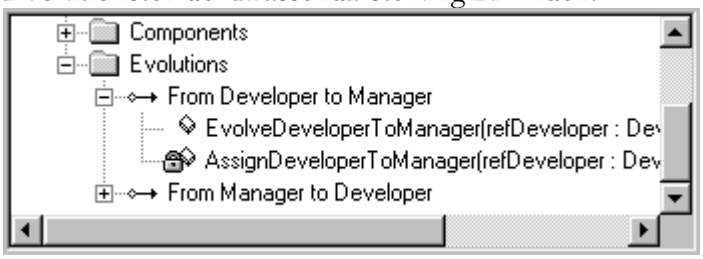


Abbildung 5-20: Darstellung der Objektevolutionen

5.3.2.10 Beobachtungsbeziehungen

Bei Hilfsklassen fehlen die Abschnitte für die inversen und die Evolutionsbeziehungen, dafür enthalten sie einen Unterbaum, der ihre Beobachtungsbeziehungen analog beschreibt.

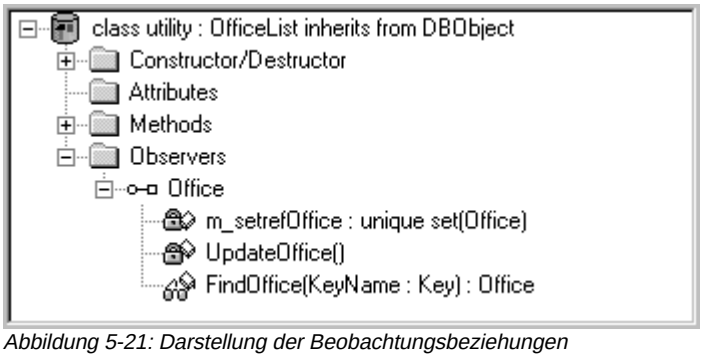


Abbildung 5-21: Darstellung der Beobachtungsbeziehungen

5.3.2.11 Die ODL-Ansicht

In der ODL-Ansicht kann der für ein bestimmtes sich im Fokus befindendes Element der Klassendarstellung generierte Code in der Schemadefinitionssprache O_2C eingesehen werden. Ist im aktuellen Fall eine Klasse selektiert, wird die Definition dieser Klasse angezeigt.

```
class Office inherit DBObject public
  type tuple (
    public m_strBuilding : string,
    public m_nFloor : number,
    public m_nNumber : integer,
    private m_setrefEmployee : unique set(Emplo
  method
    public Create() : boolean,
    public Destroy() : boolean,
    public Init() : boolean,
    public IsConsistent() : boolean,
    private IsValid() : boolean,
    public AttachEmployee(refEmployee : Employe
    public DetachEmployee(refEmployee : Employe
    public AttachInverseEmployee(refEmployee :
    public DetachInverseEmployee(refEmployee :
end;
```

Abbildung 5-22: Darstellung einer Klassendefinition in der ODL-Ansicht

Ist im Gegensatz dazu eine Methode Gegenstand des Interesses, wird entsprechend deren Implementation im Fenster ausgegeben.

```
method body GetSallary in class Company {
  o2 integer : nReturn /* return value for meth

  /* preconditions */
  if (!refDev.IsConsistent()) return (0);

  /* implementation */
  -- TODO: Add your implementation code here

  /* postconditions */
  if (!IsConsistent()) return (0);

  return (nReturn);
};/* end GetSallary */
```

Abbildung 5-23: Darstellung einer Methodenimplementation in der ODL-Ansicht

5.3.3 Anzeigen eines Transaktionsdiagramms

Durch einen Doppelklick auf ein Transaktionsdiagramm in der Navigationsansicht oder durch die **File | Open** Aktion wird die gewählte Diagrammdatei untersucht und interpretiert.
Analog zu der Verarbeitung der Schemadiagrammdateien werden dem Benutzer fatale Fehler sofort angezeigt und Warnungen in die Ausgabezeile geschrieben.
Konnte die Datei vollständig und fehlerfrei gelesen und die entsprechenden Datenstrukturen angelegt werden, erscheint die Transaktionsansicht.

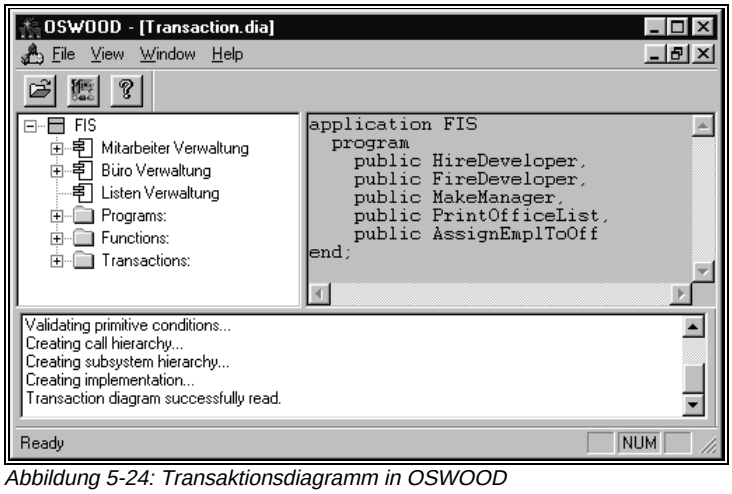


Abbildung 5-24: Transaktionsdiagramm in OSWOOD

Das Transaktionsfenster ist in zwei Bereiche - Aufrufhierarchie- und ODL-Ansicht - unterteilt.

5.3.3.1 Aufrufhierarchieansicht

In der hierarchischen Ansicht bildet die Applikation (Hauptprogramm) die Wurzel, welcher in der zweiten Ebene die Subsysteme untergeordnet sind. Zudem sind der Übersichtlichkeit halber sämtliche Programme, Funktionen und Transaktionen in entsprechende Sammlungen aufgenommen.

Die Aufrufbeziehungen zwischen den Elementen werden in den Aufrufbaum übertragen, indem ein Element einem anderen genau dann untergeordnet wird, wenn ein Aufruf vom ersten zum zweiten definiert worden ist.

Abbildung 5-25: Darstellung der Aufrufhierarchien in OSWOOD

Enthalten die Programme, Funktionen oder Transaktionen pseudocodeartige Methodenaufrufe, werden diese gleichsam in die hierarchische Ansicht integriert.

Für die Darstellung der einzelnen Elemente werden die folgenden Symbole verwendet:

	Applikation/Hauptprogramm
	Subsystem
	Programm
	Funktion
	Transaktion
	Methode

Tabelle 5-11: Symbole in der hierarchischen Ansicht eines Transaktionsdiagramms

5.3.3.2 ODL-Ansicht

In der ODL-Ansicht lässt sich für jedes in der hierarchischen Ansicht enthaltene Element der von OSWOOD generierte Code betrachten. Für Applikationen werden lediglich die Programmdeklarationen gezeigt, während bei Programmen und Transaktionen nur deren Implementation kreiert wird. Bei Funktionen wird gar beides ins ODL-Fenster geschrieben. Subsysteme sind nur Strukturierungshilfen und werden deshalb für die Codegenerierung nicht mitberücksichtigt.

5.3.4 Schemaevolution

Die Visualisierungen der Diagramme in OSWOOD erlauben dem Entwerfer eine Voransicht der späteren Umsetzung in Schemadefinitionscode und führen häufig dazu, dass Entwurfsentscheide in bezug auf Klasseneigenschaften oder Beziehungsmodelle überprüft und/oder modifiziert werden müssen. Es gilt dabei zu beachten, dass Änderungen bei einer Rückkehr zum Entwurfswerkzeug Hardy und dortiger Anpassung der Diagramme erst in OSWOOD erkannt werden können, wenn die neu gespeicherten Dateien erneut geöffnet werden. Es ist also insbesondere nicht möglich, Manipulationen an den Ausgangsdateien in OSWOOD online festzustellen, dafür ist es aber auch nicht notwendig, das Programm zu beenden und neu zu starten.
Ist der Entwerfer mit den erzielten Resultaten zufrieden, lässt er sich den Schemadefinitionscode erzeugen und verwendet diesen anschliessend, um gegenüber der Datenbankumgebung ein Schema zu kreieren (vgl. die folgende Abschnitte). OSWOOD kann aber nicht mehr eingesetzt werden um Anpassungen, welche aufgrund von neuen oder erweiterten Anforderungen notwendig werden, an einem bereits bestehenden Schema vorzunehmen.
Vielmehr soll der Entwurf von Schemaänderungen und die entsprechende Erzeugung von Schemamanipulationscode Gegenstand von weiteren Forschungen sein. Dazu bedarf es aber auch neuen Erkenntnissen in Bezug auf die Standardisierung derartiger Modifikationen seitens der Anbieter kommerzieller objektorientierter Datenbanksysteme.

5.3.5 Umsetzen in ODL

Bei den beiden verfügbaren Diagrammansichten handelt es sich lediglich um Visualisierungen, in welchen keine Manipulationen vorgenommen werden können. Natürlich wäre eine entsprechende Erweiterung des Werkzeuges - inklusive Rückschreibung in Hardydateien - denkbar, aber nicht implementiert.
Somit steht dem Benutzer in einem Diagrammfenster als einzige Aktion die Umsetzung in ODL zur Verfügung. Durch die Wahl des Menüpunktes **File | Save as ODL** werden die bereits erzeugten und in den ODL-Ansichten gezeigten Codefragmente zusammengezogen und in eine vom Benutzer gewünschte Datei - üblicherweise mit der Extension „*.o2c“ - geschrieben.
Die Codespeicherung führt keine weiteren Validierungen mehr durch, weshalb diese Verarbeitung, sofern genügend Speicherplatz auf dem Massenspeichermedium vorhanden ist, nicht fehlschlagen kann.

Sofort nach der Speicherung wird die geschriebene Datei in OSWOOD eröffnet, wo sie den erweiterten Bedürfnissen angepasst und mit den getätigten Manipulationen gespeichert werden kann.

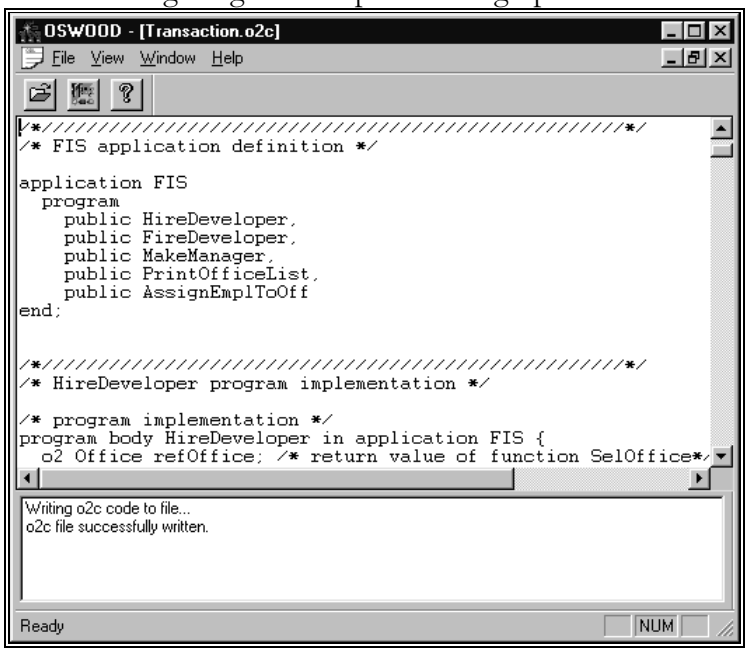


Abbildung 5-26: Darstellung einer O_2C-Datei in OSWOOD

5.3.6 Importieren der generierten O_2C-Dateien in O_2

Als Resultat der vorigen Umsetzung der Diagrammdateien in O_2C erhält man üblicherweise je Schemadiagramm und je Transaktionsdiagramm eine entsprechende o2c-Datei. Diese kann in die Datenbankumgebung O_2 eingelesen werden.

5.3.6.1 Import mittels O_2-Shell

Die O_2-Shell ist eine interaktive, rein textbasierte Datendefinitions- und manipulationsschnittstelle für Administratoren [O_2Admin 95]. In ihr kann eine Vielzahl aus dem Satz der unter O_2C verfügbaren Anweisungen direkt gegenüber der Datenbasis abgesetzt werden. Darunter befinden sich Befehle für die Schemata- oder Basenverwaltung, für ad hoc Anfragen, für Transaktionen aber auch Programm- oder Applikationsaufrufen. Insbesondere ist im Syntax der O_2-Shell die Anweisung

`#DateiName`

verfügbar, welche die in der Datei `DateiName` enthaltenen O_2C-Anweisungen der Reihe nach abarbeitet. Mit Hilfe dieser Anweisung lassen sich also die von OSWOOD erzeugten Schemadefinitionsdateien in das Datenbankschema übertragen.

5.3.6.2 Import mittels O_2-Tools

O_2-Tools ist eine grafische Umgebung für das interaktive Verwalten von Schemata, Basen und Applikationen [O_2Tools 95]. Das Werkzeug stellt Datenbankadministratoren verschiedene Browser, Editoren und sonstige Hilfsmittel zur Verfügung. Eine kurze Einführungsbeschreibung findet man in [Geppert 96, pp. 41-47].

Der Import der von OSWOOD generierten Dateien kann auch über die Navigation durch verschiedene Menüpunkte und die Auswahl der Quelldateien geschehen. Die Beschreibung des genauen Vorgehens soll [O_2Tools 95] entnommen werden.

5.4 Bewertung der Arbeit mit OSWOOD

Das Werkzeug OSWOOD begleitet den Entwerfer während des ganzen Entwurfsprozesses und stellt ihm die unterschiedlichsten Ansichten zur Verfügung. Auf der einen Seite sieht er die von ihm gezeichneten Diagramme, die daraus interpretierten Klassen- und Applikationsstrukturen, auf der anderen Seite aber auch den daraus resultierenden O_2C-Code.
Kennt sich der Entwerfer mit den Sprachelementen der Schemadefinitionssprache aus, was nicht immer gegeben sein muss und gemäss der Forderung nach der Implementationsunabhängigkeit des Entwurfes nicht gegeben sein soll, können Entwurfsfehler frühzeitig erkannt werden. Die Fehler können auch nur dann festgestellt werden, wenn bereits in einer frühen Phase die Zielumgebung festgelegt worden ist, was häufig nicht der Fall ist. Üblicherweise soll beim Entwurf die Implementation nicht vorweggenommen werden.
Natürlich lässt sich das von OSWOOD generierte Resultat auch als Zwischenergebnis („Metaschema"), interpretieren, welches dann nicht wie vorgeschlagen in O_2 eingelesen, sondern auf eine beliebige andere Datenbankumgebung abgebildet wird. Darüber hinaus liesse sich OSWOOD gegebenenfalls auch für die Generierung anderer Schemadefinitionssprachen wie C++-ODL ausbauen.
[Booch 94, p. 176] schreibt über die Rolle von Werkzeugen im Entwurfsprozess, dass sich diese ausgezeichnet eignen, um die Konsistenz, die Sicherheit und die Vollständigkeit der Diagramme zu prüfen und deren Leistungsfähigkeit zu analysieren. Der Einsatz von Werkzeugen verhindert also im Gegensatz zum papierbasierten Ansatz die Fehlinterpretation von Konstrukten oder den Ausbau der Notationen nach dem Gutdünken des Entwerfers. Werkzeuge können aber nicht oder noch nicht Wissen über die Problemdomäne ausnutzen, um Fehlentscheidungen im Entwurf zu erkennen.
Letztlich kommen gute Entwürfe von guten Entwerfern und nicht von guten Werkzeugen [Booch 94, p. 176].

6 Umsetzung

6.1 Einleitung

Im vorigen Kapitel wurde der werkzeugunterstützte Entwurfsprozess ausgeführt. In diesem Kapitel soll nun genauer beschreiben werden, wie einzelnen Konstrukte der verschiedenen Diagrammtypen von DEIMOS in Elemente der Schemadefinitionssprache O_2C umgesetzt werden.
Ein Hardydiagramm besteht aus Knoten und Kanten. Die Typen der Knoten- und Kantenkonstrukte sind dabei für einen bestimmten Diagrammtyp eingeschränkt, d.h. es können nicht beliebige Elemente in einem Diagramm enthalten sein, sondern nur diejenigen, welche in der entsprechenden Diagrammtypendefinition vorgesehen sind. Somit entfällt bereits die Behandlung vieler Ausnahmesituationen.
Das Zeichenwerkzeug lässt auch die Definition von Einschränkungen für die Verwendung der Diagrammelemente zu. Damit sind wiederum eine Reihe von gefährlichen Situationen verhindert.
Mit einer Kante werden immer genau zwei Knoten verbunden. Wie diese Verbindungen nun von OSWOOD in Sachverhalte des konzeptuellen Entwurfes umgesetzt werden, bildet der Gegenstand dieses Kapitels. Die Bewertung der Umsetzung schliesslich wird am Ende des Kapitels diskutiert.

6.2 Umsetzung eines Schemadiagramms

In einem Schemadiagramm können die folgenden Knoten und Kanten

- Einstiegspunkte,
- Datenbankklassen,
- abstrakte Klassen,
- Hilfsklassen und
- Kommentare

als Knoten und

- Vererbungsbeziehungen,
- Komponentenbeziehungen,
- inverse Beziehungen,
- Instantiierungsbeziehungen,
- Evolutionsbeziehungen,
- Beobachtungsbeziehungen und
- Notizbezüge

als Kanten enthalten sein. Bei der Umsetzung soll grundsätzlich jedes Knotenelement einem Objekt zugeordnet werden und anschliessend jedes Kantenelement in eine Beziehung zwischen zwei Objekten umgesetzt werden. (Der Begriff „Objekt“ bezieht sich auf die Implementierung in einem objektorientierten Umfeld und soll nicht mit dem Entwurfskonstrukt „Objekt“ verwechselt werden.)

6.2.1 Umsetzung der Knoten

Grundsätzlich wird jedem Knoten eine Instanz einer entsprechenden Klasse zugeordnet. Im Falle eines Klassenknotens ist für das Objekt eine Definition und eine Implementation in der Schemadefinitionssprache O_2C zu generieren, während für die Einstiegspunkte lediglich die Deklaration zu erzeugen ist. Die Notizenknoten werden zwar für Reproduktionszwecke in speziellen

Objekten abgelegt, aber für das Schreiben der O_2C-Dateien nicht weiter verwendet.

6.2.1.1 Einstiegspunkte

Einstiegspunkte sind Instanzen von Klassen, welche durch eine explizite Anweisung in der Schemadefinition persistent gemacht werden.

In der Hardydatei wird ein Einstiegspunkt durch seinen Typ identifiziert.

```
type = "entry point"
```

Der persistente Name kann vom Entwerfer als Eigenschaft des Knotens eingegeben werden.

```
'persistent name' = persistenterName,
```

In O_2C wird ein Einstiegspunkt mit der Anweisung

```
name persistenterName : typedef;
```

definiert. Dabei wird der persistente Name verwendet, um durch die Datenbasis navigieren zu können. Als Typ kann jede im Schema definierte Datenbankklasse gewählt werden. Um diese Zuordnung aber vornehmen zu können, muss mit Hilfe des Instantiierungspfeil diejenige Klasse gefunden werden, mit welcher der Einstiegspunkt verbunden ist (vlg. Umsetzung der Instantiierung, Seite 172).

Im konkreten Fall (Beispiel FIS) kann ein Einstiegspunkt in der Hardydatei folgendermassen enthalten sein:

```
node('persistant name' = "Macrohard",
  type = "entry point",
  id = 86660,
  card = 87936,
  images = [86663],
  arcs = [86664, 86668]).
```

Listing 6-1: Beispiel eines Einstiegspunktes

Bei der Umsetzung in die Definitionsanweisung unter Zuhilfenahme der Instantiierungspfeiles resultiert etwa die folgende Zeile:

```
name Macrohard : Company;
```

Listing 6-2: Beispiel einer Definition eines Einstiegspunktes

6.2.1.2 Definition der Datenbankklassen, abstrakten Klassen und Hilfsklassen

Die Datenbankklassen, abstrakten Klassen und Hilfsklassen werden in der Hardydatei im wesentlichen identisch behandelt, weshalb im Folgenden stellvertretend für diese Konstrukte die Behandlung der Datenbankklasse als Klasse im Allgemeinen beschrieben sein soll.

Eine Datenbankklasse kann durch die Typenangabe identifiziert werden.

```
type = "database class"
```

Gemäss der Diagrammdefinition ist der Entwerfer angehalten, die beiden Eigenschaften des Klassenknotens

```
'class name' = ClassName
properties = Properties
```

zu erfassen. Der Name der Klasse muss dabei - um Konflikten vorzubeugen - für den gesamten Entwurf eindeutig vergeben werden. Im Attribut 'Eigenschaften' lassen sich die Eigenheiten der Klasse beschreiben. Dies sind insbesondere deren Methoden, deren Attirubte aber auch die

Schlüssel und die Klasseninvarianten (vlg Definition der Eigenschaften, Seite 74).
In der Schemadefinitionssprache O_2C wird eine Klasse nach dem folgenden Syntax definiert (vlg. [Geppert 96, pp. 32-35]):

```
klassendef ::= class classname
           [inherit cl_name (,cl_name)*]
           [public] type typedef
           [method methoden]
end;
methoden   ::= methode (,methode)*
methode    ::= [public|private]
methoddef
methoddef  ::= name [parameter]
[:typedef]
parameter  ::= ( name : typedef (,name :
typedef)*)
ex_typedef ::= type typname : typedef ;
typedef    ::= atomar | complex
atomar     ::= primitiv | klassenname |
typname
primitiv   ::= integer | real |
character |
           boolean | string
koplex     ::= tupel | menge | liste
tupel      ::= tuple ( att_spez
(,att_spez)* )
att_spez   ::= att_name : typedef
menge      ::= [unique] set ( typename |
classname)
liste      ::= list ( typename |
classname )
```

Syntaxdiagramm 6-1: Klassendefinition in O_2C

Eine Klassendefinition besteht also im wesentlichen aus der Angabe des Namens und der Vererbungsbeziehungen, der Beschreibung der Datenstruktur und der Definition der Klassensignatur unter Angabe des Zugriffsschutzes.
Im konkreten Fall (Beispiel FIS) könnte eine Datenbankklasse `Company` wie folgt in der Hardydatei enthalten sein:

```
node('class name' = "Company",
  properties = "ListEmployees()
R GetSallary(refDev:Dev):integer
W SetSallery(refEmpl:Empl,nSallery:integer):boolean
r GetName(): string
w GetEmployee(a:a,b:b):Empl
Office<m_nNumber>
Developer<name,firstname>",
  type = "database class",
  id = 86207,
  card = 87936,
  images = [86211],
  arcs = [86292, 86664, 87321]).
```

Listing 6-3: Datenbankklasse in Hardydatei

In der Schemadefinitionsdatei müsste die entsprechenden Klasse etwa folgendermassen repräsentiert werden:

```
class Company inherit DBObject public
class Company inherit DBObject
  type tuple (
```

```
    public Name : string,
    public Address : address,
    public NrOfEmpl : number,
    private m_setrefOffice : unique set(Office),
    private m_refDeveloper : Developer,
    private m_refManager : Manager)
  method
    public Create(setrefOffice : unique set (Office),
refDeveloper : Developer) : boolean,
    public Destroy() : boolean,
    public Init(Name : string, Address : address) : Company,
    public IsConsistent() : boolean,
    private IsValid() : boolean,
    public ListEmployees() : boolean,
    public GetSallary(refDev : Dev) : integer,
    public SetSallery(refEmpl : Empl, nSallery : integer) :
boolean,
    public GetName() : string,
    public GetEmployee(a : a, b : b) : Empl,
    public AddOffice(m_nNumber : integer) : Office,
    public AddRefOffice(refOffice : Office) : boolean,
    public CreateOffice(m_nNumber : integer) : Office,
    public RemoveOffice(refOffice : Office) : boolean,
    public DestroyOffice(refOffice : Office),
    public FindOffice(m_nNumber : integer) : Office,
    public AddDeveloper(Name : string, Knowledge : string,
Firstname : string) : Developer,
    public AddRefDeveloper(refDeveloper : Developer) :
boolean,
    public CreateDeveloper(Name : string, Knowledge :
string, Firstname : string) : Developer,
    public RemoveDeveloper(refDeveloper : Developer) :
boolean,
    public DestroyDeveloper(refDeveloper : Developer),
    public FindDeveloper(Name : string, Firstname : string)
: Developer,
    public AddManager(MercedesType : string) : Manager,
    public AddRefManager(refManager : Manager) : boolean,
    public CreateManager(MercedesType : string) : Manager,
    public RemoveManager(refManager : Manager) : boolean,
    public DestroyManager(refManager : Manager),
    public FindManager() : Manager,
    public EvolveDeveloperToManager(refDeveloper :
Developer, MercedesType : string) : Manager,
    private AssignDeveloperToManager(refDeveloper :
Developer, refManager : Manager),
    public EvolveManagerToDeveloper(refManager : Manager,
Name : string, Knowledge : string, Firstname : string) :
Developer,
    private AssignManagerToDeveloper(refManager : Manager,
refDeveloper : Developer)
end;
```

Listing 6-4: Definition einer Datenbankklasse in O_2C

6.2.1.3 Implementation der Datenbankklassen, abstrakten Klassen und Hilfsklassen

Bei der Umsetzung werden einer Klasse einige implizite Eigenschaften hinzugefügt. Zum einen werden sämtliche Klassen von der vordefinierten Klasse DBObject abgeleitet und zum anderen werden in den Klassen gewisse zusätzliche virtuelle Methoden implementiert. Diese Methoden werden sowohl für die einheitliche Behandlung der Objekte als auch für die Implementation der impliziten Methoden in Beziehungsfunktionen verwendet und sollen die Sicherheit des Datenschemas durch die implizite Prüfung der Konsistenz erhöhen (vgl. “Fortpflanzung und Konsistenzsicherung”, Seite 112).

Der Klasse werden standardmässig folgende virtuellen Funktionen hinzu generiert:

`IsValid`

überprüft die vom Benutzer im Entwurf für eine Klasse bestimmten invarianten Konsistenzbeziehungen. Diese

Bedingungen beziehen sich immer auf die aktuelle Instanz, ohne Beziehungen zu anderen Objekten zu berücksichtigen, weshalb diese Methode immer am Ende einer öffentlichen Methode aufzurufen ist.

`IsConsistent`

Die Konsistenzprüfungsmethode stellt zu Beginn durch den Aufruf der Methode `IsValid` sicher, dass die Invarianten nicht verletzt sind. Anschliessend prüft sie im impliziten Teil die Kardinalitätsbedingungen, welche für die Beziehungen, an denen die aktuelle Instanz teilnimmt, vom Entwerfer definiert worden sind.
Im expliziten Teil der Implementierung lassen sich vom Entwickler weitere Konsistenzbeziehungen testen.
Diese Methode eignet sich daher ideal, um am Ende einer Transaktion, durch deren Aufruf für jedes der modifizierten Objekte, zu entscheiden, ob die Transaktion mit einem `commit` erfolgreich abgeschlossen werden kann oder durch einen `abort` rückgängig gemacht werden muss.

`Init`

Die `Init`-Methode wird vom Objektkonstruktor implizit aufgerufen. Deshalb darf sie als am ehesten als benutzerdefinierbarer Konstruktor angesehen werden. Für jedes Attribut, welches zur Erzeugungszeit initialisiert werden soll, muss ihr ein entsprechender Wert als Parameter übergeben werden. Derartige Attribute lassen sich als Klasseneigenschaften der Form `Name : Typ =` definieren. OSWOOD sorgt bei der Umsetzung dafür, dass die Initialisierungsmethode eine diesbezüglich korrekte Signatur erhält.
In der Implementation werden neben diesen Initialisierungen auch die Standardwerte zugewiesen, die vom Entwerfer für die expliziten Attribute angegeben worden sind.
Anschliessend wird durch den Aufruf der Methode `IsValid` die Initialkonsistenz des Objektes geprüft.

`Create`

Diese Methode sorgt für eine korrekte Erzeugung einer Instanz einer bestimmten Klasse. Insbesondere werden in dessen Implementation die zur Erreichung der Konsistenz zwingenden Beziehungen zu anderen Objekte aufgebaut.
Bei Komponentenbeziehungen, die eine Minimalkardinalität von grösser 0 verlangen, werden die entsprechenden Instanzen erzeugt (`new`), initialisiert (implizit) und konsistent gemacht (`Create`).
Diese implizite Erzeugung ist nur dann möglich, wenn die impliziten Initialisierungsmethoden der betroffenen Objekte keine Parameter verlangen (vlg. `Init`-Methode). Kann dies nicht garantiert werden, müssen die Instanzen ausserhalb dieser Methode kreiert werden und entsprechende Referenzen übergeben werden. Die jeweils sichere Signatur für die `Create`-Methode wird von OSWOOD automatisch erstellt.
Am Ende der Methodenausführung wird durch den Aufruf

der Methode `IsConsistent` sichergestellt, dass die Konsistenzbedingungen nun allesamt für die Instanz erfüllt sind.

`Destroy`

Der Aufruf dieser Methode kümmert sich um die Löschung einer Instanz. Sie wird also nicht zerstört - im Sprachumfang von O_2C ist kein `delete` Konstrukt verfügbar -, es wird ihr vielmehr die Voraussetzung für die Persistenz entzogen, indem alle Referenzen zu ihr abgebaut werden.
Insbesondere werden dabei sämtliche inversen Beziehungen, welche die Instanz zur Zeit noch unterhält, abgebaut und sämtliche Komponenten, welche durch entsprechende Beziehungen mit dem aktuellen Objekt verbunden sind, durch den Aufruf derselben Methode zerstört und aus den Referenzmengen entfernt.
Die Konsistenz der aktuellen Instanz kann am Schluss dieses Anweisungsblocks nicht mehr gegeben sein, weshalb die entsprechenden Methoden nicht aufgerufen werden.
Alle Methoden, mit Ausnahme der `Init`- und in manchen Fällen der `Create`-Methode, sind parameterlos (d.h. sie beziehen sich nur auf die Definitionen aus dem Entwurf und garantieren deshalb eine Kontextfreiheit) und retournieren einen bool'schen Wert, über welchen der Erfolg des Aufrufes geprüft werden kann. Die Initialisierungsmethode hingegen besitzt für alle zu initialisierenden Attribute einen Parameter und retourniert gemäss der Forderung des Datenbanksystems den Objektidentifikator des eben erzeugten Objektes.
Neben den impliziten, virtuellen Methoden, die für eine Klasse generiert werden, müssen Implementationsgerüste für die vom Entwerfer definierten expliziten Methoden der Klasse erzeugt werden. Ein derartiges Gerüst besteht aus den folgenden Teilen:

Rückgabevariable

Falls für die Methode ein Rückgabetyp definiert wurde, wird eine entsprechende lokale Variable dieses Typs angelegt.

Vorbedingungen

Für sämtliche übergebenen Parameter wird in diesem Abschnitt dessen Gültigkeit geprüft. Handelt es sich bei dem Parameter um eine Klasse, so wird dessen Validität mit Hilfe der Methode IsConsistent getestet. Im Falle einer Übergabevariablen eines atomaren Datentyps (vgl. [Geppert 96, p. 50), wird lediglich in einer Kommentarzeile darauf hingewiesen, dass dessen Gültigkeit zu validieren ist.

Implementation

Der Entwickler ist durch eine Kommentarzeile der Form „TODO: ..." aufgerufen, in diesem Teil seine Methodenimplementierung einzufügen.

Nachbedingungen

Handelt es sich bei der Methode um eine auf die Datenbasis schreibende, wird durch den Aufruf der Methode `IsConsistent`, welche implizit die Invarianz des Objektes prüft, sichergestellt, dass die Instanz nach dem Verlassen der

Methode konsistent ist. Bei nur lesenden Methoden entfällt diese Prüfung.

Rückgabe

Im Erfolgsfall wird die lokale Variable für den Rückgabewert zurückgegeben, während bei einer Fehlerbedingung ein vordefinierter Wert zurückgeliefert wird. Dieser Wert ist im Falle einer geforderten Objektreferenz die Nullreferenz und für atomare Rückgabewerte ein Leerdatum (Leerstring, 0, etc.).

In der Schemadefinitionssprache O_2C ist der folgende Syntax für die Implementation einer Klassenmethode vorgesehen:

```
methodimpl ::= method body methodname
          in class classname block
block      ::= { variables, statement };
variables  ::= (vardef)*
vardef     ::= o2 (typename|classname)
varname ;
```

Syntaxdiagramm 6-2: Methodendefinition in O_2C

In einem konkreten Fall (Beispiel FIS) könnten die Implementationen der virtuellen Methoden wie folgt aussehen:

```
method body IsValid in class Company {
  /* check invariant constraints */
  if (!(self->m_strName != "Microsoft")) return (false);

  return (true);
};/* end IsValid */
```

Listing 6-5: Virtuelle Methode IsValid() für die Klasse Company

```
method body IsConsistent in class Company {
  /* check object's validity */
  if (!self->IsValid()) return (false);

  /* check cardinality constraints */
  if ((count(self->m_setrefOffice<45))||
(count(self->m_setrefOffice>45))) return (false);
  if ((count(self->m_setrefDeveloper<1))||
(self->count(m_setrefDeveloper>15))) return (false);
  if ((count(self->m_setrefManager<1))||
(count(self->m_setrefManager>15))) return (false);

  return (true);
};/* end IsConsistent */
```

Listing 6-6: Virtuelle Methode IsConsistent() für die Klasse Company

```
method body Init in class Company {
  /* initialize the explicit attributes */
  self->NrOfEmpl = 1;
  self->Name = Name;
  self->Address = Address;

  /* initialize the implicit attributes (components) */
  self->m_setrefOffice = unique set();
  self->m_refDeveloper = (o2 Developer)0;
  self->m_refManager = (o2 Manager)0;

  /* check object's validity */
  if (!self->IsValid()) return ((o2 Company)0);

  return (self);
};/* end Init */
```

Listing 6-7: Virtuelle Methode Init() für die Klasse Company

```
method body Create in class Company {
  o2 integer i; /* counter used in for statements */
  o2 Office refOffice; /* local variable for component
creating */

  /* check preconditions */
  for (i=0;i<45;i++) {
    if (setrefOffice[i] == (o2 Office)0) return (false);
  } /* endfor */
```

```
  if (refDeveloper == (o2 Developer)0) return (false);

  /* create components */
  for (refOffice in setrefOffice) {
    if (self->FindOffice(refOffice->m_nNumber) != (o2
Office)0) return (false);
    self->m_setrefOffice += unique set (refOffice);
  } /* endfor */
  self->m_refDeveloper = refDeveloper;

  /* check object's consistency */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end Create */
```

Listing 6-8: Virtuelle Methode Create() für die Klasse Company

```
method body Destroy in class Company {
  o2 integer i; /* counter used in for statements */
  o2 Office refOffice; /* local variable for comp destroying
*/
  o2 Developer refDeveloper; /* local variable for comp
destroy */
  o2 Manager refManager; /* local variable for comp
destroying */

  /* destroy components */
  for (refOffice in self->m_setrefOffice) {
    refOffice->Destroy();
  }/* endfor */
  m_setrefOffice = unique set();
  if (self->m_refDeveloper != (o2 Developer)0) {
    self->m_refDeveloper->Destroy();
    self->m_refDeveloper = (o2 Developer)0;
  }/* endif */
  if (self->m_refManager != (o2 Manager)0) {
    self->m_refManager->Destroy();
    self->m_refManager = (o2 Manager)0;
  }/* endif */

  return (true);
};/* end Destroy */
```

Listing 6-9: Virtuelle Methode Destroy() für die Klasse Company

Eine benutzerdefinierte explizite Methode könnte wie folgt implementiert werden:

```
method body SetSallery in class Company {
  o2 boolean : bReturn /* return value for method */

  /* preconditions */
  if (!refEmpl.IsConsistent()) return (false);
  -- ASSERT: nSallery is valid;

  /* implementation */
  -- TODO: Add your implementation code here

  /* postconditions */
  if (!self->IsConsistent()) return (false);

  return (bReturn);
};/* end SetSallery */
```

Listing 6-10: Explizite Methode SetSallery() in der Klasse Company

6.2.2 Umsetzung der Kanten

Grundsätzlich wird jeder Kante eine Instanz einer entsprechenden Kantenklasse zugeordnet. Kanten verbinden immer zwei Knoten miteinander und affektieren demnach entweder denjenigen Knoten, von dem sie herstammen, oder denjenigen, auf den sie zulaufen, oder beide gleichzeitig.

Wenn Kanten im Sinne von DEIMOS Beziehungen darstellen (Komponentenbeziehungen, inverse Beziehungen), werden beide Knoten beeinflusst. Repräsentieren die Kanten Bezüge (Vererbungsbeziehungen, Instantiierung), sind nur die Zielknoten betroffen. Falls die Kanten beobachtender Natur sind

(Beobachtungsbeziehungen), werden nur die Ausgangsknoten tangiert.
Kanten können aber auch eine Fähigkeit eines Knotens darstellen, wie dies im Falle der Evolutionsbeziehung zutrifft, und demnach einem speziellen Umsetzungsprozess unterworfen sein. Sind Notizen über einen Notizbezug mit einem Knoten verbunden, wird diese Abhängigkeit gleichsam durch eine Kante dargestellt. Derartige Kanten werden aber für das Schreiben der O_2C-Dateien nicht weiter verwendet und dementsprechend herausgefiltert.
In jedem Fall aber stellen Kanten Eigenschaften der Knoten dar und werden somit während der Interpretation in die Datenstrukturen der Knotenklassen übertragen.
Kanten werden in der Hardydatei als eigene Elemente abgelegt. Die Variable

`type = Type`

lässt eine Identifikation des Pfeiltyps zu. Die Typenbezeichung stammt aus der Diagrammdefinition. Der Eintrag

`connections = [[NodeIdFrom, NodeIdTo]]`

gibt darüber Aufschluss, welche beiden Knoten miteinander verbunden sind. Die in den eckigen Klammern eingefassten Zahlen repräsentieren die Identifikatoren der Knoten. Die erste Zahl definiert den Ausgangsknoten, die zweite Zahl steht für den Zielknoten.
Nachfolgend sei auf die verschiedenen Pfeiltypen näher eingegangen und deren Umsetzung in die Schemadefinitionssprache erläutert.

6.2.2.1 Vererbungsbeziehungen

Vererbungsbeziehungen werden in der Hardydatei anhand ihres Typs erkannt.

`type = "inheritance"`

Eine Vererbungsbeziehung affektiert die Klassendefinition dahingehend, als dass der Name der Vaterklasse in die `inherit` Deklaration eingefügt werden muss (vgl. Syntaxdiagramm 6-1, Seite 156).

`inherit ClassName (,ClassName)*`

Bei der Interpretation der Vererbungsbeziehungen muss besonders darauf geachtet werden, dass in den Ableitungen keine Zyklen enthalten sind.
In einem konkreten Fall (Beispiel FIS) könnte folgender Pfeil in der Hardydatei enthalten sein:

```
arc(type = "inheritance",
  id = 87313,
  card = 87936,
  images = [87316],
  connections = [[86217, 87245]],
  arc_image_type = "inheritance arc").
```

Listing 6-11: Vererbungspfeil in einer Hardydatei

In der Schemadefinitionsdatei würde sich dieser etwa wie folgt äussern:

```
class Developer inherit Employee public
  type tuple (
    ...
               )
  method
    ...
end;
```

Listing 6-12: Vererbungsbeziehung in der Schemadefinitionsdatei

6.2.2.2 Komponentenbeziehung

Eine Komponentenbeziehung drückt eine existenzabhängige Aggregation zwischen zwei Klassen aus. Wird eine derartige Beziehung zwischen zwei Klassen eingerichtet, so wird der Besitzerklasse (Ausgangsklasse des Pfeils) die Kontrollfähigkeit über die Komponentenklasse (Zielklasse des Pfeils) übertragen.
Komponentenbeziehungen werden in der Hardydatei anhand ihres Typs erkannt.
`type = "component"`
Der Eintrag
`cardinality = "[1,15]"`
definiert die minimale (erste Zahl), bzw. die maximale (zweite Zahl) Anzahl der zugelassenen Komponenten. Diese Angaben werden vom Entwerfer festgelegt und fügen der Datenbasis weitere Konsistenzbedingungen hinzu.
Im konkreten Fall (Beispiel FIS) könnte in der Hardydatei der Komponentenpfeil zwischen den Knoten `Company` und `Developer` wie folgt enthalten sein:

```
arc(cardinality = "[1,15]",
  inverse = "1",
  name = "",
  type = "component",
  id = 87321,
  card = 87936,
  images = [87325],
  connections = [[86207, 87245]],
  arc_image_type = "component arc").
```

Listing 6-13: Beispiel einer Komponentenbeziehung in der Hardydatei

Die Besitzerklasse übt die Kontrolle über ihre Komponenten aus, was bedeutet, dass einzig die Instanzen der aggregierten Klasse Instanzen der Komponentenklasse erzeugen und zerstören dürfen. Zudem kann ein einzelnes Exemplar ausschliesslich mit Hilfe der Aggregationsinstanz aufgefunden werden.
Aus diesem Grund müssen die folgenden Verwaltungsmethoden und -attribute für die Besitzerklasse implizit generiert werden.
`m_refset<`*`ClassName`*`> : unique set(<`*`ClassName`*`>)`
Mit Hilfe dieses Attributes werden die Referenzen auf die Komponenten verwaltet. Zudem wird für jedes erzeugte Objekte dank der Aufnahme der Referenz in diese Menge seine Persistenz erreicht.
`Add<`*`ClassName`*`>([KeyAttr1 [,KeyAttrN]*])`
Diese Methode erzeugt eine neue Instanz der Klasse `ClassName` und fügt die Referenz der Referenzmenge `m_refset<`*`ClassName`*`>` hinzu.
Ein Hinzufügen einer neuen Komponente ist natürlich nur dann möglich, wenn die obere Limite, definiert durch die Kardinalitätsangabe, nicht bereits erreicht worden ist.
Zudem kann durch die Definition von Schüsseln die Eindeutigkeit bezüglich dieser Ausprägung verlangt werden, was gleichsam im Rahmen der Vorbedingungen geprüft werden muss. Aus diesem Grund müssen dieser Methode die Werte der Schlüsselattribute des zu erzeugenden Objektes

mitgeliefert werden.
Nachdem das gewünschte Objekt kreiert worden ist, wird dessen Konsistenz durch die Aufrufe von Init() und Create() gesichert. Anschliessend werden etwaige Schlüsselwerte gesetzt und die Konsistenz der Besitzerklasse geprüft.
Im Erfolgsfall wird die Referenz auf das eben erzeugte Objekt an den aufrufenden Programmfaden zurück geliefert, während im Fehlerfall die Nullreferenz zurückgegeben wird.

`AddRef<`*`ClassName`*`>(`*`refClassName`*`:`*`ClassName`*`)`

Kann aufgrund der Schemadefinition das Komponentenobjekt durch den Aufruf der `Add`-Methode nicht auf sichere Weise erzeugt werden (Konsistenzverletzung), kann dies auch ausserhalb der Methoden der impliziten Komponentenbeziehungen erledigt werden. Die Komponentenbeziehung kann in diesem Fall trotzdem sicher aufgebaut werden, indem das erzeugte Objekt durch den Aufruf der Funktion `AddRef` zur Komponentenmenge hinzugefügt wird.

`Remove<`*`ClassName`*`>(ref<`*`ClassName`*`>:<`*`ClassName`*`>)`

Diese Methode entfernt ein bestimmtes durch den Übergabeparameter referenziertes Objekt aus der Menge der Komponenten.
Als Vorbedingungen ist zu prüfen, ob die zu löschende Instanz auch in der Komponentensammlung enthalten ist und die Minimalkardinalität durch diese Löschung nicht unterschritten wird.
Sind die beiden Voraussetzungen gegeben, so kann das Objekt durch den Aufruf der Methode `Destroy`, welche ihrerseits eine Löschung der Subkomponenten erzwingt, für die Löschung vorbereitet werden.
Durch das Entfernen des Objektzeigers aus der Referenzmenge ist die Bedingung für die Persistenz des zu löschenden Objektes nicht mehr gegeben, weshalb das Datenbankverwaltungssystem die Instanz aus der Datenbasis löscht.
Falls die aktuelle Instanz noch konsistent ist, wird der bool'sche Erfolgswert zurückgegeben, andernfalls der Misserfolgswert.

`Find<`*`ClassName`*`>([KeyAttr1 [,KeyAttrN]*])`

Diese Methode versucht, in der Menge der Komponenten des Types *`ClassName`* dasjenige Exemplar ausfindig zu machen, dessen Schlüsselattributwerte mit den Übergabeparametern übereinstimmen. Diese Suche wird gegenüber der Datenbasis als OQL-Anfrage formuliert. Kann ein solches gefunden werden, wird dessen Objektidentifikator zurück geliefert. War die Suche hingegen erfolglos, wird die Nullreferenz retourniert.

`Create<`*`ClassName`*`>([KeyAttr1 [,KeyAttrN]*])`

Häufig lassen sich Schemata nicht derart beschreiben, dass die Integrität der Datenbasis nach jedem Einfügen gegeben ist (zwingende inverse Beziehungen oder

Konsistenzbedingungen, die unter mehreren Objekten wirken).
Aus diesem Grund lässt die `Create`-Methode, welche im wesentlichen dieselben Dienste wie die Methode `Add` erfüllt, das Erzeugen von Objekten zu, ohne bei einer etwaigen Konsistenzverletzung abzubrechen.
Beim Einsatz dieses Typs von Methoden gilt es besonders zu beachten, dass diese ausschliesslich in Transaktionen aufgerufen werden und vor dem erfolgreichen Abschluss durch ein `commit` für alle beteiligten Objekte die Methoden zur Konsistenzprüfung aufgerufen werden.
`Destroy<`*`ClassName`*`>(ref<`*`ClassName`*`>:<`*`ClassNam e`*`>)`
Dasselbe gilt für die `Destroy`-Methode, welche das „unsichere“ Analogon zu der `Remove`-Methode darstellt.
Im konkreten Fall (Beispiel FIS) könnten die generierten Methoden für die Komponentenbeziehung mit Schlüssel `Name` und `Firstname` und der Karidnalitätsbedingung (minimal 1, maximal 1) zwischen `Company` und `Developer` wie folgt implementiert sein:

```
method body AddDeveloper in class Company {
  o2 Developer refDeveloper;

  /* check preconditions */
  if (self->m_refDeveloper) != (o2 Developer)0)
    return ((o2 Developer)0);

  /* create new component */
  refDeveloper = new Developer(Name, Knowledge, Firstname);
  if (!refDeveloper->Create()) return ((o2 Developer)0);
  self->m_refDeveloper = refDeveloper;

  /* check postconditions */
  if (!self->IsConsistent()) return ((o2 Developer)0)

  return (refDeveloper);
};/* end AddDeveloper */
```

Listing 6-14: Implementation der Methode AddDevloper() in der Klasse Company

```
method body AddRefDeveloper in class Company {
  /* validate params */
  if (refDeveloper == (o2 Developer)0) return (false);

  /* check preconditions */
  if (self->m_refDeveloper) != (o2 Developer)0) return ((o2
Developer)0);

  /* add referenced component */
  self->m_refDeveloper = refDeveloper;

  /* check postconditions */
  if (!self->IsConsistent()) return (false)

  return (true);
};/* end AddRefDeveloper */
```

Listing 6-15: Implementation der Methode AddRefDevloper() in der Klasse Company

```
method body CreateDeveloper in class Company {
  o2 Developer refDeveloper;

  /* check preconditions */
  if (self->FindDeveloper(Name, Firstname) != (o2
Developer)0)
    return ((o2 Developer)0);

  /* create new component */
  refDeveloper = new Developer(Name, Knowledge, Firstname);
  self->m_refDeveloper = refDeveloper;

  /* check postconditions */
  -- NOTE: This method can cause inconsistent state.
```

```
  -- TODO: Check consistency outside this method by calling
IsConsistent()

  return (refDeveloper);
};/* end CreateDeveloper */
```
Listing 6-16: Implementation der Methode CreateDevloper() in der Klasse Company
```
method body RemoveDeveloper in class Company {
  /* check preconditions */
  if (self->m_refDeveloper != refDeveloper) return (false);

  /* destroy component */
  refDeveloper->Destroy();
  self->m_refDeveloper = (o2 Developer)0;

  /* check postconditions */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end RemoveDeveloper */
```
Listing 6-17: Implementation der Methode RemoveDevloper() in der Klasse Company
```
method body DestroyDeveloper in class Company {

  /* destroy component */
  if (self->m_refDeveloper == refDeveloper)
    self->m_refDeveloper = (o2 Developer)0;

  /* check postconditions */
  -- NOTE: This method can cause inconsistent state.
  -- TODO: Check consistency outside this method by calling
IsConsistent()

};/* end DestroyDeveloper */
```
Listing 6-18: Implementation der Methode DestroyDevloper() in der Klasse Company
```
method body FindDeveloper in class Company {
  o2 unique set(Developer) refsetDeveloper; /* result set
for OQL query */

  /* query item with given key values */
  o2query(refsetDeveloper,"\
      select e from e in $1 \
      where x->Name = $2 && \
            x->Firstname = $3",
    unique set(self->m_refDeveloper),Name,Firstname);

  if (count(refsetDeveloper)==0) return ((o2 Developer)0);
  else return (element(refsetDeveloper));
};/* end FindDeveloper */
```
Listing 6-19: Implementation der Methode FindDevloper() in der Klasse Company

Im Falle der Komponentenbeziehung zu der Klasse `Developer` konnte die Verbindung in einer einfachen Referenzvariablen abgebildet werden, da höchstens eine Assoziation zwischen den beiden Klassen vorhanden sein kann. Bei der entsprechenden Beziehung zu der Klasse `Office` mit dem Schlüssel `nNumber` hingegen ist die Kardinalität [1,45] angenommen, was sich daher in einer mengenwertigen Referenzvariablen nierderschlägt. Die entsprechenden Methoden würden etwa folgendermassen generiert:

```
method body AddOffice in class Company {
  o2 Office refOffice;

  /* check preconditions */
  if (count(self->m_setrefOffice) >= 45) return ((o2
Office)0);
  if (self->FindOffice(m_nNumber) != (o2 Office)0)
    return ((o2 Office)0);

  /* create new component */
  refOffice = new Office(m_nNumber);
  if (!refOffice->Create()) return ((o2 Office)0);
  self->m_setrefOffice += unique set(refOffice);
```

```
  /* check postconditions */
  if (!self->IsConsistent()) return ((o2 Office)0)

  return (refOffice);
};/* end AddOffice */
```
Listing 6-20: Implementation der Methode AddOffice() in der Klasse Company
```
method body AddRefOffice in class Company {
  /* validate params */
  if (refOffice == (o2 Office)0) return (false);

  /* check preconditions */
  if (count(self->m_setrefOffice) >= 45) return ((o2
Office)0);
  if (refOffice in self->m_setrefOffice) return (false);
  if (self->FindOffice(refOffice->m_nNumber) != (o2
Office)0)
    return (false);

  /* add referenced component */
  self->m_setrefOffice += unique set(refOffice);

  /* check postconditions */
  if (!self->IsConsistent()) return (false)

  return (true);
};/* end AddRefOffice */
```
Listing 6-21: Implementation der Methode AddRefOffice() in der Klasse Company
```
method body CreateOffice in class Company {
  o2 Office refOffice;

  /* check preconditions */
  if (self->FindOffice(m_nNumber) != (o2 Office)0)
    return ((o2 Office)0);

  /* create new component */
  refOffice = new Office(m_nNumber);
  self->m_setrefOffice += unique set(refOffice);

  /* check postconditions */
  -- NOTE: This method can cause inconsistent state.
  -- TODO: Check consistency outside this method by calling
IsConsistent()

  return (refOffice);
};/* end CreateOffice */
```
Listing 6-22: Implementation der Methode CreateOffice() in der Klasse Company
```
method body RemoveOffice in class Company {
  /* check preconditions */
  if (!refOffice in self->m_setrefOffice) return (false);
  if (count(self->m_setrefOffice) <= 1) return (false);

  /* destroy component */
  refOffice->Destroy();
  self->m_setrefOffice -= unique set(refOffice);

  /* check postconditions */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end RemoveOffice */
```
Listing 6-23: Implementation der Methode RemoveOffice() in der Klasse Company
```
method body DestroyOffice in class Company {

  /* destroy component */
  if (refOffice in self->m_setrefOffice)
    self->m_setrefOffice -= unique set(refOffice);

  /* check postconditions */
  -- NOTE: This method can cause inconsistent state.
  -- TODO: Check consistency outside this method by calling
IsConsistent()

};/* end DestroyOffice */
```
Listing 6-24: Implementation der Methode DestroyOffice() in der Klasse Company
```
method body FindOffice in class Company {
  o2 unique set(Office) refsetOffice; /* resultset for OQL
query */
```

```
  /* query item with given key values */
  o2query(refsetOffice,"\
      select e from e in $1 \
      where x->m_nNumber = $2",
    self->m_setrefOffice,m_nNumber);

  if (count(refsetOffice)==0) return ((o2 Office)0);
  else return (element(refsetOffice));
};/* end FindOffice */
```
Listing 6-25: Implementation der Methode FindOffice() in der Klasse Company

6.2.2.3 Inverse Beziehungen

Beliebige Assoziationen zwischen zwei Klassen werden in der Terminologie von DEIMOS inverse Beziehungen genannt. Inverse Beziehungen werden in der Hardydatei anhand ihres Typs erkannt.

`type = "inverse relation"`

Durch die Einträge

`'cardinality begin' = "1"`
`'cardinality end' = "7+"`

sind die Kardinalitätseinschränkungen an den beiden Enden der Kante definiert. Diese Angaben werden vom Entwerfer festgelegt und fügen der Datenbasis weitere Konsistenzbedingungen hinzu.

Im konkreten Fall (Beispiel FIS) könnte in der Hardydatei eine inverse Beziehung zwischen den Knoten `Office` und `Employee` wie folgt abgelegt sein:

```
arc('cardinality begin' = "1",
  'cardinality end' = "7+",
  name = "inhabitants",
  type = "inverse relation",
  id = 87326,
  card = 87936,
  images = [87330],
  connections = [[86212, 87245]],
  arc_image_type = "inverse relation arc").
```
Listing 6-26: Beispiel einer inversen Beziehung in der Hardydatei

Eine inverse Beziehung muss aus Gründen der Löschfortpflanzung von beiden beteiligten Instanzen auf-, bzw. abgebaut werden können. Somit müssen bei Erkennung einer derartigen Beziehung beide Klassen um einige Methoden und Attribute erweitert werden.

`m_refset<`*`ClassName`*`> : unique set(<`*`ClassName`*`>)`

Mit Hilfe dieses Attributes werden die Referenzen auf die Objekte verwaltet, mit denen man in Beziehung steht. Durch die Speicherung dieser Referenzen werden Objekte persistent, weshalb bei der Löschung des referenzierten Objektes (vgl. „Komponentenbeziehung", Seite 163) darauf geachtet werden muss, dass diese Referenz gleichfalls entfernt wird.

`Attach<`*`ClassName`*`>(ref<`*`ClassName`*`>:<`*`ClassName`*`>)`

Die Methode baut eine inverse Beziehung zu der Klasse *`ClassName`* auf.

Als Vorbedingungen wird zum einen sichergestellt, dass die Maximalkardinalität nicht übertroffen wird, und zum anderen, dass nicht bereits eine Beziehung zu der durch den übergebenen Parameter referenzierten Instanz exisitert.

Um eine echte inverse Beziehung einrichten zu können, muss

anschliessend die beteiligte Instanz veranlasst werden, ihrerseits die Verbindung aufzubauen. Dies geschieht durch den Aufruf der Partnermethode `AttachInverse<OwnClassName>`, welche entsprechend von der Klasse `ClassName` zur Verfügung gestellt werden muss.
Konnte die Beziehung von der beteiligten Instanz erfolgreich eingerichtet werden, kann nun die gelieferte Referenz in die Referenzmenge eingetragen werden und die Nachbedingung validiert werden.

`Detach<ClassName>(ref<ClassName>:<ClassName>)`

Diese Methode bildet das beziehungsabbauende Analogon zu der Methode `Attach`.
Nach der Prüfung der Vorbedingungen, dass die Minimalkardinalität nicht unterschritten wird und überhaupt eine Beziehung zu dem übergebenen Objekt besteht, wird analog zu der Aufbaufunktion die entsprechende Partnermethode der referenzierten Instanz aufgerufen.
Nach dem Löschen aus der Referenzliste werden im Nachbedingungsblock die Konsistenzprüfungen durchgeführt.

`DetachInverse<ClName>(ref<ClName>:<ClName>)`

Diese Methode ist die beschriebene Partnerfunktion für den Aufbau der inversen Beziehung.

`AttachInverse<ClName>(ref<ClName>:<ClName>)`

Diese Methode ist die beschriebene Partnerfunktion für den Abbau der inversen Beziehung.
Alle Methoden retournieren einen bool'schen Wert, der über den Erfolg des Beziehungsauf-, bzw. -abbaus Auskunft gibt.
Im konkreten Fall (Beispiel FIS) könnten die generierten Methoden für die inverse Beziehung (1,7+) zwischen `Office` und `Employee` wie folgt implementiert sein:
Auf der Seite der Klasse `Employee`

```
method body AttachOffice in class Employee {

  /* check preconditions */
  if (self->m_refOffice != (o2 Office)0) return (false);

  /* attach referenced object */
  if (!refOffice->AttachInverseEmployee(self)) return
(false);
  self->m_refOffice = refOffice;

  /* check postconditions */
  if (!refOffice->IsConsistent()) return (false);
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end AttachOffice */
```

Listing 6-27: Implementation der Methode AttachOffice() in der Klasse Employee

```
method body DetachOffice in class Employee {

  /* check preconditions */
  if (self->m_refOffice != refOffice) return (false);

  /* detach referenced object */
  if (!refOffice->DetachInverseEmployee(self)) return
(false);
  self->m_refOffice = (o2 Office)0;

  /* check postconditions */
```

```
  if (!refOffice->IsConsistent()) return (false);
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end DetachOffice */
```
Listing 6-28: Implementation der Methode DetachOffice() in der Klasse Employee
```
method body AttachInverseOffice in class Employee {

  /* check preconditions */
  if (self->m_refOffice != (o2 Office)0) return (false);

  /* attach referenced object */
  self->m_refOffice = refOffice;

  /* check postconditions */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end AttachInverseOffice */
```
Listing 6-29: Implementation der Methode AttachInverseOffice() in der Klasse Employee
```
method body DetachInverseOffice in class Employee {

  /* check preconditions */
  if (self->m_refOffice != refOffice) return (false);

  /* detach referenced object */
  self->m_refOffice = (o2 Office)0;

  /* check postconditions */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end DetachInverseOffice */
```
Listing 6-30: Implementation der Methode DetachInverseOffice() in der Klasse Employee

Auf der Seite der Klasse `Office`
```
method body AttachEmployee in class Office {

  /* check preconditions */
  if (refEmployee in self->m_setrefEmployee) return (false);

  /* attach referenced object */
  if (!refEmployee->AttachInverseOffice(self)) return
(false);
  self->m_setrefEmployee += unique set(refEmployee);

  /* check postconditions */
  if (!refEmployee->IsConsistent()) return (false);
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end AttachEmployee */
```
Listing 6-31: Implementation der Methode AttachEmployee() in der Klasse Office
```
method body DetachEmployee in class Office {

  /* check preconditions */
  if (count(self->m_setrefEmployee) <= 7) return (false);
  if (!refEmployee in self->m_setrefEmployee) return
(false);

  /* detach referenced object */
  if (!refEmployee->DetachInverseOffice(self)) return
(false);
  self->m_setrefEmployee -= unique set(refEmployee);

  /* check postconditions */
  if (!refEmployee->IsConsistent()) return (false);
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end DetachEmployee */
```
Listing 6-32: Implementation der Methode DetachEmployee() in der Klasse Office
```
method body AttachInverseEmployee in class Office {

  /* check preconditions */
  if (refEmployee in self->m_setrefEmployee) return (false);
```

```
  /* attach referenced object */
  self->m_setrefEmployee += unique set(refEmployee);

  /* check postconditions */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end AttachInverseEmployee */
```
Listing 6-33: Implementation der Methode AttachInverseEmployee() in der Klasse Office

```
method body DetachInverseEmployee in class Office {

  /* check preconditions */
  if (count(self->m_setrefEmployee) <= 7) return (false);
  if (!refEmployee in self->m_setrefEmployee) return
(false);

  /* detach referenced object */
  self->m_setrefEmployee -= unique set(refEmployee);

  /* check postconditions */
  if (!self->IsConsistent()) return (false);

  return (true);
};/* end DetachInverseEmployee */
```
Listing 6-34: Implementation der Methode DetachInverseEmployee() in der Klasse Office

6.2.2.4 Instantiierung

In O_2C wird ein Einstiegspunkt mit der Anweisung

name *persistenterName* : *typedef*;

definiert (vgl "Einstiegspunkte", Seite 154). Dabei muss der persistente Name dem entsprechenden Knoten des Typs Einstiegspunkt entnommen werden und der Name des Typs dem entsprechenden Klassenknoten. Diese beiden Knoten sind über den Instantiierungspfeil miteinander verbunden, weshalb diese Kante für die Generierung der Definitionsanweisung für den Einstiegspunkt herangezogen werden muss.

In der Hardydatei werden derartige Pfeile anhand ihres Typeneintrages

`type = "instantiation"`

erkannt.

Nachfolgend sei für des Beispiel FIS die Repräsentation der Instantiierung der Klasse `Company` durch den Einstiegspunkt `Macrohard` in der Hardydatei aufgeführt.

```
arc(type = "instantiation",
  id = 86664,
  card = 87936,
  images = [86667],
  connections = [[86660, 86207]],
  arc_image_type = "instantiation arc").
```
Listing 6-35: Beispiel eines Instantiierungspfeil in der Hardydatei

6.2.2.5 Evolution

Wird zwischen zwei Knoten ein Evolutionspfeil gezeichnet, so wird damit die Fähigkeit der Ausgangsklasse beschrieben, Instanzmigration zur Zielklasse unternehmen zu können.

Ein Evolutionspfeil wird in der Hardydatei anhand seines Typeneintrages der Form

`type = "evolution"`

erkannt. Durch die Beschriftung des Pfeilkonstruktes mit dem Namen der Evolutionsfunktion wird der Eintrag

`'evolution function' =` *FunctionName*

geschrieben, welcher bei der Umsetzung den standardmässigen Namen der Evolutionsfunktion überschreibt.
Eine Kantendefinition des Typs 'Evolution' könnte in einem konkreten Fall (Beispiel FIS) wie folgt gefunden werden:

```
arc('evolution function' = "Enrich()",
  type = "evolution",
  id = 86258,
  card = 87936,
  images = [86262],
  connections = [[86217, 86222]],
  arc_image_type = "evolution arc").
```

Listing 6-36: Beispiel einer Evolutionsbeziehung in der Hardydatei

Die Objektevolution wird von der Besitzerklasse der beiden beteiligten Klassen kontrolliert, weshalb dieser zusätzliche Methoden zur Verwaltung des Migrationsprozesses hinzugefügt werden müssen.

`Evolve<`*`SrcClass`*`>To<`*`DestClass`*`>(`
`ref<`*`SrcClass`*`>:<`*`ScrClass`*`>):<`*`DestClass`*`>`

Mit Hilfe dieser Methode wird die referenzierte Instanz der Ausgangsklasse `SrcClass` in eine neue Instanz der Klasse `DestClass` umgesetzt.
Da für die untersuchte Datenbasis die Objektevolution nicht unterstützt wird, muss diese durch das Erzeugen einer neuen Instanz der Zielklasse, die Zuweisung der gemeinsamen Daten und die Löschung der obsoleten Instanz simuliert werden.
Für die Zuweisung der den beiden Instanzen gemeinen Daten wird die Methode `Assign` zu Hilfe genommen.
Nach erfolgreicher Migration (insbesondere, wenn die Konsistenz erreicht werden konnte) wird die Referenz auf die neue Instanz der Zielklasse zurückgeliefert.

`Assign<`*`SrcClass`*`>To<`*`DestClass`*`>(`
`ref<`*`SrcClass`*`>:<`*`ScrClass`*`>,`
`ref<`*`DestClass`*`>:<`*`DestClass`*`>)`

Mit dieser Methode werden Daten vom Ausgangsobjekt in das Zielobjekt übertragen. Die Zuweisung beschränkt sich ausschliesslich auf die gemeinsamen Attribute, die in der bezüglich der Vererbungsbeziehungen gemeinsamen Vaterklasse enthalten sind.
Eingeschlossen sind neben den expliziten, benutzerdefinierten Attributen auch implizite, welche aufgrund von Beziehungen, welche die Klasse unterhält generiert worden sind. Damit ist sichergestellt, dass Verbindungen zu anderen Objekten, die vom Ausgangsobjekt ausgehen, trotz der Änderung der Objektidentität bei der Migration zum Zielobjekt erhalten bleiben.
Im Beispiel FIS können Instanzen der Klasse `Developer` zu Instanzen der Klasse `Manager`, welche beide Kinder der Klasse `Employee` sind, migrieren. Der Evolutionsprozess wird vom Besitzerobjekt der Klasse `Company` gesteuert.

```
method body EvolveDeveloperToManager in class Company {
  o2 Manager refManager;

  /* create new Manager */
  refManager = self->AddManager(MercedesType);
  if (refManager == (o2 Manager)0) return ((o2 Manager)0);
```

```
  /* assign shared data */
  self->AssignDeveloperToManager(refDeveloper,refManager);

  /* remove obsolete Developer */
  if (!self->RemoveDeveloper(refDeveloper)) return ((o2
Manager)0);

  return (refManager);
};/* end EvolveDeveloperToManager */
```
Listing 6-37: Implementation der Methode EvolveDeveloperToManager() in der Klasse Company

```
method body AssignDeveloperToManager in class Company {

  /* assign data of parent Employee */
  refManager->m_strName = refDeveloper->m_strName;
  refManager->m_Address = refDeveloper->m_Address;
  refManager->m_refOffice = refDeveloper->m_refOffice;

};/* end AssignDeveloperToManager */
```
Listing 6-38: Implementation der Methode AssignDeveloperToManager() in der Klasse Company

6.2.2.6 Beobachtungsbeziehungen

Beobachtungsbeziehungen sind einseitige Beziehungen von transienten zu persistenten Objekten.
In der Hardydatei können sie über den Eintrag
`type = "observe"`
identifiziert werden. Durch die Definition der Kardinalität
`cardinality = CardinalityConstraint`
kann die Integrität einer Instanz zusätzlich gesichert werden.
Der Eintrag
`name = ObservedCriteria`
stammt aus der Pfeilbeschriftung und hat durch seine rein beschreibende Natur die Aufgabe, das Verständnis und die Lesbarkeit zu fördern.
Im konkreten Fall des Beispiels FIS findet sich in der Hardydatei folgende Eintragung:

```
arc(cardinality = "N*",
  name = "Offices first floor",
  type = "observe",
  id = 86278,
  card = 87936,
  images = [86282],
  connections = [[86242, 86212]],
  arc_image_type = "observe arc").
```
Listing 6-39: Beispiel einer Beobachtungsbeziehung in der Hardydatei

Da dieser Beziehungstyp einseitig ist, kann nicht sichergestellt werden, ob das Beobachtungskriterium zu jeder Zeit erfüllt ist. Löschungen und Erzeugungen können die Ansicht beeinflussen und ungültig machen. Aus diesem Grund ist streng darauf zu achten, dass vor jedem Zugriff über diese Beziehung auf die Objekte in der Datenbasis die Referenzmenge aktualisiert wird.
Daraus folgt, dass für die Hilfsklassen, die Beobachtungsbeziehungen zur Datenbasis unterhalten, zusätzliche Methoden und Attribute generiert werden müssen.
`m_setref<ClassName> : unique set(<ClassName>)`
Attribut zur Speicherung der Referenzen auf die beobachteten Objekte der Datenbasis vom Typ `ClassName`.
Vorsicht! Diese Menge wird durch Manipulationen auf der Datenbasis nicht aktualisiert!

`Update<`*`ClassName`*`>()`
Aktualisiert die Beobachtungsansicht.
Diese Methode ist zur Sicherung der Integrität stets zu Beginn von öffentlichen Methoden aufzurufen.
Die Methode wird nur prototypartig implementiert. Der Entwickler ist angehalten, das Beobachtungskriterium gegebenenfalls an das gewünschte Verhalten anzupassen.
Im konkreten Fall (Beispiel FIS) könnten die generierten Methoden für die Beobachtungsbeziehung von der Klasse `OfficeList` zu `Office` wie folgt implementiert sein:

```
method body UpdateOffice in class OfficeList {
  o2 unique set(Office) refsetOffice; /*result set for OQL
query*/

  /* update view by querying database */
  o2query(refsetOffice,"SELECT e FROM e IN $1",self-
>m_setrefOffice);
  -- TODO: Edit above o2query satisfying your criteria */

};/* end UpdateOffice */
```

Listing 6-40: Die Methode UpdateOffice in der Klasse OfficeList

6.3 Umsetzung eines Transaktionsdiagramms

In einem Transaktionsdiagramm können

- Applikationen,
- Subsysteme,
- Programme,
- Funktionen und
- Transaktionen

als Knoten und

- Aufrufe

als Kanten enthalten sein. Analog zu den Schemadiagrammen sollen bei der Umsetzung grundsätzlich die Knotenelemente einem Objekt zugeordnet werden und anschliessend die Kantenelemente in Beziehungen zwischen den Objekten umgesetzt werden. Im Gegensatz zu den Schemadiagrammen können Transaktionsdiagramme geschachtelt sein (in Hardy können Subsystemknoten zu weiteren Subsystem- oder Transaktionsdiagrammen expandiert werden). Bei der Umsetzung müssen also zusätzlich diese Gliederungsebenen entfernt werden.

6.3.1 Umsetzung der Knoten

Grundsätzlich wird jedem Knoten eine Instanz einer entsprechenden Klasse zugeordnet. Für Hauptprogrammknoten ist die Applikationsdefinition in O_2C zu generieren, während für die Knoten Programm, Funktion und Transaktion prototypartige Implementationen zu erzeugen sind.
Subsystemknoten dienen lediglich der Strukturierung der Ansicht in verschiedene Teilansichten, weshalb diese nur für die visuelle Aufbereitung innerhalb der OSWOOD-Ansichten verarbeitet und für die Implementation herausgefiltert werden.

6.3.1.1 Applikationen

Applikationen sind Programm- und Transaktionssammlungen und bilden in bezug auf eine Datenbasis eine abgeschlossene Funktionseinheit, die aus verschiedenen Lese- und

Schreibtransaktionen besteht. Derartige Konstrukte werden anhand des Typeneintrages

```
type = "main program"
```

erkannt. Der vom Schemaentwerfer definierte Name, repräsentiert durch die Eingeschaft

```
name = "FIS"
```

identifiziert die Applikation und stellt den späteren Zugriff für den Aufruf sicher.
Für das Administrationsbeispiel FIS kann ein entsprechender Eintrag wie folgt gefunden werden:

```
node(name = "FIS",
  type = "main program",
  id = 87784,
  card = 87770,
  images = [87789],
  arcs = [87817, 87821, 87825]).
```

Listing 6-41: Beispiel einer Applikation in der Hardydatei

In der Schemadefinitionssprache O_2C wird eine Applikation nach dem folgenden Syntax definiert (vgl. [Geppert 96, p. 37]):

```
appdef     ::= application AppName
               program
                    programme[,
                    tra_tionen]
end;
programme  ::= program (,program)*
program    ::= [public|private]
ProgramName
tra_tionen ::= tra_tion (,tra_tion)*
tra_tion   ::= [public|private]
TransName(
               (ParamName:ParamType)*):Type
```

Syntaxdiagramm 6-3: Syntax einer Applikationsdefinition in O_2C

Eine Applikationsdefinition besteht also im wesentlichen aus der Angabe des Namens und der darin enthaltenen Programme und Transaktionen. Erstere werden aber nicht mit ihrer gesamten Signatur dargestellt. Daraus folgt, dass der Name des Programmes für eine Applikation eindeutig gewählt werden muss. Insbesondere sind also keine Möglichkeiten für die Überladung gegeben.
Im konkreten Fall (Beispiel FIS) ist die Applikation FIS wie folgt in der generierten Datei zu finden:

```
application FIS
  program
    /* programs */
    public HireDeveloper,
    public FireDeveloper,
    public MakeManager,
    public PrintOfficeList,
    public AssignEmplToOff,

    /* transactions */
    public FireDev(d:Dev) : boolean,
    public HireDev(0:Office) : boolean,
    public MakeMan(d:Dev) : Man,
    public AssignEmplToOff(e:Empl,o:Office) : integer,
    public DeleteOffice() : boolean
end;
```

Listing 6-42: Definition der Applikation FIS

6.3.1.2 Programme

Programme sind Sammlungen von Funktionen und Transaktionen. Sie kapseln also wie in anderen Terminologien, welche diesen Begriff verwenden, Anweisungsblöcke um ein höher abstrahiertes Verhalten modellieren zu können. Derartige Konstrukte werden anhand des Typeneintrages

`type = "program"`

erkannt. Der Entwerfer kann durch die Definition der Attribute

`name = ProgramName,`
`'pseudo code' = PseudoCode`

sowohl den Namen des Programmes als auch die pseudocodeartige Beschreibung des Programmablaufes festlegen.

Für das Administrationsbeispiel FIS kann ein entsprechender Eintrag wie folgt gefunden werden:

```
node(name = "HireDeveloper",
  'pseudo code' = "",
  type = "program",
  id = 87832,
  card = 87770,
  images = [87834],
  arcs = [87859]).
```

Listing 6-43: Beispiel eines Programmes in der Hardydatei

In der Schemadefinitionssprache O_2C wird eine Applikation nach dem folgenden Syntax definiert (vgl. [Geppert 96, p. 59]):

```
programm   ::= program body ProgramName
               in application AppName block
block      ::= { [variables] statement }
variables  ::= (vardef)*
vardef     ::= o2 (TypeName|ClassName)
VarName ;
```

Syntaxdiagramm 6-4: Programmdefinition in O_2C

Eine Programmdefinition besteht also im wesentlichen aus einem Kopfteil, welcher die Verknüpfung zur Applikation herstellt, und dem Anweisungsblock, der von Variablendeklarationen angeführt sein kann.

Eine von OSWOOD generierte Programmimplementation folgt stets einem bestimmten Muster und enthält dementsprechend immer dieselben Abschnitte

Variablendefinition

Für jeden durch eine Kantenverbindung angedeuteten Aufruf einer Funktion oder Transaktion wird eine lokale Variable deklariert, welche den Rückgabewert der entsprechenden Subroutine aufnehmen kann.

Zudem wird für jeden Übergabeparameter eine lokale Variable des geforderten Typs instantiiert.

Vorbedingungen

Der Entwickler ist aufgefordert, in dieser Sektion die für eine erfolgreiche Verarbeitung notwendigen Voraussetzungen zu prüfen.

Initialisierung

Es kann notwendig sein, gewissen in einer Parameterliste eines Subroutinenaufrufes enthaltenen Variablen Initialwerte

zuzuweisen. Solche Zuweisungen sind idealerweise in diesem Abschnitt unterzubringen.

Implementation

Die im Diagramm gezeichneten Aufrufe von Funktionen und Transaktionen werden gemäss der in den Eigenschaften der Pfeile definierten Reihenfolge in den Implementationsteil eingebaut.

Etwaiger Pseudocode wird in einem Kommentarblock ausgegeben, und dem Entwickler zur Ausarbeitung angeboten.

Nachbedingungen

Der Entwickler kann durch die Prüfung von Nachbedingungen gefährliche Situationen erkennen und durch geeignete Massnahmen korrigierend einwirken. Derartige Validierungen sollen im Nachbedingungsteil am Schluss der Programmimplementation untergebracht werden.

Im konkreten Fall (Beispiel FIS) ist die Applikation FIS wie folgt in der generierten Datei zu finden:

```
program body HireDeveloper in application FIS {
  o2 Office refOffice; /* return value of function
SelOffice*/
  o2 boolean refboolean; /* return value of function
HireDev*/
  o2 Office 0;  /* parameter for function HireDev*/

  /* check preconditions */
  -- TODO: Add your validation code here

  /* initialization section */
  -- TODO: Add your initialization code here

  /* implementation section */
  refOffice=SelOffice();
  refboolean=HireDev(0);

  /* check postconditions */
  -- TODO: Add your validation code here

} /* end HireDeveloper */
```

Listing 6-44: Implementation des Programmes HireDeveloper() in der Applikation FIS

6.3.1.3 Funktionen

In Funktionen sind Aufrufe zu anderen Funktionen und/oder zu Transaktionen enthalten. Sie werden anhand des Typeneintrages

`type = "function"`

erkannt. Der Entwerfer kann durch die Definition der Funktionsattribute

`'name (params) return' =` *`FunctionDefinition`*`,`
`'pseudo code' =` *`PseudeCode`*

sowohl deren Name, Parameter und Typ, als auch deren pseudocodeartige Ablaufbeschreibung festlegen.

Für das Administrationsbeispiel FIS kann ein entsprechender Eintrag wie folgt gefunden werden:

```
node('name (params)
return'="SelDev(strName:String,nNumber:Integer):Dev",
  'pseudo code' = "",
  type = "function",
  id = 87841,
  card = 87770,
  images = [87843],
  arcs = [87859, 87867]).
```

Listing 6-45: Beispiel einer Funktion in der Hardydatei

In der Schemadefinitionssprache O_2C wird eine Funktion nach dem folgenden Syntax definiert (vgl. [Geppert 96, p. 59]):

```
funktion   ::= prototyp implement
prototyp   ::= FunctionName ( [params] )
[: Type]
params     ::= param (,param)*
param      ::= ParamName : ParamType
implement  ::= function body
FunctionName block
block      ::= { [variables] statement }
variables  ::= (vardef)*
vardef     ::= o2 (TypeName | ClassName)
VarName ;
```

Syntaxdiagramm 6-5: Functionsdefinition in O_2C

Eine Funktionsdefinition besteht aus einer Prototypdeklaration, die aber im Gegensatz zu den Programmdefinitionen keine Verknüpfung zur Applikation herstellt (Funktionen sind also nicht an Applikationen gebunden), und dem tatsächlichen Funktionskörper. In letzterem ist ein Anweisungsblock enthalten, der von Variablendeklarationen angeführt sein kann.

Eine von OSWOOD generierte Funktionsimplementation folgt stets einem bestimmten Muster und enthält dementsprechend immer dieselben Abschnitte:

Variablendefinition

Für jeden durch eine Kantenverbindung angedeuteten Aufruf einer Funktion oder Transaktion wird eine lokale Variable deklariert, welche den Rückgabewert der entsprechenden Subroutine aufnehmen kann.

Zudem wird für jeden Übergabeparameter eine lokale Variable des geforderten Typs instantiiert.

Der Rückgabewert wird in einer lokalen Variablen zwischengespeichert, die gleichsam in dieser Sektion deklariert wird

Vorbedingungen

Die in der Schnittstelle übergebenen Parameter werden als Vorbedingungen validiert. Handelt es sich bei diesen Daten um Instanzen von Datenbankklassen, lässt sich die Prüfung durch die vordefinierte virtuelle Funktion `IsValid` erledigen.

Initialisierung

Es kann notwendig sein, gewissen Variablen, insbesondere den in einer Parameterliste eines Subroutinenaufrufes enthaltenen, Initialwerte zuzuweisen. Solche Zuweisungen sind idealerweise in diesem Abschnitt unterzubringen.

Implementation

Die im Diagramm gezeichneten Aufrufe von Funktionen und Transaktionen werden gemäss der in den Eigenschaften der Pfeile definierten Reihenfolge in den Implementationsteil eingebaut.

Etwaiger Pseudocode wird in einem Kommentarblock ausgegeben und dem Entwickler zur Ausarbeitung angeboten.

Nachbedingungen

Der Entwickler kann durch die Prüfung von Nachbedingungen gefährliche Situationen erkennen und durch geeignete Massnahmen korrigierend einwirken. Derartige Validierungen sollen im Nachbedingungsteil am Schluss der Programmimplementation untergebracht werden.

Rückgabe

Falls für die Funktion ein Typ definiert worden ist, wird ein entsprechender Rückgabewert an den aufrufenden Programmfaden zurückgeliefert. Dabei hat die Zuweisung dieses Wertes in der Implementation zu erfolgen und dessen Validierung im Rahmen der Nachbedingungen zu erfolgen.

Im konkreten Fall (Beispiel FIS) ist die Applikation FIS wie folgt in der generierten Datei zu finden:

```
function body SelectDev in application FIS {
  o2 Dev refDevReturn; /* return value */
  o2 Man refMan; /* return value of function MakeMan */
  o2 Dev d;  /* parameter for function MakeMan */

  /* check preconditions */
  -- ASSERT: strName.IsValid());
  -- ASSERT: nNumber.IsValid());

  /* initialization section */
  -- TODO: Add your initialization code here

  /* implementation section */
  refMan = MakeMan(d);

  /* check postconditions */
  -- TODO: Add your validation code here

  return (refDevReturn);
} /* end SelectDev */
```

Listing 6-46: Definition der Funktion SelectDev() in der Applikation FIS

6.3.1.4 Transaktionen

In Transaktionen sind die schreibenden Zugriffe auf die persistenten Objekte der Datenbasis verpackt. Sie werden anhand des Typeneintrages

`type = "transaction"`

erkannt. Der Entwerfer kann durch die Definition des Attributes

`'name (params) return' =` *`TransactionDefinition`*,

sowohl dessen Name, Parameter und Typ festlegen. Die gewünschten Aufrufe der schreibenden Klassenmethoden werden als

`calls =` *`Calls`*

in der Hardydatei ausgewiesen.

Für das Administrationsbeispiel FIS kann ein entsprechender Eintrag wie folgt gefunden werden:

```
node('name (params) return' = "MakeManager(d:Developer) :
Manager",
  calls = "Macrohard.AddDev()",
  type = "transaction",
  id = 87850,
  card = 87770,
  images = [87852],
  arcs = []).
```

Listing 6-47: Beispiel einer Transaktion in der Hardydatei

In der Schemadefinitionssprache O_2C wird eine Transaktion nach dem folgenden Syntax definiert (vgl. [Geppert 96, p. 59]):

```
transakt   ::= transaction body
FunctionName
           in application AppName block
block      ::= { [variables]
           transaction
           statement
           (commit | abort)
 }
variables  ::= (vardef)*
vardef     ::= o2 (TypeName|ClassName)
VarName ;
```

Syntaxdiagramm 6-6: Transaktionsdefinition in O_2C

Eine Transaktionsdefinition besteht aus dem tatsächlichen Transaktionskörper (die Prototypdeklaration ist in der Applikation bereits enthalten), der analog zu den Programmdefinitionen die Verknüpfung zur Applikation herstellt und einen Anweisungsblock enthält, welcher von Variablendeklarationen angeführt sein kann. Eine Transaktion muss mit dem Schlüsselwort `transaction` eröffnet werden und je nach Erfolg der Aktionen durch ein `commit` oder `abort` wieder geschlossen werden.

Eine von OSWOOD generierte Transaktionsimplementation folgt stets einem bestimmten Muster und enthält dementsprechend immer dieselben Abschnitte:

Variablendefinition

Für jeden durch eine Kantenverbindung angedeuteten Aufruf einer Funktion oder Transaktion wird eine lokale Variable deklariert, welche den Rückgabewert der entsprechenden Subroutine aufnehmen kann.

Zudem wird für jeden Übergabeparameter eine lokale Variable des geforderten Typs instantiiert.

Der Rückgabewert wird in einer lokalen Variablen zwischengespeichert, die gleichsam in dieser Sektion deklariert wird. Dasselbe gilt für den bool'schen Wert, welcher die Konsistenz der Datenbasis widerspiegelt.

Vorbedingungen

Die in der Schnittstelle übergebenen Parameter werden als Vorbedingungen validiert. Handelt es sich bei diesen Daten um Instanzen von Datenbankklassen, lässt sich die Prüfung durch die vordefinierte virtuelle Funktion `IsValid` erledigen.

Initialisierung

Es kann notwendig sein, gewissen in einer Parameterliste eines Subroutinenaufrufes enthaltenen Variablen Initialwerte zuzuweisen. Solche Zuweisungen sind idealerweise in diesem Abschnitt unterzubringen.

Transaktionsbeginn

Vor dem ersten schreibenden Zugriff auf die persistenten

Objekte der Datenbasis muss eine neue Transaktion eröffnet werden.

Implementation

Die im Diagramm gezeichneten Aufrufe von Funktionen und Transaktionen werden gemäss der in den Eigenschaften der Pfeile definierten Reihenfolge in den Implementationsteil eingebaut.

Etwaiger Pseudocode wird in einem Kommentarblock ausgegeben und dem Entwickler zur Ausarbeitung angeboten.

Nachbedingungen

Es ist für die Erhaltung der Datenintegrität von eminenter Wichtigkeit, dass alle manipulierten Objekte der Datenbasis (vornehmlich diejenigen, auf welche schreibende Methoden angewendet wurden) auf Konsistenz geprüft werden.

Transaktionsschluss

Ist die Konsistenz am Ende einer Transaktion nicht gegeben, muss sie abgebrochen werden. Die aufrufende Routine sollte anschliessend derartige Fehlverarbeitungen anhand des zurückgelieferten Wertes erkennen können und geeignete Massnahmen auslösen, sei dies im Minimalfall lediglich die Unterrichtung des Benutzers.

Rückgabe

War die Objektmanipulation erfolgreich, was anhand des Konsistenzindikators festgestellt werden kann, so werden durch die Auslösung eines Transaktionsschlusses die neuen Attributwerte auf die Datenbasis zurückgeschrieben. Die Transaktion terminiert in diesem Fall mit einem Erfolgswert.

Im konkreten Fall (Beispiel FIS) ist die Applikation FIS wie folgt in der generierten Datei zu finden:

```
transaction body MakeMan in application FIS {
  o2 Man refManReturn; /* return value */
  o2 boolean bIsConsistent; /* local variable indicating
consistency */
  o2 boolean bSuccessful.MakeMan; /* return value of
function .MakeMan */
  o2 Manager refManager; /* object having method .MakeMan */
  o2 Object d;  /* parameter for function .MakeMan */

  /* check preconditions */
  -- ASSERT: d.IsValid());

  /* open a new transaction */
  transaction;

  /* initialization section */
  -- TODO: Add your initialization code here

  /* implementation section */
  bSuccessful.MakeMan = refManager->.MakeMan(d);

  /* check postconditions */
  bIsConsistent = true;
  bIsConsistent &&= refManager.IsConsistent();

  /* if all affected objects are consistent commit
transaction */
  if (bIsConsistent) commit;
  else abort;

  return (refManReturn);
} /* end MakeMan */
```

Listing 6-48: Definition der Transaktion MakeManager() in der Applikation FIS

6.3.2 Umsetzung der Kanten

Die Kanten werden analog zu der Umsetzung der Kanten im Schemadiagram (vgl. "Umsetzung der Kanten", Seite 162) interpretiert. Somit sind auch bei der Verarbeitung in einem Transaktionsdiagramm einzig die Einträge

`type = TypeName`

`connections = [[FromNodeId, ToNodeId]]`

von Interesse. Der Typenname dient zur Erkennung des Kantentyps, während die Variable `connections` eine gerichtete Knotenverbindung darstellt.

6.3.2.1 Aufrufe

Durch Aufrufpfeile werden Verarbeitungssprünge in Subroutinen modelliert, wobei der Ausgangsknoten den rufenden Programmteil und der Zielknoten den aufgerufenen Teil darstellen.

In der Hardydatei können sie über den Eintrag

`type = "call"`

identifiziert werden. Durch die Definition einer Rangzahl, die sich im Eintrag

`number = Number`

wiederfinden lässt, kann für Knoten, von welchen mehrere derartige Pfeile auslaufen, eine Aufrufsequenz angegeben werden.

Im konkreten Fall des Beispiels FIS findet sich in der Hardydatei folgende Eintragung:

```
arc(number = "3",
  type = "call",
  id = 87821,
  card = 87770,
  images = [87824],
  connections = [[87784, 87799]],
  arc_image_type = "call").
```

Listing 6-49: Beispiel eines Aufrufes in der Hardydatei

Aufrufe beschreiben stets Eigenschaften derjenigen Knoten, von denen sie auslaufen und werden dementsprechend nicht direkt in der Schemadefinitionssprache O_2C repräsentiert, sondern immer indirekt im Implementationsteil des beeinflussten Knotens (vgl. "Umsetzung der Knoten", Seite 176).

6.4 Bewertung der Umsetzung

Bei der Umsetzung der Entwurfskonstrukte in die Schemadefintionssprache werden Klassendeklarationen erzeugt. Somit findet der Entwickler als Resultat des Entwurfes ein Schemagerüst, das Klassen, Applikationen, Programme, usw. enthält. OSWOOD geht nun noch einen Schritt weiter und interpretiert das Verhalten, welches hauptsächlich von den Beziehungen abgelesen werden kann, zur Generierung von impliziten Attributen und Methoden.

Der Entwickler kann also nicht nur von einem starren und leeren Anwendungsgerüst, sondern von bereits vorgefertigten Implementationen des Verhaltens ausgehen. Er stellt sich aber dabei die Frage, in welchen Situationen diese Methoden verwendet werden sollen?

Die Frage kann nicht schlüssig beantwortet werden. Vielmehr sollen nachfolgend einige typische Beispiele erläutert sein.

6.4.1 Erzeugen von Instanzen

Für die nachfolgenden Ausführungen soll erneut das Beispiel der Konferenzadministration herangezogen werden: ein Programmkomitee besteht aus Mitgliedern und eingereichten Papieren. Die Papiere werden den Mitgliedern zur Prüfung zugeordnet.

6.4.1.1 Fall I:

Zwischen den Papieren und den Mitgliedern besteht eine „lose“ Verbindung.

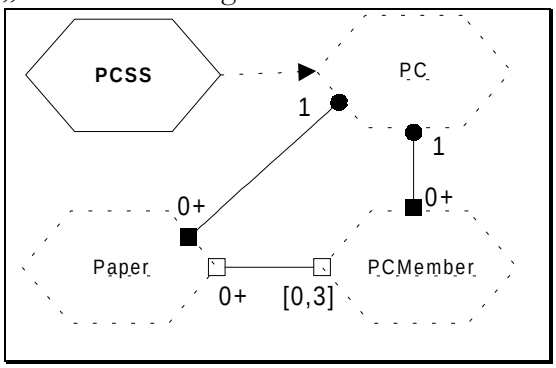


Abbildung 6-1: Schemadiagramm mit „loser“ Verbindung

Das Erzeugen von Instanzen der Klasse `Paper` und `PCMember` ist problemlos und kann einfach über die durch die Komponentenbeziehungen implizierten Methoden `AddPaper`, bzw. `AddPCMember` der Klasse `PC` erreicht werden.

6.4.1.2 Fall II:

Etwas schwieriger wird die Situation, wenn zwischen den Papieren und den Mitgliedern eine „strenge“ (zwingende) Verbindung besteht.

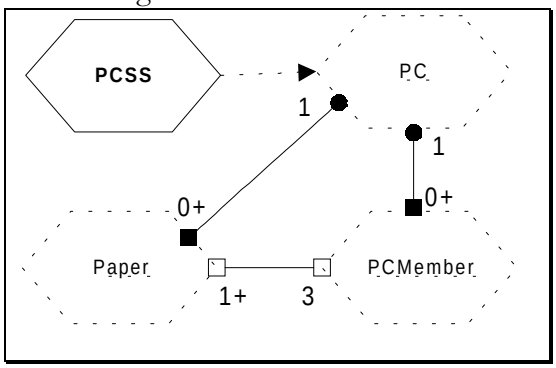


Abbildung 6-2: Schemadiagramm mit „loser“ Verbindung

In dieser Situation ist beispielsweise das Erzeugen von Instanzen der Klasse `PCMember` mit der Methode `AddPCMember` nicht mehr zu bewerkstelligen. Die Methode prüft nämlich am Ende ihrer Ausführung die Konsistenz des erzeugten Objektes, die aufgrund der Kardinalitätsverletzung in der inversen Beziehung zu `Paper` nicht gegeben ist.
Es gibt zwei Auswege aus dieser Situation:
1. Der Entwickler beschafft sich über die Methode `PC::CreatePCMember()` des gewünschten Objektes der Klasse `PC` eine Instanz der Klasse `PCMember`. Anschliessend baut er durch den Aufruf der Methode `PCMember::AttachPaper()` die Beziehung zu einem

bestimmten Papierobjekt auf und fügt das nun konsistente Mitgliedsobjekt der Migliedersammlung mittels der Methode `PC::AddRefPCMember()` hinzu.
2. Der Entwickler distanziert sich von den von OSWOOD generierten Methoden und verwendet die gängigen Datenbankoperationen wie `new`, etc.
In beiden vorgeschlagenen Auswegen liegt zu einem gewissen Zeitpunkt eine Inkonsistenz vor, weshalb der ganze Manipulationsblock in einer Transaktion unterzubringen ist, an deren Ende die Datenintegrität erneut zu gelten hat. Ähnlich verhält es sich bei der Erzeugung von Instanzen der Klasse `Paper`.

6.4.2 Löschen von Instanzen

Analoge Effekte sind bei der Löschung von Instanzen festzustellen. Wird erneut das Schemadiagramm von Abbildung 6-1 betrachtet, sind sowohl Papier- als auch Mitgliedsobjekte auf einfache Weise durch die Verwendung der impliziten Methoden `PC::RemovePaper()`, bzw. `PC::RemovePCMember()` aus der Datenbasis löschbar. Diese Methoden sorgen nämlich automatisch für den konsistenten Abbau der inversen Beziehungen, also für die Erhaltung der Konsistenz.
Anders liegt die Situation wiederum bei der Modellierung von „strengen" inversen Beziehungen (vgl. Abbildung 6-2). Hier versagen die `Remove`-Methoden im Allgemeinen, da die inversen Beziehungen nicht unter Beibehaltung der Datenintegrität abgebaut werden können. Analog stehen auch hier zwei Auswegmöglichkeiten zur Verfügung:
1. Für die Löschung einer Papierinstanz beispielsweise müssen erst die Beziehungen zu den Mitgliedern, die zur Prüfung dieses Papiers vorgesehen sind, aufgelöst werden. Schlägt die Abbaumethode `Paper::DetachPCMember()` fehl, weil ein Mitglied nur dieses einzige Papier begutachtet, muss dieses Mtiglied ebenfalls gelöscht werden (`PC::DestroyPCMember()`). Anschliessend lässt sich das gefragte Papier über die Methode `PC::DestroyPaper()` entfernen.
2. Wiederum steht es dem Entwickler offen, von den generierten Methoden Abstand zu nehmen und auf die bewährten Datenbankoperationen zurückzugreifen.
Die Löschung eines Mitgliedsobjektes erweist sich hingegen als noch problematischer. Das Wegnehmen eines einzigen Mitgliedes kann sich nämlich aufgrund der starken Kardinalitätseinschränkung beliebig fortpflanzen. Es können neben diversen Papieren auch weitere Mitglieder betroffen sein. In diesen Situationen löst man häufig das Problem, indem man anstelle der Löschung von Beziehungen neue Beziehungen zu anderen Objekten aufbaut (Umlage) oder die Kardinalitätseinschränkungen aufweicht.

6.4.3 Fazit

OSWOOD generiert nützliche und wichtige Methodenimplementierungen, die in einer Vielzahl von Anwendungen direkt verwendet werden können. In speziellen Fällen hingegen muss sich der Entwickler genau überlegen, ob die

vorgefertigten Methoden das gewünschte Verhalten auch tatsächlich abbilden. Vielfach steht er dann vor der Entscheidung, entweder gewisse Entwurfsentscheidungen anzupassen, damit die Standardmechanismen greifen können, oder einen gewissen Programmieraufwand zu betreiben, um die Entwurfsaspekte abbilden zu können.

7 Implementation

7.1 Einleitung

In diesem Kapitel soll die Implementation der Umsetzungen aus dem vorigen Kapitel erläutert werden. Dabei sind erst einige allgemeine Eigenschaften der Hardydateien dargestellt und das generelle Vorgehen bei der Interpretation dieser Inputdateien ausgeführt. Darauf werden die Interpretationen der Index-, der Schema- und der Transaktionsdiagramme betrachtet.

Die jeweiligen Beschreibungen sind folgendermassen aufgebaut:

1. Beschreibung der Aufgabe. (Was soll getan werden? Was ist das Ziel?)
2. Beschreibung des Inputs. (Welches sind die Eingangsgrössen? Wie wird mit der Umgebung zusammengearbeitet?)
3. Entwurf der Lösung. (Welches ist die notwendige Klassenstruktur? Was sind die notwendigen Fähigkeiten der Klassen? Welches Verhalten sollen sie implementieren?)
4. Implementation der Lösung. (Wie sind die wichtigen Schlüsselalgorithmen aufgebaut und implementiert?)

Zum Schluss soll die Implementation bewertet werden, ob sich die Wahl der Entwicklungsumgebung als glücklich erwiesen hat, und wie gut das gesteckte Ziel erreicht werden konnte. Zudem sollen die bekannten Probleme und die möglichen zukünftigen Ausbauschritte diskutiert werden.

7.1.1 Werkzeuge

Für die Implementation wurden die folgenden Werkzeuge eingesetzt:

Hardware:

Prozessor
Intel Pentium 90
Arbeitsspeicher
32 MB RAM

Software:

Betriebssystem
Microsoft Windows 95, Microsoft Windows NT 3.51 und Microsoft Windows NT 4.0 alternierend
Entwicklungsumgebung
Microsoft Developer Studio (MSDEV), Version 4.2
Programmiersprache
Microsoft Visual C++ (MSVC), Version 4.2
Klassenbibliothek
Microsoft Foundation Class Library (MFC), Version 4.2

7.2 Umsetzung einer Hardydatei

In allen Umsetzungen muss der von Hardy gelieferte Input, die Hardydatei, gelesen und interpretiert werden. Dazu muss vorerst das Format der Hardydatei erkannt und ein Parser entwickelt werden, welcher derartige Dateien lesen, interpretieren und die gelesenen Informationen für die spätere Verwendung sinnvoll in eine Datenstruktur ablegen kann.

7.2.1 Dateiformat

Eine Hardydatei liegt als Textdatei vor und folgt dem untenstehenden Syntax:

```
file            ::= (item)+
```

```
item            ::= ident( property
(,property)* ).
Property        ::= variable = value
```

Syntaxdiagramm 7-1: Dateiformat einer Hardydatei

Dabei bedeuten die kursiv dargestellten terminalen Symbole folgendes:

ident

Beliebige Zeichenfolge. Identifiziert den Typ einer Sektion. Die Sektionstypen werden von Hardy vergeben und heissen beispielsweise `card`, `item`, `node`, `arc`, usw.

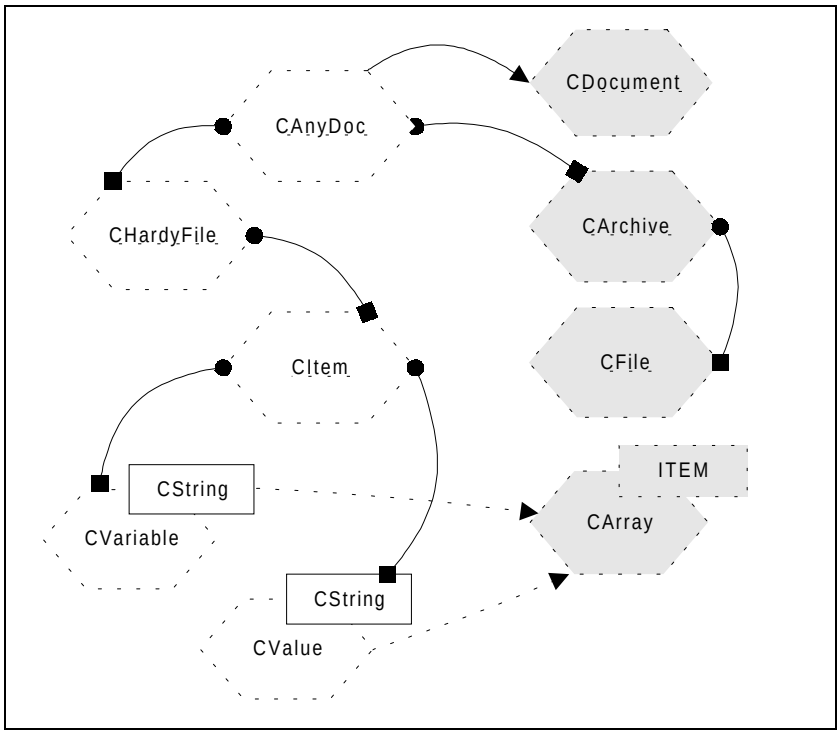
variable

Beliebige Zeichenfolge. Benennt eine bestimmte Eigenschaft der entsprechenden Sektion. Die Variablennamen werden z.T. von Hardy selbst vergeben, etwa `id`, `card` usw., können aber auch aus der Symbolbibliothek stammen wie z.B. `classname` für einen Klassenknoten

value

Beliebige Zeichenfolge. Identifiziert den Wert einer Variablen. Numerische Ausdrücke stehen in numerischer Form, während Texte in Anführungszeichen eingefasst sind. Spezielle Zeichen wie z.B. das Anführungszeichen selbst sind entsprechend durch einen „Backslash" maskiert.

7.2.2 Klassendiagramm



Klassendiagramm 7-1: Klassen für die Umsetzung einer Hardydatei

Klassen von OSWOOD

C*Any*Doc

Verwaltet die Informationen der Hardydatei in Abhängigkeit des Kontextes (z.B. `CIndexDoc` für Indexdateien).

`CHardyFile`

Kapselt die Hardydatei und stellt Funktionen zum Auslesen der Einträge zur Verfügung.

`CItem`
Repräsentiert eine Sektion in einer Hardydatei.
`CVariable`
String-Array zur Speicherung der Variablennamen.
`CValue`
String-Array zur Speicherung der Variablenwerte.
Klassen der MFC
`CDocument`
MFC-Klasse zur Verwaltung von Dokumenten.
`CArchive`
MFC-Klasse zur Verwaltung von Archiven.
`CFile`
MFC-Klasse zur Verwaltung von Dateien.
`CArray`
MFC-Klassenvorlage zur Verwaltung von Array beliebiger (homogener) Typen oder Klassen.
`CString`
MFC-Klasse zur Verwaltung von Texten.

7.2.2.1 CHardyFile

CHardyFile
`CHardyFile(Carchive* parDiagram)` Erzeugt eine Instanz und verbindet diese mit dem Archiv.
`int GetToken(const CString &strDelimiter, CString &strToken) const` Liest Zeichen für Zeichen aus dem Archiv bis entweder das gelesene Zeichen mit einem in `strDelimiter` übergebenen Zeichen übereinstimmt oder das Ende der Datei erreicht worden ist. Damit auch Einträge über mehrere Zeilen gelesen werden können, werden „Carriage-Return-„ und „LineFeed-„Zeichen überlesen. Die bis zur Abbruchbedingung gelesenen Zeichen werden in den Parameter `strToken` geschrieben.
`int GetString(const CString &strDelimiter, CString &strToken) const` Arbeitet analog zu `GetToken`, kann aber auch Einträge über mehrere Zeilen lesen, indem die „Carriage-Return-„ und „LineFeed-„Zeichen überlesen werden, und/oder Einträge, die spezielle Zeichen (insbesondere Zeichen, die in `strDelimiter` vorkommen) enthalten.
`CArchive* m_parHardyFile` Zeigt auf das aktuell geöffnete Archiv.

Klassenbeschreibung 7-1: CHardyFile

```
class CHardyFile {
                  // construction / destruction
                  public:
                        CHardyFile(CArchive* parDiagram);

                  // methods
                  public:
                        int GetToken(const CString
&strDelimiter,CString &strToken) const;
                        void GetString(const char
&chDelimiter,CString &strToken) const;
                        void Move(int nBytes=1);

                  // members
                  public:
                        CArchive* m_parHardyFile;
};
```

Listing 7-1: Definition der Klasse CHardyFile

7.2.3 Lesen einer Hardydatei

```
                        try {
                             while (TRUE) {
(1)
                        nDelIndex=DiagramFile.GetToken("(",strToke
n);
(2)                                  Item.ReadEntry(DiagramFile);
(3)                                  AddItem(Item);
                                     DiagramFile.Move(1);
                        //overread point
                             }//endwhile
                        }//endtry
(4)                     catch(CArchiveException* e) {
(5)                          if (e-
>m_cause!=CArchiveException::endOfFile) {
(6)                                  throw CFatalError("A fatal
error occured ...");
                             }//endif
                             e->Delete();
                        }//endcatch
```

Listing 7-2: Lesen einer Hardydatei

(1) Lesen bis zum Zeichen „(“ und Bezeichner der Sektion speichern.

(2) Die Variabeln und deren Werte auslesen und einer Instanz von `CItem` zuweisen.

(3) Das Element der Sammlung von Elementen hinzufügen.

(4) Wird beim Lesen aus der Hardydatei eine Ausnahme ausgelöst, wird diese gleich anschliessend im `catch`-Block behandelt.

(5) Handelt es sich bei der aufgefangenen Ausnahme um die Angabe, dass die Datei zu Ende gelesen worden ist, wird die Verarbeitung beendet.

(6) In allen anderen Fällen aber deutet die Ausnahme auf einen fatalen Fehler hin und muss dementsprechend weitergeleitet werden.

7.2.3.1 Lesen der Variablen und deren Werte

```
void CItem::ReadEntry(CHardyFile& DiagramFile) {
...
                        while (TRUE) {

                             // read variable name
(1)
                        nDelPos=DiagramFile.GetToken("=",strVariab
le);
(2)                          m_astrVariable.Add(strVariable);

                             // read variable's value
(3)
                        nDelPos=DiagramFile.GetToken(",)",strValue
);
(4)                          m_astrValue.Add(strValue);

                             // check for special handling
(5)                          if
(strVariable==VARIABLE_TYPE_IDENTIFIER_TYPE) {

                        m_strType=strValue.Mid(1,strValue.GetLengt
h()-2);
                                     SetItemType();
                                     SetImageId();
                             }//endif

                             // if parenthesis has been found
sction is read completely
(6)                          if (nDelPos==1) return;
                        }//endwhile
...
}//end ReadEntry
```

Listing 7-3: Lesen einer Sektion

(1) Lesen bis zum Zeichen „=“ und Variablenname speichern.

(2) Variablenname speichern.

(3) Lesen bis zum Zeichen „,“ oder „)“ und Variablenwert speichern.
(4) Variablenwert speichern.
(5) Wurde der Typeneintrag gelesen, so kann dem nicht typisierten Element ein Typ aus der Liste der möglichen Typen zugeordnet werden.
(6) Wurde das Zeichen „)“ gelesen, so ist die Sektion vollständig gelesen worden und die Kontrolle kann an den übergeordneten Prozess zurückgegeben werden.

7.2.3.2 Element hinzufügen

Diese Funktion ist ein Bestandteil des Besitzers der Elementsammlungen, da nur dieser entscheiden kann, welche der gelesenen und erkannten Elementtypen er in seine Sammlung aufnehmen will.
Zudem müssen in Abhängigkeit der weiteren Verwendung der Elemente gewisse Validierungen vorgenommen und weitere Variablen interpretiert und aufbereitet werden (vgl. weitere Abschnitte).

7.3 Umsetzung der Indexdatei

In der Indexdatei sind die Informationen über die Diagramme enthaltenen. OSWOOD soll diese Datei interpretieren, um dem Benutzer ein Navigationsfenster zur Verfügung stellen zu können, in dem er die einzelnen erkannten Diagramme betrachten kann.

7.3.1 Die Indexdatei als Input

Eine Indexdatei liegt als Textdatei vor und folgt dem untenstehenden Syntax:

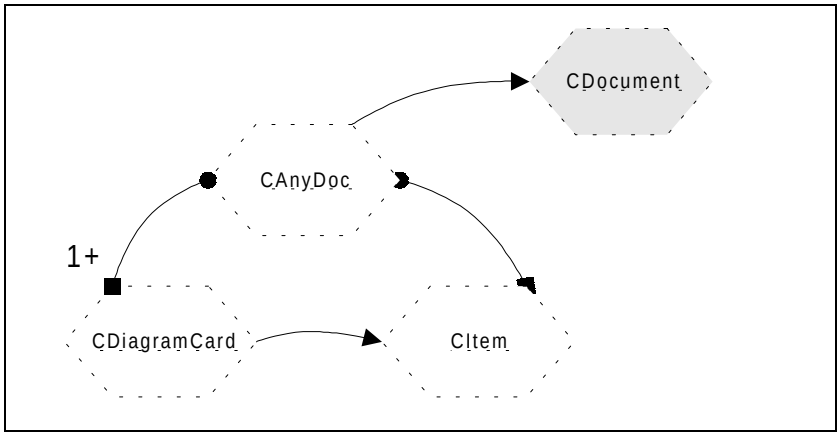
```
index          ::= (item)+
item           ::= ident( property
(,property)* ).
property       ::= variable = value
ident          ::= header | card | item |
link
variable       ::= type | id | card |
filename |
diagram_type | variable
```

Syntaxdiagramm 7-2: Format einer Schemadiagrammdatei

7.3.1.1 Elemente einer Indexdatei

`header`
enthält Informationen über die Indexdatei im Allgemeinen (wird von OSWOOD nicht verwendet).
`card`
beschreibt eine in der Indexdatei referenzierte Diagrammkarte. Insbesondere enthält diese Sektion den Typ und den Namen der Diagrammdatei.
`item`
dient zur Verwaltung der Identifikatoren (wird von OSWOOD nicht verwendet).
`link`
beschreibt die vom Benutzer definierten Verknüpfungen zu den Diagrammen.

7.3.2 Klassendiagramm



Klassendiagramm 7-2: Klassen für die Umsetzung einer Indexdatei

Klassen von OSWOOD:

`CIndexDoc`
verwaltet die Informationen der Indexdatei.

`CDiagramCard`
kapselt eine Diagrammkarte.

`CItem`
speichert die aus der Indexdatei gelesenen Einträge, die nicht weiter typisiert und verwendet werden.

Klassen der MFC

`CDocument`
verwaltet die Informationen eines Dokumentes.

7.3.2.1 CDiagramCard

CdiagramCard

`CdiagramCard(const Citem& Item)`
Erzeugt eine Instanz und übernimmt die Attribute der Vaterklasse.
`void Validate(void)`
Liest den Variablenwert `filename` und weist ihn dem Attribut `m_strFileName` zu.
Bestimmt aufgrund des Varibalenwertes `diagram_type` den Diagrammtyp und speichert diesen im Attribut `m_Type`. Als Diagrammtypen stehen Kontext-, Transaktions- oder Schemadiagramm zur Auswahl. Kann keiner dieser Typen erkannt werden, wird der Typ auf „unbekannt“ gesetzt.

`m_strFileName`
Speichert den Dateinamen und den Pfad der zum Diagramm gehörigen Diagrammdatei (*.DIA)
`m_Type`
Speichert den Typ eines Diagramms

Klassenbeschreibung 7-2: CDiagramCard

```
class CDiagramCard : public CItem {
                    // type defs
                    public:
                         typedef enum {
                                typeContext=0,
                                typeSchema=1,
                                typeTransaction=2,
                                typeUnknown=0
                         } Type;

                    // construction / destruction
                    public:
                         CDiagramCard() {};
```

```
                        CDiagramCard(const CItem& Item) :
CItem(Item) {};

                  // operations
                  public:
                        CDiagramCard& operator=(CDiagramCard&
DiagramCard);

                  // methods
                  public:
                        virtual void Validate(void);
                        CString GetLabel(void);

                  // members
                  public:
                        CString m_strFileName;
                        Type m_Type;
};
```

Listing 7-4: Definition der Klasse CDiagramCard

7.3.3 Lesen der Indexdatei

Die Indexdatei wird gemäss obiger Beschreibung unter gleichzeitigem kontextspezifischen Einfügen und Validieren der Elemente in die entsprechenden Elementsammlungen gelesen. Anschliessend werden gewisse invarianten Bedingungen geprüft.

7.3.3.1 Element hinzufügen

Für jeden Eintrag des Typs `card` wird ein Objekt der Klasse `CDiagramCard` instanziert. Alle anderen Typen werden als Instanzen der Klasse `CItem` gespeichert.

```
void CIndexDoc::AddItem(const CItem& Item) {
(1)               switch (Item.GetItemType()) {
(2)                     case CItem::typeDiagramCard: {
(3)                           CDiagramCard
DiagramCard(Item);
(4)                           DiagramCard.Validate();
(5)
                  m_aDiagramCard.Add(DiagramCard);
                              break;
                              }//endcase

(6)                     default: {
                              CItem anItem(Item);
                              m_aItem.Add(anItem);
                              break;
                              }

                  }//endswitch
}//end AddItem
```

Listing 7-5: Einfügen eines Elementes in das Indexdokument

(1) Weil das Element bereits beim Lesen der Datei typisiert worden ist, kann an dieser Stelle die Information bereits ausgenutzt werden.

(2) Handelt es sich bei dem Element um eine Diagrammbeschreibung,

(3) so wird eine Instanz der Klasse `CDiagramCard` erzeugt (die bereits gelesenen Daten werden von der Vaterklasse in die Kindklasse übertragen),

(4) die Instanz validiert und

(5) in die Sammlung der erkannten Diagramme aufgenommen.

(6) Alle anderen Einträge werden lediglich als untypisierte Elemente der Sammlung für die spätere Reproduktion hinzugefügt.

7.3.3.2 Validierung

Nach der vollständigen Interpretation der Indexdatei liegen die enthaltenen Diagramme als Instanzen vor. Sind keine

derartigen Objekte erkannt worden, handelt es sich bei der gelesenen Datei nicht um eine gültige Indexdatei im Sinne von Hardy. Damit lassen sich folgende Fehlerbedingungen erkennen:
Die gelesene Datei

- ist keine Hardydatei,
- ist leer,
- ist keine Indexdatei oder
- enthält kein Diagramm.

Ist die gelesene Datei im Sinne der obigen Kriterien korrekt, kann sie dennoch für OSWOOD unbrauchbar sein, da sie keine Diagrammtypen enthält, die weiter verwendet werden könnten (Verhalten siehe "Benutzeraktionen", Seite 141).

7.4 Umsetzung der Schemadiagrammdatei

In der Schemadiagrammdatei ist der konzeptuelle Entwurf der Datenbankklassen und die Beziehungen untereinander enthalten. OSWOOD soll diese Datei interpretieren, um dem Benutzer sowohl den Komponentenbaum, die Vererbungshierarchie als auch den Aufbau der einzelnen Klassen und deren Umsetzung in der Schemadefinitionssprache O_2C anzeigen zu können.

7.4.1 Die Schemadiagrammdatei als Input

7.4.1.1 Dateiformat

Eine Schemadiagrammdatei liegt als Textdatei vor und folgt dem untenstehenden Syntax:

```
index      ::= (item)+
item       ::= ident( property
(,property)* ).
property   ::= variable = value
ident      ::= diagram | card | node |
arc |
node_image | arc_image
variable   ::= type | id | card | 'class
name' |
properties | 'persistant name' |
connections | cardinality |
inverse |'cardinality begin' |
'cardinality end' | variable
```

Syntaxdiagramm 7-3: Syntax einer Schemadiagrammdatei

7.4.1.2 Elemente einer Schemadiagrammdatei

`diagram`
enthält Informationen über die Diagrammdatei im Allgemeinen (wird von OSWOOD nicht verwendet).
`card`
enthält die Information über die Diagrammkarte und bildet so die Verbindung zu der Indexdatei. Insbesondere enthält diese Sektion den Typ Diagrammdatei.
`node`
enthält die Informationen über einen Knoten im Diagramm. Neben seinem Typ und den vom Benutzer zugewiesenen Attributen werden auch dessen Verbindungen zu anderen

Knoten definiert. Eine Referenz auf das Element `node_image` legt seine Erscheinungsform und seinen -ort fest.

`arc`
beschreibt einen Verbinder zwischen zwei Knoten. Dieser ist gleichsam typisiert und enthält Verweise auf die Knoten, die durch ihn verbunden werden.

`node_image`
definiert das Aussehen eines Knotens.

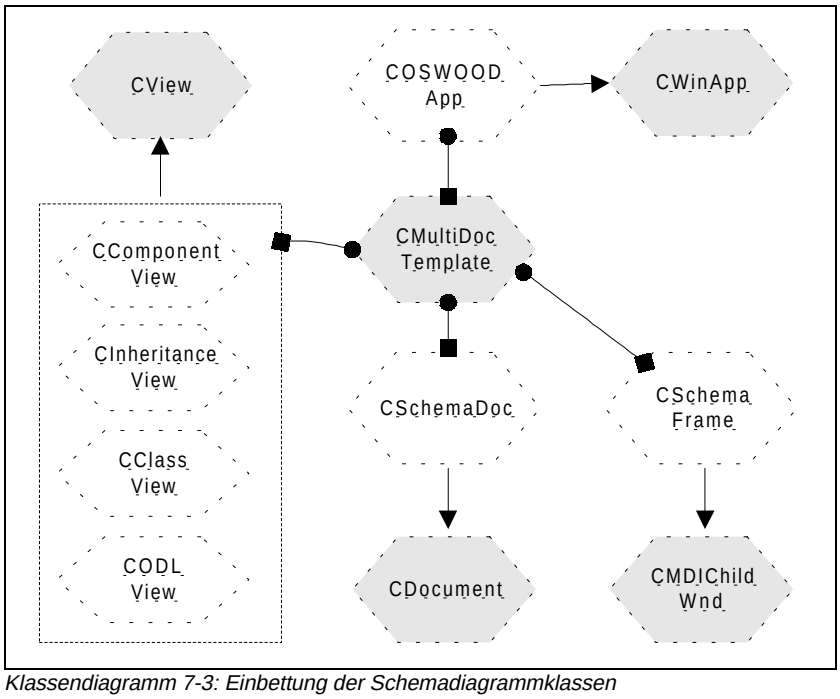
`arc_image`
definiert das Aussehen eines Verbinders.

7.4.2 Ablaufbeschreibung

Der Analyseprozess läuft in mehreren Stufen ab:

1. Lesen der Schemadiagrammdatei (Umsetzung der Elemente in Knoten und Kanten),
2. Validierungen,
3. Aufbau des Vererbungsgraphen,
4. Aufbau der Komponentenbaumes,
5. Umsetzen der Kanten in Beziehungen (Komponenten, inverse, Bbeobachtung, Evolution),
6. Erzeugung der Klassendefinitionen und Methodenimplementierungen,
7. Ergänzen der Klassen mit vordefinierten Methoden und
8. Schreiben der ODL-Datei.

7.4.3 Einbettung



Klassendiagramm 7-3: Einbettung der Schemadiagrammklassen

Klassen von OSWOOD

`COSWOODApp`
Applikationsklasse von OSWOOD. Verwaltet das Hauptfenster und die Dokumententypen. Einzige Klasse, welche eine globale Instanz besitzt.

`CSchemaDoc`
verwaltet die Informationen eines Schemadiagramms.
`CSchemaFrame`
verwaltet das Dokumentenfenster.
`CComponentView`
verwaltet die Komponentenansicht.
`CInheritanceView`
verwaltet die Vererbungsansicht.
`CClassView`
verwaltet die Klassenansicht.
`CODLView`
verwaltet die ODL-Ansicht.
Klassen der MFC
`CWinApp`
Applikationsklasse.
`CMultiDocTemplate`
kapselt einen bestimmten Dokumententyp, der durch seine Dokumenten-, Rahmen- und Ansichtsklasse identifiziert wird.
`CMDIChildWnd`
verwaltet das Kindfenster in einer MDI-Applikation.
`CView`
bildet die Basisklasse der Ansichtsklassen.
`CDomument`
verwaltet die Informationen eines Dokumentes.

7.4.3.1 CSchemaDoc

Die Klasse speichert sämtliche Informationen, die in einem Schemadiagramm enthalten sind und sämtliche Resultate der Interpretation. Sie steuert den Ablauf und reagiert auf Benutzeraktionen.

CSchemaDoc : public CDocument
`void ReadDiagram(CHardyFile& HardyFile)` Liest eine Hardydatei (vgl. "Lesen einer Hardydatei", Seite 193)
`void CreateItemList(CHardyFile& HardyFile)` Erstellt die Liste der Elemente
`void AddItem(const CItem& Item)` Fügt ein bestimmtes Element der entsprechenden Sammlung hinzu. Ruft für das Element insbesondere die `Validate`-Methode auf
`void ValidatePrimitives(void) const` Führt gewisse Validierungen durch (vgl. "Validierungen", Seite 213)
`void CreateInheritanceHierarchy(void)` Erzeugt den Vererbungsgraphen (vgl. "Aufbau des Vererbungsgraphen", Seite 217)
`void CreateComponentTree(void)` Erzeug die Komponentenbäume (vgl. "Aufbau des Komponentenbaumes", Seite 222) und erstellt implizit die Komponentenbeziehungen
`void CreateInverseRelations(void)` Erzeugt die inversen Beziehungen
`void CreateObserverRelations(void)`

Erzeugt die Beobachtungsbeziehungen

`void CreateEvolutions(void)`
Erzeugt die Evolutionsbeziehungen

`void CreateDefinition(void)`
Erzeugt die Definitionen und Implementationen der verschiedenen Elemente für die ODL-Ansicht

`void WriteO2CFile(CArchive& arFile)`
Schreibt die verschiedenen Codefragmente gesammelt in eine vom Benuzter angegebene Datei

---

`CArray<CItem,CItem&> m_aItem`
Sammlung der nicht näher typisierten Einträge in der Hardydatei (Reproduktion)

`CArray<CdatabaseClass,CDatabaseClass&> m_aDatabaseClass`
Sammlung der Datenbankklassen

`CArray<CEntryPoint,CEntryPoint&> m_aEntryPoint`
Sammlung der Einstiegspunkte

`CArray<CabstractClass,CAbstractClass&> m_aAbstractClass`
Sammlung der abstrakten Klassen

`CArray<CClassUtility,CClassUtility&> m_aClassUtility`
Sammlung der Hilfsklassen

`CArray<CNote,CNote&> m_aNote`
Sammlung der Notizen

`CArray<CinheritanceArc,CInheritanceArc&> m_aInheritanceArc`
Sammlung der Vererbungsbeziehungen

`CArray<CcomponentArc,CComponentArc&> m_aComponentArc`
Sammlung der Komponentenbeziehungen

`CArray<CinstantiationArc,CInstantiationArc&> m_aInstantiationArc`
Sammlung der Instantiierungsbeziehungen

`CArray<CobserverArc,CObserverArc&> m_aObserverArc`
Sammlung der Beobachtungsbeziehungen

`CArray<CinverseRelationArc,CInverseRelationArc&> m_aInverseRelationArc`
Sammlung der inversen Beziehungen

`CArray<CevolutionArc,CEvolutionArc&> m_aEvolutionArc`
Sammlung der Evolutionsbeziehungen

`CArray<CNoteConnectorArc,CNoteConnectorArc&> m_aNoteConnectorArc`
Sammlung der Notizenverbinder

`CInheritanceGraph m_InheritanceGraph`
Vererbungsgraph als Instantiierung einer `CGraph` Klassenvorlage (vgl. “Die Graphklassen”, Seite 213)

`CComponentTree m_ComponentTree`

Komponentenbaum als Instantiierung einer `CGraph` Klassenvorlage (vgl. “Die Graphklassen”, Seite 213)

Klassenbeschreibung 7-3: CSchemaDoc

```
class CSchemaDoc : public CDocument {
// Methods
private:
                  // reading methods
                  void ReadDiagram(CHardyFile& HardyFile);
                  void CreateItemList(CHardyFile&
HardyFile);
                  void AddItem(const CItem& Item);

                  void ValidatePrimitives(void) const;

                  void CreateInheritanceHierarchy(void);
                  void AddDerivedClasses(int
nCurrentParent,HITEM hParent,CArray<int,int&>&
anAddedClassId);

                  void CreateComponentTree(void);
                  void AddComponents(CClass*
pParentClass,CArray<int,int&>& anVisitedClass,int
nParentId,HITEM hParent);

                  void CreateInverseRelations(void);
                  void CreateObserverRelations(void);
                  void CreateEvolutions(void);

                  void CreateDefinition(void);
                  CString GetClassParentString(CClass*
pClass);

                  // writing methods
                  void WriteO2CFile(CArchive& arFile);
                  void WriteClass(CClass* pClass,CArchive&
arFile,CString strTypeName);

                  //utilities
                  CClass* GetClass(int nId);
                  BOOL IsElementOf(CArray<int,int&>&
anArray,int& nElement);
                  CGraphNode<CClass>*
IsParentElementOf(CGraphNode<CClass>*
pItem,CArray<int,int&>& anArray);
                  BOOL IsChildElementOf(CGraphNode<CClass>*
pItem,CArray<int,int&>& anArray);

// Attributes
public:
                  // item list
                  CArray<CItem,CItem&> m_aItem;

                  // node lists
                  CArray<CDatabaseClass,CDatabaseClass&>
m_aDatabaseClass;
                  CArray<CEntryPoint,CEntryPoint&>
m_aEntryPoint;
                  CArray<CAbstractClass,CAbstractClass&>
m_aAbstractClass;
                  CArray<CClassUtility,CClassUtility&>
m_aClassUtility;
                  CArray<CNote,CNote&> m_aNote;

                  // arc lists
                  CArray<CInheritanceArc,CInheritanceArc&>
m_aInheritanceArc;
                  CArray<CComponentArc,CComponentArc&>
m_aComponentArc;
                  CArray<CInstantiationArc,CInstantiationArc
&> m_aInstantiationArc;
                  CArray<CObserverArc,CObserverArc&>
m_aObserverArc;
                  CArray<CInverseRelationArc,CInverseRelatio
nArc&> m_aInverseRelationArc;
                  CArray<CEvolutionArc,CEvolutionArc&>
m_aEvolutionArc;
                  CArray<CNoteConnectorArc,CNoteConnectorArc
&> m_aNoteConnectorArc;

                  // inheritance graph
```

```
CInheritanceGraph m_InheritanceGraph;
CComponentTree m_ComponentTree;
```

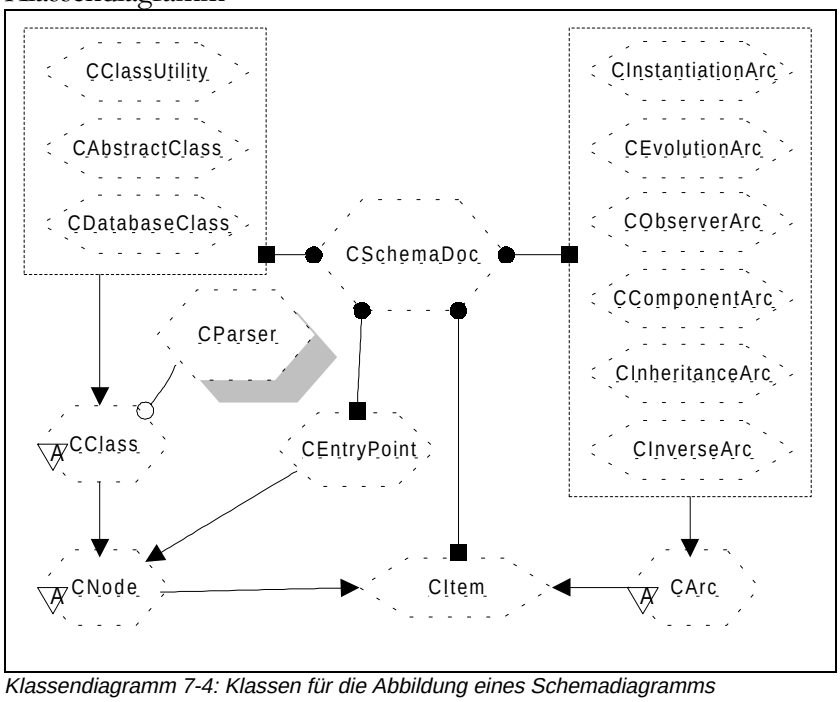
Listing 7-6: Klassendefinition von CSchemaDoc

7.4.4 Lesen der Schemadiagrammdatei

Das Lesen der Schemadiagrammdatei läuft in den folgenden Schritten ab:

- Analyse der Hardydatei (Parsing),
- Erzeugen der Elemente,
- Analyse der Eigenschaften der Klassen (Attribute und Methoden),
- Erzeugung der Implementation der expliziten Methoden,
- Umsetzung der Contraints,
- Umsetzung der Attribute und der Schlüssel
- Analyse und Umsetzung der Kardinalitätsangaben an Kanten.

Klassendiagramm



Klassendiagramm 7-4: Klassen für die Abbildung eines Schemadiagramms

Bemerkung: geht ein Pfeil von einer Gruppe aus oder führt ein Pfeil zu einer Gruppe, so bedeute dies, dass jedes Gruppenelement eine derartige Beziehung eingeht.

Klassen von OSWOOD

`CItem`

speichert die für alle Elemente gemeinsamen Informationen wie Typ, Name, etc. Verwaltet die nicht einer der nachfolgenden Klassen zuordbaren Elemente für Reproduktionszwecke.

`CNode`

Abstrakte Klasse für die Abbildung eines Diagrammknotens.

`CEntryPoint`

widerspiegelt einen persistenten Einstiegspunkt.

`CClass`

Abstrakte Klasse für die Zusammenfassung aller klassenspezifischen Eigenschaften und Fähigkeiten.

`CParser`

ist eine Hilfsklasse und stellt Methoden für das syntaktische

Analysieren von Klasseneingeschaften zu Verfügung. Wird aus diesem Grund nur von `CClass` verwendet.

`CDatabaseClass`
Darstellung einer Datenbankklasse.

`CAbstractClass`
Darstellung einer abstrakten Klasse.

`CClassUtility`
Darstellung einer Hilfsklasse.

`CArc`
Abstrakte Klasse für die Zusammenfassung aller kantenspezifischen Eigenschaften.

`CComponentArc`
Darstellung einer Kante des Typs „Komponente“.

`CInheritanceArc`
Darstellung einer Kante des Typs „Vererbung“.

`CInverseRelationArc`
Darstellung einer Kante des Typs „inverse Beziehung“.

`CInstantiationArc`
Darstellung einer Kante des Typs „Instantiierung“.

`CObserverArc`
Darstellung einer Kante des Typs „Beobachtung“.

`CEvolutionArc`
Darstellung einer Kante des Typs „Evolution“.

Nachfolgend seien einige dieser Klassen (Basisklassen) näher betrachtet:

7.4.4.1 CItem

CItem
`CItem(const CItem& Item)` Erzeugt eine Instanz und übernimmt die Attribute aus der im Parameter referenzierten Klasse. Dieser Konstruktor wird verwendet, um die Spezialisierungen vornehmen zu können.
`Void ReadEntry(CHardyFile& DiagramFile)` Liest die Einträge aus einer Sektion in der Hardydatei (vgl. “Lesen der Variablen und deren Werte”, Seite 193) und speichert diese in den Attributen `m_astrVariable` und `m_astrValue`
`void SetItemClass(const CString& strId)` Klassifiziert das Element auf Grund seines Sektionstyps und speichert diesen im Attribut `m_Class`
`void GetItemClass(void)` Liefert den Sektionstyp
`void SetItemType(void)` Typisiert das Element auf Grund des gelesenen Namenseintrages, welcher im Attribut `m_strType` gespeichert, ist (vgl. `ReadEntry`) und setzt das Attribut `m_nType`
`void GetItemType(void)` Liefert den Typ des Elementes
`void SetImageId(void)` Setzt die Ikone für die Visualisierung des Elementes in der Baumansicht (`m_nImageId`)

`void GetVariableValue(const CString& strIdentifier, CString& strValue)`
Sucht aus den Variablensammlungen denjenigen Eintrag, welcher mit dem Parameter `strIdentifier` übereinstimmt und liefert den entsprechenden Wert zurück. Diese Methode wird von den Derivaten von `CItem` innerhalb der Methode `Validate()` verwendet, um weitere elementspezifische Informationen auszuwerten.

`Void GetVariableValue(const CString& strIdentifier, int& nValue)`
Entspricht der obigen Methode für numerische Eigenschaften

`virtual void Validate(void)`
Validierungsmethode für das Auslesen der relevanten Information und der Interpretation. Liest insbesondere den Indentifikations- und den Karteneintrag und speichert diese Werte in den Attributen `m_nId` und `m_nCard`.
Diese Methode wird von den Derivaten überschrieben und den Bedürfnissen angepasst

---

`int m_nId`
Eindeutiger (von Hardy vergebener) Elementidentifikator. Wird vor allem für die Auswertung der Kanteninformation verwendet.

`int m_nCard`
Eindeutiger (von Hardy vergebener) Kartenidentifikator. Wird für die Erkennung der Elementexpansion (Transaktionsdiagramme) verwendet

`CString m_strType`
Identifikationsstring für die Erkennung des Elementtyps

`Carray<CString,CString&> m_astrVariable`
Sammlung der Variablen

`CArray<CString,CString&> m_astrValue`
Sammlung der Variablenwerte

Klassenbeschreibung 7-4: CItem

```
class CItem {
                    // construction/destruction
                    public:
                         CItem() {};
                         CItem(const CItem& Item);

                    // types
                    public:
                         typedef enum {
                                typeUnknownClass,
                                typeDiagram,
                                typeCard,
                                typeNode,
                                typeArc,
                                typeNodeImage,
                                typeArcImage
                         } ItemClass;
                         typedef enum {
                                typeUnknown,
                                typeDatabaseClass,
                                typeEntryPoint,
                                typeClassUtility,
                                typeAbstractClass,
                                typeNote,
                                typeInheritance,
                                typeComponent,
                                typeObserver,
                                typeInstantiation,
                                typeEvolution,
                                typeInverseRelation,
                                typeNoteConnector,
```

```
                                        typeDiagramCard,
                                        typeMainProgram,
                                        typeSubSystem,
                                        typeProgram,
                                        typeFunction,
                                        typeTransaction,
                                        typeCall
                                  } ItemType;
                                  typedef enum {
                                        imageUnknown=IDB_UNKNOWN,

                            imageDatabaseClass=IDB_DATABASECLASS,

                            imageEntryPoint=IDB_ENTRYPOINT,

                            imageClassUtility=IDB_CLASSUTILITY,

                            imageAbstractClass=IDB_ABSTRACTCLASS,
                                  } ImageId;

                            // operations
                            public:
                                  CItem& operator=(const CItem& Item);

                            // methods
                            public:
                                  void ReadEntry(CHardyFile&
DiagramFile);
                                  void SetItemClass(const CString&
strId);
                                  ItemClass GetItemClass(void) const;
                                  void SetItemType(void);
                                  ItemType GetItemType(void) const;
                                  void SetImageId(void);
                                  BOOL GetVariableValue(const CString&
strIdentifier,CString& strValue) const;
                                  BOOL GetVariableValue(const CString&
strIdentifier,int& nValue) const;
                                  virtual void Validate(void);

                            // members
                            public:
                                  int m_nId;
                                  int m_nCard;
                                  CString m_strType;
                                  ItemClass m_Class;
                                  ItemType m_nType;
                                  ImageId m_nImageId;
                                  CArray<CString,CString&>
m_astrVariable;
                                  CArray<CString,CString&> m_astrValue;
};//end class CItem
```

Listing 7-7: Klassendefinition von CItem

7.4.4.2 CNode

CNode : public CItem
`virtual void Validate(void)` Liest aus dem Kanteneintrag die Kantenidentifikatoren und füllt diese Werte in die Sammlung `m_anArc` ab.
`CArray<int,int&> m_anArc` Sammlung der KantenIndentifikatoren

Klassenbeschreibung 7-5: CNode

```
class CNode : public CItem {
                            // construction/destruction
                            public:
                                  CNode() {};
                                  CNode(const CItem& Item) :
CItem(Item) {};

                            // operations
                            public:
                                  CNode& operator=(CNode& Node);

                            // methods
                            public:
                                  virtual void Validate(void);
```

```
                    // members
                    private:
                         CArray<int,int&> m_anArc;
};  // end class CNode
```

Listing 7-8: Klassendefinition von CNode

7.4.4.3 CEntryPoint

CEntryPoint : public CNode
`virtual void Validate(void)` Liest den persistenten Namen aus den Einträgen und speichert diesen im Attribut `m_strPersistentName`.

`CString m_strPersistentName`
Persistenter Name des Einstiegspunktes
`CString m_strDefinition`
Definitionsstring des Einstiegspunktes in O_2C.

Klassenbeschreibung 7-6: CEntryPoint

```
class CEntryPoint : public CNode {
                    // construction/destruction
                    public:
                         CEntryPoint() {};
                         CEntryPoint(const CItem& Item) :
CNode(Item) {};

                    // operations
                    public:
                         CEntryPoint& operator=(CEntryPoint&
EntryPoint);

                    // methods
                    public:
                         virtual void Validate(void);

                    // attributes
                    public:
                         CString m_strPersistentName;
                         CString m_strDefinition;
};  //end class CEntryPoint
```

Listing 7-9: Klassendefinition von CEntryPoint

7.4.4.4 CClass

CClass : public CNode
`virtual void Validate(void)` Liest den Klassennamen und die Eigenschaften und ruft die Methode `ParsePropertyEntry()` auf, um die Eigenschaften zu analysieren.
`void ParsePropertyEntry(void)` Teilt den Eigenschaftsstring in Substrings (jede Eigenschaft muss auf einer neuen Zeile definiert werden, vgl. "Definition der Datenbankklassen, abstrakten Klassen und Hilfsklassen", Seite 155) und ruft aufgrund eines bestimmten Eigenschaftsmerkmals die spezifische Analysefunktion auf.
`void ParseMethodProperty(CString strProperty)` Legt ein neues Objekt der Klasse `CexplicitMethod` an und ruft dessen `Create`-Methode auf. Diese ruft die entsprechende Analysemethode des `CParser`-Objektes auf und erzeugt die Implementation in O_2C. Anschliessend wird das neue Objekt in der Sammlung `m_aExplicitMethod` gespeichert.
`void ParseAttributeProperty(CString strProperty)` Legt ein neues Objekt der Klasse `CexplicitAttribute` an und ruft dessen `Create`-Methode auf. Diese ruft die entsprechende

Analysemethode des `CParser`-Objektes auf. Anschliessend wird das neue Objekt in der Sammlung `m_aExplicitAttribute` gespeichert.

`void ParseKeyAttributeProperty(Cstring strProperty)`
Legt ein neues Objekt der Klasse `CkeyAttribute` an und ruft dessen `Create`-Methode auf. Diese ruft die entsprechende Analysemethode des `CParser`-Objektes auf. Anschliessend wird das neue Objekt in der Sammlung `m_aKeyAttribute` gespeichert.

`void ParseConstraintProperty(CString strProperty)`
Ruft die entsprechende Analysemethode des `CParser`-Objektes auf und speichert das Resultat in der Sammlung `m_astrConstraint`.

`CString m_strClassName`
Name der Klasse

`CString m_strProperties`
Definitionsstring der Eigenschaften. Wird nur für Reproduktionszwecke gespeichert.

`CArray<CExplicitMethod,CExplicitMethod&> m_aExplicitMethod`
Sammlung der expliziten Methoden

`CArray<CExplicitAttribute,CExplicitAttribute&> m_aExplicitAttribute`
Sammlung der expliziten Attribute

`CArray<CKeyAttribute,CKeyAttribute&> m_aKeyAttribute`
Sammlung der Schlüsselattribute

`CArray<CString,CString&> m_astrConstraint`
Sammlung der Konsistenzbedingungen

Klassenbeschreibung 7-7: CClass

```
class CClass : public CNode {
                    // construction/destruction
                    public:
                         CClass() {};
                         CClass(const CItem& Item) :
CNode(Item) {};

                    // operations
                    public:
                         CClass& operator=(CClass& Class);

                    // methods
                    public:
                         virtual void Validate(void);

                         void ParsePropertyEntry(void);
                         void ParseMethodProperty(CString
strProperty);
                         void ParseConstraintProperty(CString
strProperty);
                         void
ParseKeyAttributeProperty(CString strProperty);
                         void ParseAttributeProperty(const
CString& strProperty);

                         void CreateComponentRelation(CClass*
pClass,CString strCardinality,CString
strCardinalityInverse);
                         void CreateInverseRelation(CClass*
pClass,CString strCardinality,CString
strCardinalityInverse);
                         void CreateObserverRelation(CClass*
pClass,CString strCardinality,CString
strCardinalityInverse);
                         void CreateEvolutionRelation(CClass*
pFromClass,CClass* pToClass,CClass* pParentClass);
```

```
                        void CreateVirtualMethods(void);
                        void CreateDefinition(CString
strParents);

                        void Show(HTREEITEM
hClassRoot,CTreeCtrl& TreeCtrl,UINT nMask,CClassView*
pClassView);
                  private:
                        CString GetLabel(void);

                  // attributes
                  public:
                        // user defined values
                        CString m_strClassName;
                        CString m_strProperties;

                        // user defined members

                  CArray<CExplicitAttribute,CExplicitAttribu
te&> m_aExplicitAttribute;

                  CArray<CExplicitMethod,CExplicitMethod&>
m_aExplicitMethod;

                        // virtual functions inherited from
DBObject

                  CArray<CVirtualMethod,CVirtualMethod&>
m_aConstructor;

                  CArray<CVirtualMethod,CVirtualMethod&>
m_aVirtualMethod;

                        // relations

                  CArray<CComponentRelation,CComponentRelati
on&> m_aComponentRelation;

                  CArray<CInverseRelation,CInverseRelation&>
m_aInverseRelation;

                  CArray<CObserverRelation,CObserverRelation
&> m_aObserverRelation;

                        // evolution

                  CArray<CEvolutionRelation,CEvolutionRelati
on&> m_aEvolutionRelation;

                        // key attributes
                        CArray<CKeyAttribute,CKeyAttribute&>
m_aKeyAttribute;

                        // constraints
                        CArray<CString,CString&>
m_astrConstraint;

                        // class definition string in ODL
                        CString m_strParents;
                        CString m_strDefinition;
}; // end class CClass
```

Listing 7-10: Klassendefinition von CClass

7.4.4.5 Derivate der Klasse CClass

Die weiteren Derivate der Klasse `CClass` unterschieden sich kaum von der Basisklasse. Einzig die Klasse `CDatabaseClass` erhält ein zusätzliches Attribut `CArray<CKeyAttribute,CKeyAttribute&> m_aKeyAttribute` für die Speicherung der Schlüsselattribute. Der Vollständigkeit halber seien aber die Klassendefinitionen trotzdem dargestellt:

```
class CDatabaseClass : public CClass {
                  // construction/destruction
                  public:
                        CDatabaseClass() {};
                        CDatabaseClass(const CItem& Item) :
CClass(Item) {};
```

```
                    // operations
                    public:
                         CDatabaseClass&
operator=(CDatabaseClass& DatabaseClass);

                    // attributes
                    public:
                         // user defined members
                         CArray<CKeyAttribute,CKeyAttribute&>
m_aKeyAttribute;
}; //end class CDatabaseClass
```

Listing 7-11: Klassendefinition von CDatabaseClass

```
class CAbstractClass : public CClass {
                    // construction/destruction
                    public:
                         CAbstractClass() {};
                         CAbstractClass(const CItem& Item) :
CClass(Item) {};

                    // operations
                    public:
                         CAbstractClass&
operator=(CAbstractClass& AbstractClass);
}; //end class CAbstractClass
```

Listing 7-12: Klassendefinition von CAbstractClass

```
class CClassUtility : public CClass {
                    // construction/destruction
                    public:
                         CClassUtility() {};
                         CClassUtility(const CItem& Item) :
CClass(Item) {};

                    // operations
                    public:
                         CClassUtility&
operator=(CClassUtility& ClassUtility);
}; //end class CClassUtility
```

Listing 7-13: Klassendefinition von CClassUtility

7.4.4.6 CArc

CArc : public CItem
`virtual void Validate(void)` Liest aus der Verbindungseigenschaft den Identifikator des Ausgangs- und des Zielknotens aus und weist diese Werte den Attributen `m_nFromNode` und `m_nToNode` zu.

`int m_nFromNode`
Identifikator des Ausgangsknotens
`int m_nToNode`
Identifikator des Zielknotens

Klassenbeschreibung 7-8: CArc

```
class CArc : public CItem {
                    // construction/destruction
                    public:
                         CArc() {};
                         CArc(const CItem& Item) : CItem(Item)
{};

                    // operations
                    public:
                         CArc& operator=(CArc& Arc);

                    // methods
                    public:
                         virtual void Validate(void);

                    // attributes
                    public:
                         int m_nFromNode;
                         int m_nToNode;
}; //end class CArc
```

Listing 7-14: Klassendefinition von CArc

7.4.4.7 Derivate der Klasse CArc

Die verbindungstypspezifischen Derivate der Klasse `CArc` unterscheiden sich kaum von ihrer Basisklasse. Die Komponenten- und die inverse Beziehung speichern zusätzlich die vom Entwerfer erfassten Kardinalitätseigenschaften und müssen diese entsprechend in ihrer `Validate`-Methode aus den Variablenwerten herausziehen. Um der Vollständigkeit zu genügen, sollen die Klassendefinitionen nachfolgend abgebildet werden:

```
class CComponentArc : public CArc {
                  // construction/destruction
                  public:
                        CComponentArc() {};
                        CComponentArc(const CItem& Item) :
CArc(Item) {};

                  // operations
                  public:
                        CComponentArc&
operator=(CComponentArc& ComponentArc);

                  // methods
                  public:
                        virtual void Validate(void);

                  // attributes
                  public:
                        CString m_strCardinality;
                        CString m_strCardinalityInverse;
}; //end class CComponentArc
```

Listing 7-15: Klassendefinition von CComponentArc

```
class CInverseRelationArc : public CArc {
                  // construction/destruction
                  public:
                        CInverseRelationArc() {};
                        CInverseRelationArc(const CItem&
Item) : CArc(Item) {};

                  // operations
                  public:
                        CInverseRelationArc&
operator=(CInverseRelationArc& InverseRelationArc);

                  // methods
                  public:
                        virtual void Validate(void);

                  // attributes
                  public:
                        CString m_strCardinality;
                        CString m_strCardinalityInverse;
}; //end class CInverseRelationArc
```

Listing 7-16: Klassendefinition von CInverseRelationArc

```
class CInstantiationArc : public CArc {
                  // construction/destruction
                  public:
                        CInstantiationArc() {};
                        CInstantiationArc(const CItem& Item)
: CArc(Item) {};

                  // operations
                  public:
                        CInstantiationArc&
operator=(CInstantiationArc& InstantiationArc);

};  //end class CInstantiationArc
```

Listing 7-17: Klassendefinition von CInstantiationArc

```
class CObserverArc : public CArc {
                  // construction/destruction
                  public:
                        CObserverArc() {};
                        CObserverArc(const CItem& Item) :
CArc(Item) {};

                  // operations
                  public:
```

```
                    CObserverArc& operator=(CObserverArc&
ObserverArc);

}; //end class CObserverArc
```
Listing 7-18: Klassendefinition von CObserverArc
```
class CEvolutionArc : public CArc {
                // construction/destruction
                public:
                     CEvolutionArc() {};
                     CEvolutionArc(const CItem& Item) :
CArc(Item) {};

                // operations
                public:
                     CEvolutionArc&
operator=(CEvolutionArc& EvolutionArc);

}; //end class CEvolutionArc
```
Listing 7-19: Klassendefinition von CEvolutionArc
```
class CInheritanceArc : public CArc {
                // construction/destruction
                public:
                     CInheritanceArc() {};
                     CInheritanceArc(const CItem& Item) :
CArc(Item) {};

                // operations
                public:
                     CInheritanceArc&
operator=(CInheritanceArc& InheritanceArc);

}; //end class CInheritanceArc
```
Listing 7-20: Klassendefinition von CInheritanceArc

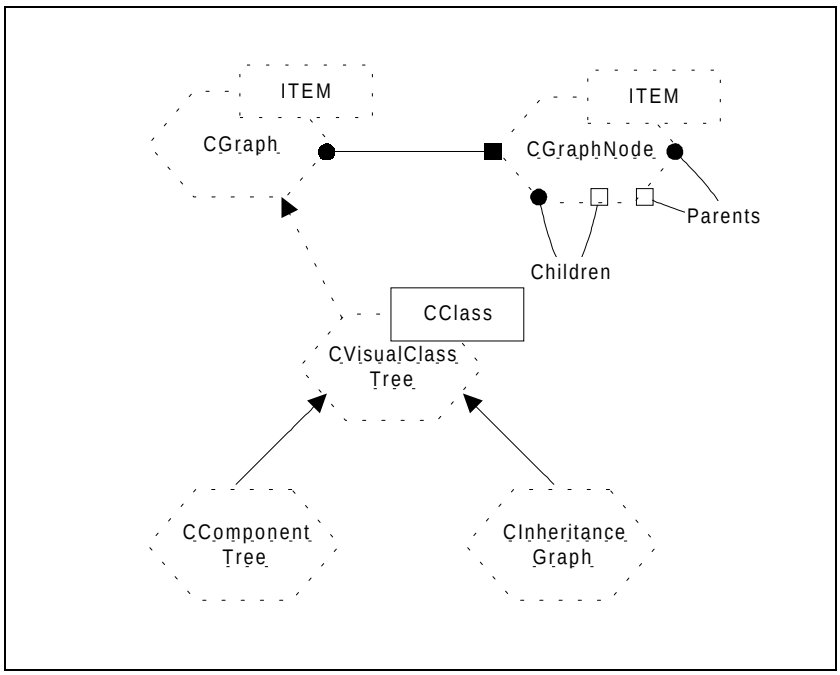
7.4.5 Validierungen

Nachdem die Elemente aus der Hardydatei vollständig gelesen worden sind, können bereits weiterreichende Validierungen der Diagramminformationen durchgeführt werden.
Es wird insbesondere geprüft, ob

- mindestens ein Einstiegspunkt im Diagramm enthalten ist.
- gleichviele Instantiierungspfeile wie Einstiegspunkte im Diagramm enthalten sind.

Die Prüfung der zweiten Bedingung stellt zudem sicher, dass auch mindestens eine Datenbankklasse definiert worden ist, da ein Instantiierungspfeil nur dann gezeichnet werden kann, wenn er auf eine derartige Klasse zuläuft.
Weitere Validierungen (Eindeutigkeit, Zyklenfreiheit, etc.) werden laufend während der nachfolgenden Verarbeitung vorgenommen und müssen demnach nicht speziell in diesem Abschnitt behandelt werden.

7.4.6 Die Graphklassen



Klassendiagramm 7-5: Klassen für den Aufbau der Graphen

Klassen von OSWOOD

`CGraph`

Klassenvorlage für die Modellierung eines Graphen mit beliebigen Knotenelementen.

`CGraphNode`

Klassenvorlage für einen Knoten in einem Graphen.

`CVisualClassTree`

Instatiierung eines Graphen mit Knoten vom Typ `CClass`, erweitert um verschiedene Visualisierungsmethoden zur Überführung in ein `CTreeCtrl`-Objekt (MFC).

`CInheritanceGraph`

Vererbungsgraph.

`CComponentTree`

Komponentenbaum.

7.4.6.1 CGraph

CGraph<ITEMTYPE>

`HITEM AddNode(ITEMTYPE* pClass, HITEM hGraphItem =GRAPHROOT)`

Fügt einen neuen Knoten im Graphen als Kinde des Knotens `hGraphItem` ein. Wird kein Vater angegeben, wird das Element als Wurzel hinzugefügt

`void SetParent(HITEM hChild, HITEM hParent) const`

Setzt den Vater des Elementes `hChild` auf den Wert `hParent`

`HGRAPHITEM GetFirstRoot(void) const`

Liefert das erste Wurzelobjekt des Graphen (Element ohne Vater)

`HGRAPHITEM GetNextRoot(HGRAPHITEM hRoot) const`

Liefert das nächstfolgende Element in der Liste der Wurzelobjekte. Mit der Iteration über die Methoden `GetFirstRoot()` und `GetNextRoot()` lassen sich alle Wurzelobjekt auflisten

`HGRAPHITEM Find(const ITEMTYPE* pClass) const`

Liefert für ein gegebenes Objekt den Zeiger auf das entsprechende

Knotenelement

`ITEMTYPE* GetClass(HGRAPHITEM hGraphItem) const`
Liefert für ein gegebenes Knotenelement den entsprechenden Zeiger auf das darin gespeicherte Objekt

`CPtrArray m_apNode`
Sammlung aller Knoten

Klassenbeschreibung 7-9: CGraph

```
#define GRAPHITEM CGraphNode<ITEMTYPE>
#define HGRAPHITEM CGraphNode<ITEMTYPE>*
#define GRAPHROOT NULL

template<class ITEMTYPE>
class CGraph {
                    public:
                         ~CGraph();
                    public:
                         HITEM AddNode(ITEMTYPE* pClass,HITEM
hGraphItem=GRAPHROOT);
                         void SetParent(HITEM hChild,HITEM
hParent) const;
                         HGRAPHITEM GetFirstRoot(void) const;
                         HGRAPHITEM GetNextRoot(HGRAPHITEM
hRoot) const;
                         HGRAPHITEM GetFirstLeaf(void) const;
                         HGRAPHITEM GetNextLeaf(HGRAPHITEM
hLeaf) const;
                         HGRAPHITEM Find(const ITEMTYPE*
pClass) const;
                         inline const ITEMTYPE*
GetClass(HGRAPHITEM hGraphItem) const;
                    private:
                         inline void SetParent(HGRAPHITEM
hChild,HGRAPHITEM hParent) const;
public:
                    CPtrArray m_apNode;
}; //end class CGraph
```

Listing 7-21: Klassendefinition von CGraph

7.4.6.2 CGraphNode

CGraphNode<ITEMTYPE>

`void SetParent(const HGRAPHNODE)`
Fügt den gegebenen Knoten zur Sammlung der Väter hinzu

`void SetChild(const HGRAPHNODE)`
Fügt den gegebenen Knoten zur Sammlung der Kinder hinzu

`inline BOOL IsRoot(void) const`
Gibt an, ob der Knoten eine Wurzel ist (Menge der Väter leer)

`inline BOOL HasChilds(void) const`
Gibt an, ob der Knoten Kinder hat (Menge der Kinder nicht leer)

`HGRAPHNODE GetFirstChild(void) const`
Liefert den ersten Knoten aus der Sammlung aller Kinder

`HGRAPHNODE GetNextChild(const HGRAPHNODE hChild) const`
Liefert den nächsten Knoten aus der Sammlung der Kinder. Mit den Funktionen `GetFirstChild()` und `GetNextChild()` lässt sich einfach über die Menge der Kinder iterieren.

`ITEMTYPE* GetItem(void) const`
Liefert das mit dem Knoten verbundene Graphelement

`CPtrArray m_apParentNode`
Sammlung der Zeiger auf die Väter

`CPtrArray m_apChildNode`
Sammlung der Zeiger auf die Kinder

`ITEMTYPE* m_pItem`
Zeiger auf das durch den Knoten repräsentierte Objekt

Klassenbeschreibung 7-10: CGraphNode

```
#define HGRAPHNODE CGraphNode<ITEMTYPE>*

template<class ITEMTYPE>
class CGraphNode {
                  public:
                        CGraphNode(ITEMTYPE*);
                        ~CGraphNode(void);
                  public:
                        void SetParent(const HGRAPHNODE);
                        void SetChild(const HGRAPHNODE);
                        inline BOOL IsRoot(void) const;
                        inline BOOL HasChilds(void) const;
                        HGRAPHNODE GetFirstChild(void) const;
                        HGRAPHNODE GetNextChild(const
HGRAPHNODE hChild) const;
                        inline BOOL IsLeaf(void) const;
                        inline BOOL HasParents(void) const;
                        HGRAPHNODE GetFirstParent(void)
const;
                        HGRAPHNODE GetNextParent(const
HGRAPHNODE hParent) const;
                        ITEMTYPE* GetItem(void) const;
private:
                  CPtrArray m_apParentNode;
                  CPtrArray m_apChildNode;
                  ITEMTYPE* m_pItem;
}; //end class CGraphNode
```

Listing 7-22: Klassendefinition von CGraphNode

7.4.6.3 CVisualClassTree

Die Klasse `CVisualClassTree` instantiiert die vorgängig erläuterte Klassenvorlage `CGraph` mit dem Typen `CClass`

```
class CVisualClassTree :  public
CGraph<CClass>
```

Für die Anzeige eines Graphen bietet sie eine Funktion `PrintTree()` zur Verwendung an. Diese setzt den Graphen in einen Baum um und muss deshalb davon ausgehen können, dass sich im Graphen keine Zyklen befinden. Die Prüfung dieser Voraussetzung bleibt der Implementation des Aufbaualgorithmus' vorenthalten. Die Umsetzung sei nachfolgend beschrieben:

```
void CVisualClassTree::PrintTree(CTreeCtrl& TreeCtrl,
HTREEITEM hRoot) const {
                  const CClass* pClass;

                  ...

(1)               for (CGraphNode<CClass>*
hCurrentRoot=GetFirstRoot();
(7)                                     hCurrentRoot!=NULL;
(6)
                  hCurrentRoot=GetNextRoot(hCurrentRoot)) {

(2)                     pClass=GetClass(hCurrentRoot);

(3)                     HTREEITEM
hTreeItem=TreeCtrl.InsertItem(...);

(4)                     if (hCurrentRoot->HasChilds())
(5)
                  PrintChilds(TreeCtrl,hCurrentRoot,hTreeIte
m);

                  }//endfor
}//end PrintTree
```

Listing 7-23: Aufbau eines visuellen Baumes

(1) Erster Wurzelknoten beschaffen.
(2) Objektzeiger beschaffen.
(3) Objekt in den visuellen Baum einfügen.
(4) Prüfen, ob das Objekt Kinder besitzt.
(5) Wenn ja, diese ebenfalls in den Baum einfügen.
(6) Nächstes Wurzelobjekt beschaffen,
(7) solange noch eines vorhanden ist.

Die Umsetzung der Funktion zum Einfügen der Kinder sei nachfolgend beschrieben:

```
void CVisualClassTree::PrintChilds(CTreeCtrl& TreeCtrl,
const CGraphNode<CClass>* hParent,
HTREEITEM hTreeParent) const {
                const CClass* pClass;

                ...

(1)             for (CGraphNode<CClass>*
hCurrentChild=hParent->GetFirstChild();
(7)                          hCurrentChild!=NULL;
(6)                          hCurrentChild=hParent-
>GetNextChild(hCurrentChild)) {

(2)                  pClass=GetClass(hCurrentChild);

(3)                  HTREEITEM
hTreeItem=TreeCtrl.InsertItem(...);

(4)                  if (hCurrentChild->HasChilds())
(5)
                PrintChilds(TreeCtrl,hCurrentChild,hTreeIt
em);
                }//endfor
}//end PrintChilds
```

Listing 7-24: Klassendefinition von CGraph

(1) Erster Kindknoten beschaffen.
(2) Objektzeiger beschaffen.
(3) Objekt in den visuellen Baum einfügen.
(4) Prüfen, ob das Objekt weitere Kinder besitzt.
(5) Wenn ja, Kindeskinder ebenfalls in den Baum einfügen.
(6) Nächstes Kindobjekt beschaffen,
(7) solange noch eines vorhanden ist.

7.4.7 Aufbau des Vererbungsgraphen

Da im betrachteten Datenbanksystem Mehrfachvererbung zugelassen ist, muss die Vererbungshierarchie als Graph modelliert werden. Der Graph besteht dabei aus beliebig vielen Baumelementen des Typs Datenbankklasse, abstrakte Klasse und Hilfsklasse. Da während des Aufbaus aber auch während des Auslesens der genaue Typ des Elementes nicht von Bedeutung ist, werden die Knoten des Graphen mit den Instanzen der Basisklasse `CClass` verbunden.
Für den Aufbau der Vererbungshierarchie müssen also die Liste der Vererbungspfeile sowie die Liste der Datenbankklassen, der Hilfsklassen und der abstrakten Klassen berücksichtigt werden.
Für den Aufbau des Graphen wird das Vorgehen gewählt, welches zuerst aus der Menge der Klassen die Wurzelelement herauszieht und anschliessend von ihnen ausgehend die Subklassen bestimmt. Wurzelobjekte werden im Folgenden auch „Urväter“ genannt, also Klassen, die bezüglich der Vererbung keine weiteren Väter mehr besitzen.

Die Suche nach den Urvätern muss beziehungsüberdeckend sein, weil ansonsten Zyklen in den Beziehungen unentdeckt bleiben könnten.
Das generelle Vorgehen kann in die Schritte „Suche nach dem Urvater", „rekursives Füllen des Graphen" und „Einfügen der verbleibenden Klassen" unterteilt werden. Dabei kann der erste Abschnitt etwa wie folgt umrissen werden:
1. Aus der Liste der Vererbungsbeziehungen wird ein noch nicht besuchtes Element genommen und diejenige Klasse aus den Klassenlisten gesucht, die als Vater dieser Beziehung zu interpretieren ist (Ende der Kante).
Die Kindklasse, die Vaterklasse sowie die Beziehung werden als „besucht" gekennzeichnet
2. Aus der Liste der Vererbungsbeziehungen wird ein Element gesucht, welches die gefundene Klasse selber als Kindklasse ausweist (Beginn der Kante).
3. Wird ein solches Element gefunden, wird der Schritt 2 für diejenige Klasse wiederholt, die als Vater dieser Beziehung zu interpretieren ist (Ende der Kante). Die Klasse sowie die Beziehung wird als „besucht" gekennzeichnet.
Wird kein solches Element gefunden (oder sind alle Beziehungen schon besucht), ist der Urvater gefunden. Der Urvater wird in die Liste der Wurzeln der Vererbungshierarchie aufgenommen. Die Klassen werden allesamt auf „unbesucht" zurückgesetzt.
Wird in Schritt 2 eine Klasse als Vater gefunden, die bereits besucht worden ist, ist der Graph der Vererbungsbeziehungen nicht zyklenfrei. Eine entsprechende Fehlerbedingung muss ausgelöst werden.
Im Folgenden soll die Implementation des Algorithmus' abgebildet und mit verschiedenen Kommentaren erläutert werden.
Vorbereitungen:

```
void CSchemaDoc::CreateInheritanceHierarchy(void) {
                        int i;

                        // create and initialize visitor array
(1)                     CArray<BOOL,BOOL&> abArcVisited;
                        abArcVisited.SetSize(m_aInheritanceArc.Get
Size());
                        for
(i=0;i<abArcVisited.GetSize();abArcVisited[i++]=FALSE);

                        // create and initialize array of used
arcs
(2)                     CArray<BOOL,BOOL&> abArcUsed;
                        abArcUsed.SetSize(m_aInheritanceArc.GetSiz
e());
                        for
(i=0;i<abArcUsed.GetSize();abArcUsed[i++]=FALSE);

                        // create list of adams
(3)                     CArray<int,int&> anArcToAdam;
                        CArray<int,int&> anAdamId;
```

Listing 7-25: Funktion CreateInheritanceHierarchy (Vorbereitungen)

(1) Aufbau eines Vektors, welcher die bereits besuchten Vererbungsbeziehungen speichert. Alle Kanten werden auf den Status 'nicht besucht' initialisiert.

(2) Aufbau eines Vektors, welcher die bereits für eine Vererbung verwendeten Vererbungsbeziehungen speichert. Alle Kanten werden auf den Status 'nicht benutzt' initialisiert.

(3) Aufbau eines Vektors, welcher die Vererbungsbeziehungen, die auf einen Urvater zeigen, speichert.
(4) Aufbau eines Vektors, welcher die Identifikatoren der Urväter speichert.

Suche nach den Urvätern

```
                        // do while there are unvisited
inheritance relations
                        int nCurrentArc;
(1)                     while (TRUE) {

                              // get next not yet visited
inheritance relation
(2)                           for (nCurrentArc=0;
nCurrentArc<m_aInheritanceArc.GetSize();
nCurrentArc++) {
                                    if
(!abArcVisited[nCurrentArc]) break;
                              }//endfor
(3)                           if
(nCurrentArc==m_aInheritanceArc.GetSize()) break;
(4)                           abArcVisited[nCurrentArc]=TRUE;

                              // search adam
(5)                           while (TRUE) {
                                    // search parent class for
current class
(6)                                 for (int
j=0;j<m_aInheritanceArc.GetSize();j++) {
                                          if
(m_aInheritanceArc[j].m_nFromNode ==
m_aInheritanceArc[nCurrentArc].m_nToNode)
                                                break;
                                    }//endfor

                                    // if there is no parent, an
adam candidate is found
(7)                                 if (j ==
m_aInheritanceArc.GetSize()) {
                                          // check if this is an
new one
(8)                                       for
(i=0;i<anArcToAdam.GetSize();i++) {
                                                if
(m_aInheritanceArc[anArcToAdam[i]].m_nToNode ==
m_aInheritanceArc[nCurrentArc].m_nToNode)
                                                      break;
                                          }//endif
                                          if
(i==anArcToAdam.GetSize())
(9)
                        anArcToAdam.Add(nCurrentArc);
                                                break;
                                    }//endif

                                    // if there is an arc found
which was already used to find an adam
(10)                          if (abArcUsed[j]) break;

                                    // if there is an arc found
which was already visited, there is a cycle
(11)                          if (abArcVisited[j])
                                          throw CFatalError
("Class daigram contains a cylce in the inheritance");

                                    // the current is not the adam
so continue search with his parent
(12)                          nCurrentArc=j;

                                    // set relation as visited
(13)                          abArcVisited[nCurrentArc]=TRUE;

                              }//endwhile

                              // set the visited arcs as used
(14)                    for (i=0;i<abArcUsed.GetSize();i++) {

                        abArcUsed[i]=(abArcUsed[i]||abArcVisited[i
]);
                              }//endfor
```

```
                    }//endwhile
```
Listing 7-26: Funktion CreateInheritanceHierarchy (Suche nach den Urvätern)

(1) Äussere Schleife wird solange durchlaufen, wie noch nicht besuchte Beziehungen existieren.
(2) Über die Menge aller Beziehungen wird iteriert, bis eine noch nicht besuchte Kante gefunden werden konnte. Diese wird zur aktuellen Kante.
(3) Falls keine solche Kante mehr gefunden werden konnte, wird die äussere Schleife verlassen.
(4) Die aktuelle Kante wird als 'besucht' markiert.
(5) Die innere Schleife wird durchlaufen, bis eine der untenstehenden Abbruchbedingungen erfüllt ist.
(6) Aus der Menge der Beziehungen wird eine Kante gesucht, welche die durch die aktuelle Kante referenzierte Klasse als Ausgangsknoten besitzt.
(7) Konnte keine derartige Kante gefunden werden, könnte es sich bei der durch die aktuelle Kante referenzierte Klasse um einen Urvater handeln.
(8) Der Kandidat könnte aber bereits in der Menge der Urväter enthalten sein, weshalb in dieser nach dem Kandidaten gesucht werden muss.
(9) Handelt es sich tatsächlich um einen neuen Kandidaten, wird er in die Liste der Urväter aufgenommen.
(10) Wurde die aktuelle Kante bereits für das Auffinden eines Urvaters verwendet, bringt diese Kante keine neuen Erkenntnisse mehr, weshalb aus der inneren Schleifen herausgesprungen wird.
(11) Wurde die aktuelle Kante bereits einmal besucht, so hat man es mit einem Zyklus in der Vererbung zu tun und muss die Verarbeitung mit der Auslösung einer Ausnahme beenden.
(12) Handelt es sich bei der aktuellen Kante nicht um eine, auf einen Urvater zeigende, so wird mit dessen 'Nachfolgekante' als aktuelle Kante weitergesucht.
(13) Die neue Kante wird ebenfalls als 'besucht' gekennzeichnet.
(14) Konnte ein Urvater gefunden werden, so müssen die für dessen Auffindung benötigten Kanten als 'benutzt' markiert werden.

Auffüllen der Graphen mit den Kindern der Urväter

```
                    // fill the inheritance graph recursively
(1)                 CArray<int,int&> anAddedClassId;
                    CClass* pClass;
                    int nAdamId;
                    HITEM hRoot;
(2)                 for (i=0;i<anArcToAdam.GetSize();i++) {

                    nAdamId=m_aInheritanceArc[anArcToAdam[i]].
m_nToNode;
                         pClass=GetClass(nAdamId);
                         ASSERT(pClass);
(3)
                    hRoot=m_InheritanceGraph.AddNode(pClass);
                         anAddedClassId.Add(nAdamId);
(4)
                    AddDerivedClasses(nAdamId,hRoot,anAddedCla
ssId);
                    }//endif
```
Listing 7-27: Funktion CreateInheritanceHierarchy (rekursives Füllen des Graphen)

(1) Aufbau eines Vektors, welcher die bereits in den Graphen aufgenommenen Klassenidentifikatoren speichert. (Wird für das Einfügen der restlichen Klassen verwendet.)

(2) Jeder gefundene Urvater
(3) wird in den Vererbungsgraphen eingesetzt.
(4) Dasselbe passiert auch mit dessen Kindern.

Hinzufügen der Kinder eines Urvaters:

```
void CSchemaDoc::AddDerivedClasses(int nParentId,HITEM hParent,
CArray<int,int&>& anAddedClassId) {
                        CClass* pClass;
                        HITEM hChild;
                        int nChildId;

                        // search in the list of arcs for all
nodes having an arc to current parent
(1)                     for (int
i=0;i<m_aInheritanceArc.GetSize();i++) {

                             // for each such node found get his
class, add the node to inheritance graph
(2)                          if
(m_aInheritanceArc[i].m_nToNode==nParentId) {

                        nChildId=m_aInheritanceArc[i].m_nFromNode;
                                  pClass=GetClass(nChildId);
                                  ASSERT(pClass);
(3)
                        hChild=m_InheritanceGraph.AddNode(pClass,h
Parent);
                                  anAddedClassId.Add(nChildId);

                                  // and look for his children
(4)
                        AddDerivedClasses(nChildId,hChild,anAddedC
lassId);
                             }//endif
                        }//endfor
}//end AddDerivedClasses
```

Listing 7-28: Funktion AddDerivedClasses (rekursives Füllen des Graphen)

(1) Suche in der Menge der Vererbungsbeziehungen alle Beziehungen,
(2) welche auf den Vater zeigen.
(3) Füge diese Klasse
(4) und dessen Kinder dem Vererbungsgraphen hinzu.

Hinzufügen der verbleibenden Klassen:

```
                        // add the non used database classes as
roots
(1)                     for (i=0;i<m_aDatabaseClass.GetSize();i++)
{
                             if
(!IsElementOf(anAddedClassId,m_aDatabaseClass[i].m_nId)){
                                  m_InheritanceGraph.AddNode(
(CClass*)&m_aDatabaseClass[i]);
                             }//endif
                        }//endif

                        // add the non used abstract classes as
roots
(2)                     for (i=0;i<m_aAbstractClass.GetSize();i++)
{
                             if
(!IsElementOf(anAddedClassId,m_aAbstractClass[i].m_nId)){
                                  m_InheritanceGraph.AddNode(
(CClass*)&m_aAbstractClass[i]);
                             }//endif
                        }//endif

                        // add the non used class utilities as
roots
(3)                     for (i=0;i<m_aClassUtility.GetSize();i++)
{
                             if
(!IsElementOf(anAddedClassId,m_aClassUtility[i].m_nId)) {
                                  m_InheritanceGraph.AddNode(
(CClass*)&m_aClassUtility[i]);
                             }//endif
                        }//endif
}//end CreateInheritanceHierarchy
```

Listing 7-29: Funktion CreateInheritanceHierarchy (Hinzufügen der verbleibenden Klassen)

(1) Suche in der Menge aller Datenbankklassen,
(2) aller abstrakter Klassen und
(3) aller Hilfsklassen nach noch nicht eingefügten Elementen und definiere diese als Wurzelknoten.

7.4.8 Aufbau des Komponentenbaumes

Von der Methode DEIMOS ist gefordert worden, dass alle Datenbankklassen als direkte oder indirekte Komponenten eines Einstiegspunktes zu definieren sind. Diese Forderung lässt sich ebenfalls mit einem Graphen modellieren. Der Graph besteht dabei aus einem oder mehreren disjunkten Komponentenbäumen und jeder dieser Bäume hat einen Einstiegspunkt als Wurzelknoten. In Baumelemente können Datenbankklassen oder abstrakte Klassen enthalten sein. Hilfsklassen dürfen aufgrund ihrer Transienz nicht als Komponenten modelliert sein und dürfen deshalb nicht in den Baum aufgenommen werden.
Wiederum ist während des Aufbaus und während des Auslesens der genaue Typ des Elementes nicht von Bedeutung, weshalb die Knoten des Graphen mit den Instanzen der Basisklasse `CClass` verbunden werden.
Für den Aufbau der Vererbungshierarchie muss also die Liste der Einstiegspunkte, deren Instatiierungskanten, die Liste der Komponentenpfeile sowie die Liste der Datenbank- und der abstrakten Klassen berücksichtigt werden.
Das gewählte Vorgehen nimmt zuerst alle durch einen Einstiegspunkt instantiierten Klassen auf und füllt anschliessend von ihnen ausgehend rekursiv den Baum mit den entlang der Komponentenbeziehungen referenzierten Elementen.
Es muss sichergestellt sein, dass jede Datenbankklasse genau einmal in einem derartigen Baum auftritt und dass jede Komponentenbeziehung angetroffen wurde (keine Beziehungen zu anderen Klassentypen).
Im Folgenden soll die Implementation des Algorithmus' abgebildet und mit verschiedenen Kommentaren erläutert werden.

Suche die Wurzeln der Komponentenbäume:

```
void CSchemaDoc::CreateComponentTree(void) {
                        int i;

                        // find the root(s) of the component tree
                        // set these classes to state visited
                        CArray<int,int&> anRootId;
                        CArray<int,int&> anVisitedClass;
                        int nRootCandidate;
(1)                     for
(i=0;i<m_aInstantiationArc.GetSize();i++) {
(2)
                        nRootCandidate=m_aInstantiationArc[i].m_nT
oNode;
(3)                          if
(!IsElementOf(anRootId,nRootCandidate)) {
(4)                                anRootId.Add(nRootCandidate);

                        anVisitedClass.Add(nRootCandidate);
                             }//endif
                        }//endfor
```

Listing 7-30: Funktion CreateComponentTree (Suche nach den Wurzelobjekten)

(1) Für alle Instantiierungspfeile
(2) wird die zugehörige Klasse gefunden.
(3) Falls diese noch nicht in der Menge der Wurzeln enthalten ist,

(4) wird er in diese Menge aufgenommen.

Rekurives Füllen der Komponentenbäume:

```
                        // fill the component tree recursively
                        CClass* pClass;
                        HITEM hRoot;
(1)                     for (i=0;i<anRootId.GetSize();i++) {
(2)                          pClass=GetClass(anRootId[i]);
                             ASSERT(pClass);
(3)
                        hRoot=m_ComponentTree.AddNode(pClass);
(4)
                        AddComponents(pClass,anVisitedClass,anRoot
Id[i],hRoot);
                        }//endfor
```

Listing 7-31: Funktion CreateComponentTree (Rekursives Füllen)

(1) Für alle Wurzeln

(2) wird die Klasse beschafft,

(3) diese

(4) und deren Komponenten werden in den Komponentenbaum eingefügt.

Hinzufügen von Komponenten:

```
void CSchemaDoc::AddComponents(CClass* pParentClass,
CArray<int,int&>& anVisitedClass,
int nParentId, HITEM hParent) {
                        CClass* pClass;
                        HITEM hChild;
                        int nChildId;

                        // search in the list of arcs for all
nodes having an arc from current parent
(1)                     for (int
i=0;i<m_aComponentArc.GetSize();i++) {

                             // for each such node found get his
class, add the node to component tree
(2)                          if
(m_aComponentArc[i].m_nFromNode==nParentId) {

                        nChildId=m_aComponentArc[i].m_nToNode;

                                  // get nodes class
(3)                               pClass=GetClass(nChildId);
                                  ASSERT(pClass);

                                  // look for duplicate usage of
class
(4)                               if
(IsElementOf(anVisitedClass,nChildId))
                                       throw
CFatalError(„defined as a component twice");

                                  anVisitedClass.Add(nChildId);

                                  // add the class to the
component tree if it is not an abstract class
(5)                               if (pClass-
>m_nType==CItem::typeAbstractClass)
                                       throw
CFatalError(„Abstract classes as components");
(6)
                        hChild=m_ComponentTree.AddNode(pClass,hPar
ent);
(7)                               pParentClass-
>CreateComponentRelation(...);

                                  // and look for his children
(8)
                        AddComponents(pClass,anVisitedClass,nChild
Id,hChild);
                             }//endif
                        }//endfor
}//end AddComponents
```

Listing 7-32: Funktion AddComponents (Rekursives Füllen)

(1) Suche aus der Menge der Komponentenbeziehungen

(2) eine, die von der aktuellen Klasse ausläuft.

(3) Beschaffe die referenzierte Klasse.
(4) Falls diese schon im Baum enthalten ist, liegt eine Verletzung der Eindeutigkeit vor.
(5) Falls es sich um eine abstrakte Klasse handelt, ist eine Forderung von DEIMOS verletzt.
(6) Die Klasse wird dem Komponentenbaum hinzugefügt.
(7) und die entsprechende Komponentenbeziehung aufgebaut.
(8) Falls die eben eingefügte weitere Komponenten besitzt, werden diese gleichsam aufgenommen (Rekursion).

Hinzufügen der verbleibenden Klassen:

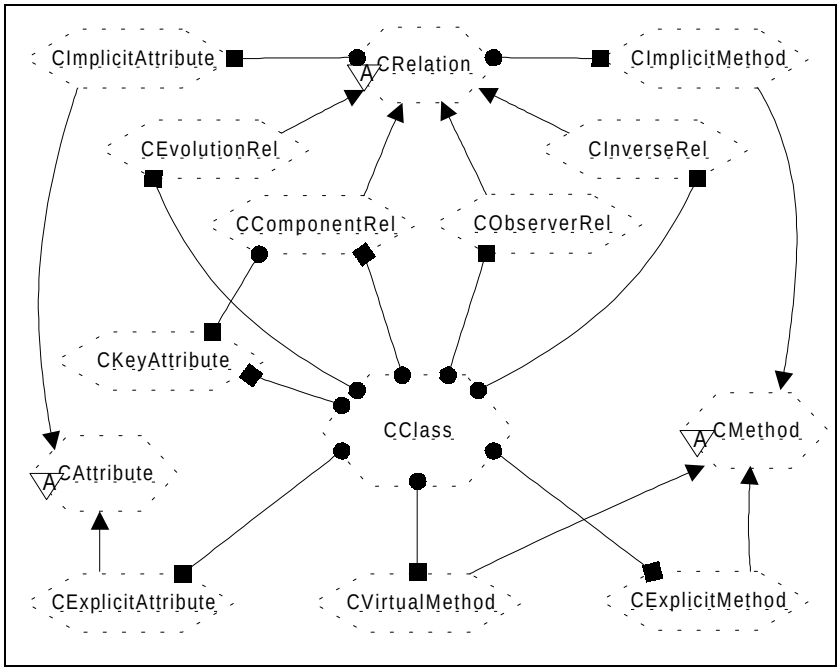
```
                        //for each database class which is not
used yet
(1)                     for (i=0;i<m_aDatabaseClass.GetSize();i++)
{
                             if
(!IsElementOf(anVisitedClass,m_aDatabaseClass[i].m_nId)){
(2)                                  throw CFatalError("Class not
defined as a component");
                             }//endif
                        }//endfor

                        // add the abstract classes as roots
(3)                     for (i=0;i<m_aAbstractClass.GetSize();i++)
{

                        m_ComponentTree.AddNode((CClass*)&m_aAbstr
actClass[i]);
                        }//endif

                        // add the class utilities as roots
(4)                     for (i=0;i<m_aClassUtility.GetSize();i++)
{

                        m_ComponentTree.AddNode((CClass*)&m_aClass
Utility[i]);
                        }//endif

}//end CreateComponentTree
```

Listing 7-33: Funktion CreateComponentTree (restliche Klassen)

(1) Die Menge der Datenbankklassen wird nach nicht besuchten Einträgen durchforscht.
(2) Werden derartige Elemente gefunden, liegt eine Verletzung der zwingenden Bedingung von DEIMOS vor
(3) Sämtliche abstrakten Klassen werden als Wurzelelemente hinzugefügt.
(4) Sämtliche Hilfsklassen werden als Wurzelelemente hinzugefügt.

7.4.9 Umsetzen der Kanten in Beziehungen

Um sämtliche Eigenschaften einer Klasse modellieren zu können, werden die nachfolgend im Diagamm dargestellten Klassen benötigt.



Klassendiagramm 7-6: Klassen für die Modellierung einer 'Klasse'

Klassen von OSWOOD

`CClass`

Modellierung einer Klasse im Sinne des Datenbankentwurfes. Basisklasse für die Klassen `CDatabaseClass`, `CAbstractClass` und `CClassUtility`.

`CRelation`

Abstrakte Basisklasse aller Relationen. Enthält insbesondere implicite Attribute und Methode für dessen Umsetzung.

`CComponentRelation`

Komponentenbeziehung.

`CInverseRelation`

Inverse Beziehung.

`CObserverRelation`

Beobachtungsbeziehung.

`CEvolutionRelation`

Evolutionsbeziehung.

`CAttribute`

Abstrakte Klasse aller Attribute.

`CExplicitAttribute`

Benutzerdefinierten Attribute (Klasseneigenschaften).

`CImplicitAttribute`

Von Relationen implizit generierte Attribute.

`CMethod`

Abstrakte Klasse aller Methoden.

`CExplicitMethod`

Benutzerdefinierte Methoden (Klasseneigenschaften).

`CImplicitMethod`

Von Relationen implizit generierte Methoden

`CVirtualMethod`

Von OSWOOD implizit generierte Methoden, welche durch die Vererbung von der rein virtuellen Klasse `DBObject` stammen.

`CKeyAttribute`

Benutzerdefinierte Schlüsselattribute (Klasseneigenschaften) und von der Komponentenrelation implizit generierte Attribute.

Im Folgenden sollen die Klassen `CClass` und `CRelation` genauer betrachtet werden:

7.4.9.1 CClass

CClass : public CItem

`void CreateComponentRelation(CClass* pClass, CString strCardinality, CString strCardinalityInverse)`
Erzeugt eine Komponentenbeziehung zu der Klasse `pClass` mit den Kardinalitäten `strCardinality` und `strCardinalityInverse`

`void CreateInverseRelation(CClass* pClass, CString strCardinality, CString strCardinalityInverse)`
Erzeugt eine inverse Beziehung zu der Klasse `pClass` mit den Kardinalitäten `strCardinality` und `strCardinalityInverse`

`void CreateObserverRelation(CClass* pClass, CString strCardinality, CString strCardinalityInverse)`
Erzeugt eine Beobachtungsbeziehung zu der Klasse `pClass` mit den Kardinalitäten `strCardinality` und `strCardinalityInverse`

`void CreateEvolutionRelation(CClass* pFromClass, CClass* pToClass,CClass* pParentClass)`
Erzeugt eine Evolutionsbeziehung von der Klasse pFromClass zu der Klasse `pToClass` mit der Klasse `pParentClass` als gemeinsame Vaterklasse

`void CreateVirtualMethods(void)`
Erzeugt die virtuellen Methoden

`void CreateDefinition(CString strParents)`
Erzeugt die Klassendefinition in ODL

`CArray<CVirtualMethod,CvirtualMethod&> m_aConstructor`
Sammlung der virtuellen Methoden, die als Konstruktoren aufgefasst werden dürfen

`CArray<CvirtualMethod,CVirtualMethod&> m_aVirtualMethod`
Sammlung der restlichen virtuellen Methoden

`CArray<CcomponentRelation,CComponentRelation&> m_aComponentRelation`
Sammlung der Komponentenbeziehungen

`CArray<CInverseRelation,CinverseRelation&> m_aInverseRelation`
Sammlung der inversen Beziehungen

`CArray<CObserverRelation,CObserverRelation&> m_aObserverRelation`
Sammlung der Beobachtungsbeziehungen

`CArray<CEvolutionRelation,CEvolutionRelation&> m_aEvolutionRelation`
Sammlung der Evolutionsbeziehungen

`CString m_strDefinition`
Definitionsstring in ODL

Klassenbeschreibung 7-11: CClass

```
class CClass : public CNode {
                    // typdefs
                    public:

                    // construction/destruction
                    public:
                         CClass() {};
                         CClass(const CItem& Item) :
CNode(Item) {};

                    // operations
                    public:
                         CClass& operator=(CClass& Class);

                    // methods
                    public:
                         virtual void Validate(void);

                         void ParsePropertyEntry(void);
                         void ParseMethodProperty(CString
strProperty);
                         void ParseConstraintProperty(CString
strProperty);
                         void
ParseKeyAttributeProperty(CString strProperty);
                         void ParseAttributeProperty(const
CString& strProperty);

                         void CreateComponentRelation(CClass*
pClass,
CString strCardinality,
CString strCardinalityInverse);
                         void CreateInverseRelation(CClass*
pClass,
CString strCardinality,
CString strCardinalityInverse);
                         void CreateObserverRelation(CClass*
pClass,
CString strCardinality,
CString strCardinalityInverse);
                         void CreateEvolutionRelation(CClass*
pFromClass,
CClass* pToClass,CClass* pParentClass);

                         void CreateVirtualMethods(void);
                         void CreateDefinition(CString
strParents);

                         void Show(HTREEITEM
hClassRoot,CTreeCtrl& TreeCtrl,
UINT nMask,CClassView* pClassView);
                    private:
                         CString GetLabel(void);

                    // attributes
                    public:
                         // user defined values
                         CString m_strClassName;
                         CString m_strProperties;

                         // user defined members

                    CArray<CExplicitAttribute,CExplicitAttribu
te&> m_aExplicitAttribute;

                    CArray<CExplicitMethod,CExplicitMethod&>
m_aExplicitMethod;

                         // virtual functions inherited from
DBObject

                    CArray<CVirtualMethod,CVirtualMethod&>
m_aConstructor;

                    CArray<CVirtualMethod,CVirtualMethod&>
m_aVirtualMethod;

                         // relations
```

```
                    CArray<CComponentRelation,CComponentRelati
on&> m_aComponentRelation;

                    CArray<CInverseRelation,CInverseRelation&>
m_aInverseRelation;

                    CArray<CObserverRelation,CObserverRelation
&> m_aObserverRelation;

                         // evolution

                    CArray<CEvolutionRelation,CEvolutionRelati
on&> m_aEvolutionRelation;

                         // key attributes
                         CArray<CKeyAttribute,CKeyAttribute&>
m_aKeyAttribute;

                         // constraints
                         CArray<CString,CString&>
m_astrConstraint;

                         // class definition string in ODL
                         CString m_strParents;
                         CString m_strDefinition;
};  //end class CClass
```

Listing 7-34: Klassendefinition von CClass

7.4.9.2 CRelation

CRelation

`void Create(CClass* pFromClass,CClass* pToClass, const CString& strCardinality, const CString& strCardinalityInverse)`
Erzeugt eine Relation zwischen den Klassen pFromClass und pToClass indem erst die übergebenen Kardinalitätsangaben mit Hilfe der Funktion `GetCardinalityBounds()` in Limiten umgesetzt werden und anschliessend ein implizites Attribut für die Verwaltung der Referenzmenge und je nach Typ der Relation implizite Methoden für die Verwaltung der Beziehung angelegt werden.
`void GetCardinalityBounds(const CString& strCardinality,int& nMinimum,int& nMaximum)`
Ermittelt die im Kardinalitätsstring definierte minimal und maximal erlaubte Anzahl von Beziehungen und liefert diese zurück (vgl. erlaubte Kardinalitätsangaben im Abschnitt "**Fehler! Kein gültiges Resultat für Tabelle.**", Seite 75)

`CImplicitAttribute m_ReferenceAttribute`
Implizites Attribut für die Verwaltung der Referenzmenge
`CArray<CImplicitMethod,CImplicitMethod&> m_aImplicitMethod`
Implizite Methode für die Verwaltung der Beziehung
`Cclass* m_pFromClass`
Zeiger auf die Ausgangsklasse der Beziehung
`Cclass* m_pToClass`
Zeiger auf die Zielklasse der Beziehung

Klassenbeschreibung 7-12: CRelation

```
class CRelation {
                    // construction/destruction
                    public:
                         //CRelation() {};

                    // operations
                    public:
```

```
                    CRelation& operator=(CRelation&
Relation);

                // methods
                public:
                    void Create(CClass*
pFromClass,CClass* pToClass,
const CString& strCardinality,
const CString& strCardinalityInverse);
                    virtual CString GetLabel(void);
                    virtual void Show(HTREEITEM
hRoot,CTreeCtrl& TreeCtrl,UINT nMask,CClassView*
pClassView);
                private:
                    void GetCardinalityBounds(const
CString& strCardinality,int& nMinimum,int& nMaximum);
                    virtual UINT GetImage(void)=0;

                // members
                public:
                    CImplicitAttribute
m_ReferenceAttribute;

                CArray<CImplicitMethod,CImplicitMethod&>
m_aImplicitMethod;
                    CClass* m_pFromClass;
                    CClass* m_pToClass;
};
```

Listing 7-35: Klassendefinition von CRelation

Der Aufbau einer Beziehung läuft für alle Beziehungstypen sehr ähnlich ab, weshalb im Folgenden der Ablauf für inverse Beziehungen stellvertretend für alle weiteren beleuchtet sein soll.

Die Instanz der Klasse CSchemaDoc ruft die Methode `CreateInverseRelations()` auf, welche unten abgebildet ist.

```
void CSchemaDoc::CreateInverseRelations(void) {
                CClass* pFromClass;
                CClass* pToClass;

                ...

(1)             for (int
i=0;i<m_aInverseRelationArc.GetSize();i++) {

                pFromClass=GetClass(m_aInverseRelationArc[
i].m_nFromNode);

                pToClass=GetClass(m_aInverseRelationArc[i]
.m_nToNode);
(2)                 pFromClass-
>CreateInverseRelation(pToClass,
m_aInverseRelationArc[i].m_strCardinalityInverse,
m_aInverseRelationArc[i].m_strCardinality);
(3)                 pToClass-
>CreateInverseRelation(pFromClass,
m_aInverseRelationArc[i].m_strCardinality,
m_aInverseRelationArc[i].m_strCardinalityInverse);
                }//endfor
}//end CreateInverseRelations
```

Listing 7-36: Methode CreateInverseRelations() in CSchemaDoc

(1) Für alle inversen Beziehungen wird ein entsprechendes Objekt für die

(2) Ausgangs- und die

(3) Zielklasse erzeugt.

Die entsprechende Implementierung in der Klasse CClass sieht wie folgt aus:

```
void CClass::CreateInverseRelation(CClass* pToClass,
CString strCardinality,CString strCardinalityInverse) {

(2)             m_aInverseRelation.Add(
(1)                 CInverseRelation(this,pToClass,
strCardinality,strCardinalityInverse));
}//end CreateInverseRelation
```

Listing 7-37: Methode CreateInverseRelation() in CClass

(1) Ein neues Objekt der Klasse `CInverseRelation` wird erzeugt

(2) und der Sammlung aller inversen Beziehungen hizugefügt.

Die Erzeugung einer inversen Beziehung ist nachfolgend dargestellt:

```
CInverseRelation::CInverseRelation(CClass* pFromClass,
CClass* pToClass,const CString& strCardinality,
const CString& strCardinalityInverse) {

(1)
                Create(pFromClass,pToClass,strCardinality,
strCardinalityInverse)

}//end constructor

void CInverseRelation::Create(CClass* pFromClass,CClass*
pToClass,
const CString& strCardinality,
const CString& strCardinalityInverse) {

                // first call base classes create function
(2)             CRelation::Create(pFromClass,pToClass,
strCardinality,strCardinalityInverse);

(3)
                m_aImplicitMethod.Add(CImplicitMethod(this
,pFromClass,pToClass,
CImplicitMethod::typeAttach));
                m_aImplicitMethod.Add(CImplicitMethod(this
,pFromClass,pToClass,
CImplicitMethod::typeDetach));
                m_aImplicitMethod.Add(CImplicitMethod(this
,pFromClass,pToClass,
CImplicitMethod::typeAttachInverse));
                m_aImplicitMethod.Add(CImplicitMethod(this
,pFromClass,pToClass,
CImplicitMethod::typeDetachInverse));

}//end Create
```

Listing 7-38: Methode Create() in CInverseRelation

(1) Der überladene Konstruktor ruft lediglich die `Create()` Methode mit den ihm übergebenen Parametern auf.

(2) Die `Create()`-Methode der Basisklasse wird aufgerufen.

(3) Anschliessend werden die impliziten Methoden für die Verwaltung einer inversen Beziehung und insbesondere deren Implementationsstrings kreiert..

Die Implementierung der `Create()`-Methode in der Basisklasse sieht wie folgt aus:

```
void CRelation::Create(CClass* pFromClass,CClass* pToClass,
const CString& strCardinality,
const CString& strCardinalityInverse) {

                m_pFromClass=pFromClass;
                m_pToClass=pToClass;

                // implicite attribute settings
                int nMinimum;
                int nMaximum;
(1)
                GetCardinalityBounds(strCardinality,nMinim
um,nMaximum);

                // attribute settings
                if (nMaximum==1) {
(2)                   m_ReferenceAttribute.Create("m_ref"+
pToClass->m_strClassName,
pToClass->m_strClassName,"",
nMinimum,nMaximum);
```

```
                    }
                else {
(3)
                m_ReferenceAttribute.Create("m_setref"+
pToClass->m_strClassName,
"unique set("+pToClass->m_strClassName+")","",
nMinimum,nMaximum);
                }//endif
}//end constructor
```

Listing 7-39: Methode CreateInverseRelation() in CClass

(1) Der übergebene Kardinalitätsstring wird analysiert. Die Minimal- und Maximalanzahl wird ermittelt.

(2) Für Beziehungen, welche maximal zu einem Objekt unterhalten werden können, wird ein einstelliges Referenzattribut erzeugt,

(3) während im anderen Fall ein mengenwertiges Referenzattribut kreiert wird.

7.4.10 Erzeugung der Klassendefinitionen und Methodenimplementierungen

Sind nun alle Kanten in Beziehungen umgesetzt, kann die Erzeugung der Klassendefinition für jeden Klasse erfolgen. Dazu wird erst die Zeile

`class` *`ClassName`* [**`inherit`** *`ClassName`* [`(,` *`ClassName`*`)*]]

erzeugt und anschliessend die

- expliziten benutzerdefinierten Attribute,
- impliziten aus den Beziehungen stammenden Referenzattribute

eingefüllt. Danach müssen die Definitionen der

- Konstruktoren,
- Konsistenzprüfungsmethoden,
- expliziten benutzerdefinierten Methoden und
- impliziten aus den Beziehungen stammenden Verwaltungsmethoden

eingetragen werden. (Beispiel einer derartigen Klassendefinition vgl. Listing 7-42.)

Die Implementierungen der impliziten Methoden wurden bereits bei der Erzeugung der entsprechenden Beziehungen kreiert, weshalb hier nur noch der Aufbau der

- expliziten Methoden und der
- virtuellen Methoden

erledigt werden muss.

7.4.11 Schreiben der ODL-Datei

Damit die erzeugte ODL-Datei in der späteren Verarbeitung immer als eine von OSWOOD generierte Datei erkannt werden kann, wird dieser zu Beginn der folgende Kopf vorangestellt:

```
/*////////////////////////////////////////////////////////
  Schema.o2c - schema definition file for O2 environment
  has been generated using design tool OSWOOD, Version 1.0


  OSWOOD has been deveolped for:
  ------------------------------------------------------
  |                                                    |
  |       Erweiterung objektorientierter Methoden      |
  |       für den konzeptuellen Datenbankentwurf       |
  |                                                    |
  |                     Diplomarbeit                   |
  |                                                    |
```

```
|  der Philosophisch-naturwissenschaftlichen Fakultät  |
|                 der Universität Bern                 |
|                                                      |
|          vorgelegt von Benno Burkhardt 1997          |
|                                                      |
--------------------------------------------------------

  Special Thanks:
        Prof. Dr. Oscar Nierstrasz, University of Berne
        Prof. Dr. Klaus Dittrich, University of Zurich
        Dr. Andreas Gepperd, Unversity of Zurich
        Andreas Behm, University of Zurich
        Dr. Thomas Wüst, CSS Assurance Coorp., Switzerland
        TeTrade AG, Switzerland

//////////////////////////////////////////////////////////*/
```

Listing 7-40: Kopf einer ODL-Datei

Als ersten Eintrag wird der Datei die Definition der Einstiegspunkte einbeschrieben

```
/*//////////////////////////////////////////////////////*/
/* persistent entry point definition(s) */

name Macrohard : Company;
```

Listing 7-41: Definition der Einstiegspunkte in der ODL-Datei

Anschliessend werden der Datei für jeder Klasse die bereits erstellten Definitionen und Implementationen angehängt. Dabei wird die Reihenfolge „Klassendefinition“ - „Methodenimplementation“ stets eingehalten.

```
/*//////////////////////////////////////////////////////*/
/* Company database class */

/* Company database class definition */
class Company inherit DBObject
  type tuple (
    public Name : string,

    ...

    private m_refManager : Manager)
  method
    public Create(...) : boolean,

    ...

    private AssignManagerToDeveloper(...)
end;

/* Company database class implementation */
method body Create in class Company {

    ...

};/* end Create */

...
```

Listing 7-42: Klassendefinition und -implementation in der ODL-Datei

7.5 Umsetzen eines Transaktionsdiagramms

In der Transaktionsdiagrammdatei ist der konzeptuelle Entwurf der Applikationen, Programme, Funktionen und Transaktionen enthalten. OSWOOD soll diese Datei interpretieren, um dem Benutzer auf der einen Seite die Aufrufhierarchie und auf der anderen Seite die Umsetzung in der Schemadefinitionssprache O_2C anzeigen zu können.

7.5.1 Die Transaktionsdiagrammdatei als Input

Eine Schemadiagrammdatei liegt als Textdatei vor und folgt dem untenstehenden Syntax:

`index ::= (item)+`

```
item            ::= ident( property
(,property)* ).
property        ::= variable = value
ident           ::= diagram | card | node |
arc |
node_image | arc_image
variable        ::= type | id | card | arcs |
name | 'name (params) return' |
connections | calls |
inverse |'pseudo code' | variable
```

Syntaxdiagramm 7-4: Syntax einer Schemadiagrammdatei

7.5.1.1 Elemente einer Schemadiagrammdatei

`diagram`
enthält Informationen über die Diagrammdatei im Allgemeinen (wird von OSWOOD nicht verwendet).

`card`
enthält die Information über die Diagrammkarte und bildet so die Verbindung zur Indexdatei. Insbesondere enthält diese Sektion den Typ Diagrammdatei. Im Gegensatz zum Schemadiagramm kann ein Transaktionsdiagramm aus mehreren Karten bestehen, weshalb auch mehrere derartige Einträge, die über die Variable `title` unterschieden werden müssen, in der Hardydatei enthalten sein können.

`node`
enthält die Informationen über einen Knoten im Diagramm. Neben seinem Typ und den vom Benutzer zugewiesenen Attributen werden auch dessen Verbindungen zu anderen Knoten definiert. Eine Referenz auf das Element `node_image` legt seine Erscheinungsform und seinen -ort fest.

`arc`
beschreibt einen Verbinder zwischen zwei Knoten. Dieser ist gleichsam typisiert und enthält Verweise auf die Knoten, die durch ihn verbunden werden.

`node_image`
definiert das Aussehen eines Knotens.

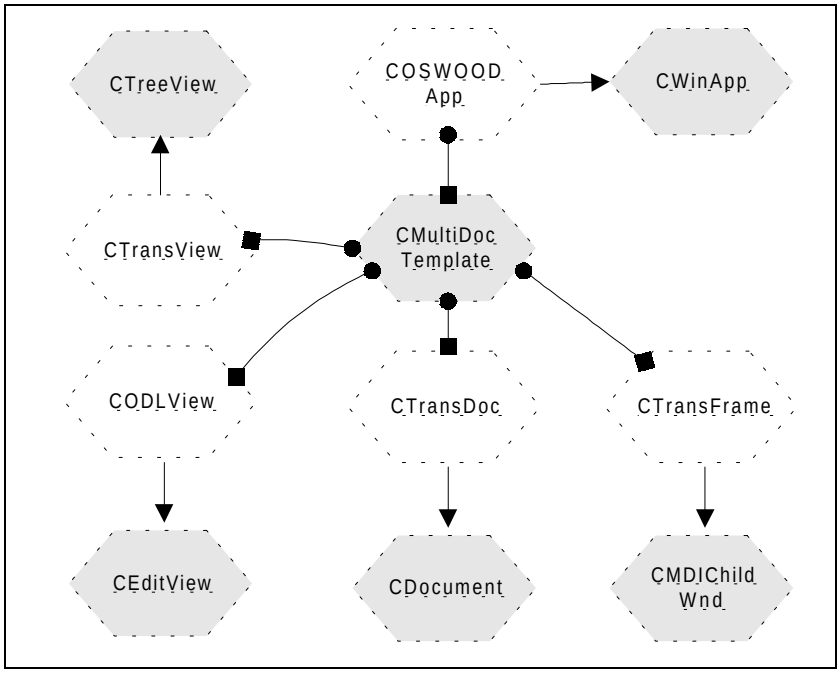
`arc_image`
definiert das Aussehen eines Verbinders.

7.5.2 Ablaufbeschreibung

Der Analyse Prozess läuft in mehreren Stufen ab:

1. Lesen der Transaktionsdiagrammdatei (Umsetzung der Elemente in Knoten und Kanten)
2. Validierungen
3. Aufbau der Aufrufhierarchie
4. Aufbau der Subsystemhierarchie
5. Erzeugung der Klassendefinitionen und Methodenimplementierungen
6. Schreiben der ODL-Datei

7.5.3 Einbettung



Klassendiagramm 7-7: Einbettung der Transaktionsdiagrammklassen

Klassen von OSWOOD

`COSWOODApp`

Applikationsklasse von OSWOOD. Verwaltet das Hauptfenster und die Dokumententypen. Einzige Klasse, welche eine globale Instanz besitzt.

`CTransDoc`

verwaltet die Informationen eines Transaktionsdiagramms.

`CTransFrame`

verwaltet das Dokumentenfenster.

`CTransView`

verwaltet die Transaktionsansicht.

`CODLView`

verwaltet die ODL-Ansicht.

Klassen der MFC

`CWinApp`

Applikationsklasse.

`CMultiDocTemplate`

kapselt einen bestimmten Dokumententyp, der durch seine Dokumenten-, Rahmen- und Ansichtsklasse identifiziert wird.

`CMDIChildWnd`

verwaltet das Kindfenster in einer MDI-Applikation.

`CTreeView`

Derivat der generellen Ansichtsklasse CView für die Verwaltung von Baumansichten.

`CEditView`

Derivat der generellen Ansichtsklasse CView für die Verwaltung von textuellen Ansichten.

`CDomument`

verwaltet die Informationen eines Dokumentes.

7.5.3.1 CTransDoc

Die Klasse speichert sämtliche Informationen, die in einem Schemadiagramm enthalten sind und sämtliche Resultate der

Interpretation. Sie steuert den Ablauf und reagiert auf Benutzeraktionen.

CTransDoc : public CDocument
`void ReadDiagram(CHardyFile& DiagramFile)` Liest gemäss beschriebenem Vorgehen die Transaktionsdiagrammdatei (vgl. "Lesen einer Hardydatei", Seite 193)
`void ValidatePrimitives(void) const` Validiert gewisse Konditionen, welche von DEIMOS gefordert werden (vgl. "Validierungen", Seite 240)
`void CreateCallHierarchy(void)` Baut die Aufrufhierarchie auf (vgl. "Aufbau der Aufrufhierarchie", Seite 241)
`void CreateSubSystemHierarchy(void)` Baut die Subsystemhierarchie auf (vgl. "Aufbau der Subsystemhierarchie", Seite 243)
`void CreateImplementation(void)` Erezugt die Implementationen für die einzelnen Elemente (vgl. "Erzeugung der Implementierungen", Seite 244)
`void WriteO2Cfile(CArchive& arFile)` Schreibt die ODL-Datei (vgl. "Schreiben der ODL-Datei", Seite 244)
`CArray<CItem,CItem&> m_aItem` Sammlung aller nicht weiter typisierten Elemente (Reproduktion)
`CMainProgram m_MainProgram` Hauptprogramm oder Applikation
`CArray<CSubSystem,CSubSystem&> m_aSubSystem` Sammlung der Subsysteme
`CArray<CProgram,CProgram&> m_aProgram` Sammlung der Programme
`CArray<CFunction,CFunction&> m_aFunction` Sammlung der Funktionen
`CArray<CTransaction,CTransaction&> m_aTransaction` Sammlung der Transaktionen
`CArray<CCallArc,CCallArc&> m_aCallArc` Sammlung der Aufrufpfeile

Klassenbeschreibung 7-13: CTransDoc

```
class CTransDoc : public CDocument {
                  // methods
                  public:
                        void ReadDiagram(CHardyFile&
DiagramFile);
                        void WriteO2CFile(CArchive& arFile);
                  private:
                        void CreateItemList(CHardyFile&
DiagramFile);
                        void AddItem(const CItem& Item);

                        void ValidatePrimitives(void) const;

                        void CreateCallHierarchy(void);
                        void CreateCallHierarchy(CProgram*
pItem);
```

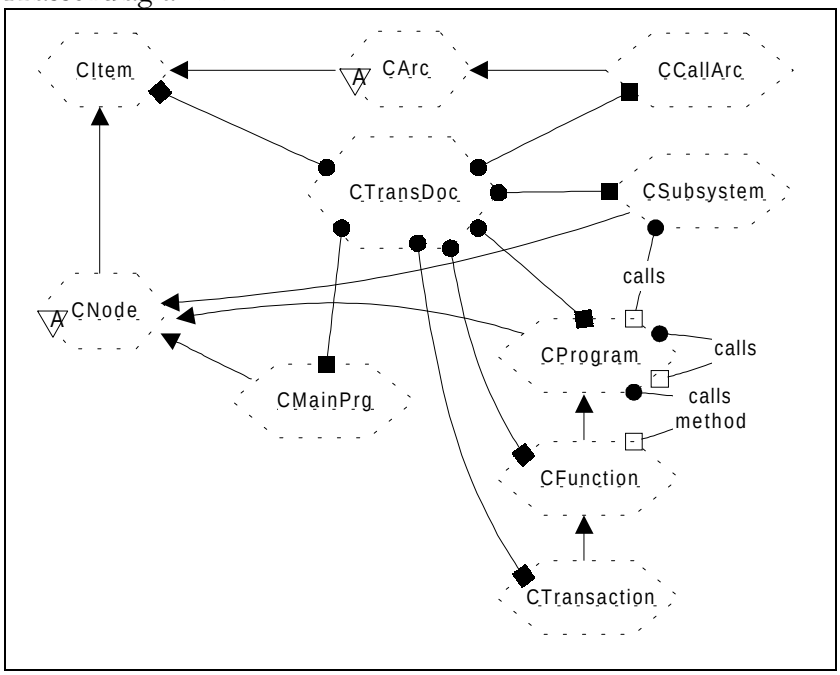
```
                        int GetCalledItemId(int
nFromItem,int& nId);
                        CProgram* GetCalledItem(int nId);

                        void CreateSubSystemHierarchy(void);
                        int GetCardId(CString
strIdentifier,const CString& strValue);
                        int GetCardId(int nNodeImageId);

                        void CreateImplementation(void);

                  // Attributes
                  public:
                        // item list
                        CArray<CItem,CItem&> m_aItem;

                        // node lists
                        CMainProgram m_MainProgram;
                        CArray<CSubSystem,CSubSystem&>
m_aSubSystem;
                        CArray<CProgram,CProgram&>
m_aProgram;
                        CArray<CFunction,CFunction&>
m_aFunction;
                        CArray<CTransaction,CTransaction&>
m_aTransaction;

                        // arc lists
                        CArray<CCallArc,CCallArc&>
m_aCallArc;

};  //end class CTransDoc
```

Listing 7-43: Klassendefinition von CTransDoc

7.5.4 Lesen der Transaktionsdiagrammdatei

Hardyfile Parsing
Erstellen der Items
Parsing der name (params) return
Methoden (Implementation der expliziten Methoden)
Contraints,
Attribute, Schlüssel)
Für Kanten: Lesen der Kardinalitätsangaben

Klassendiagramm



Klassendiagramm 7-8: die Klassen eines Transaktionsdokumentes

Klassen von OSWOOD

`CMainProgram`
kapselt ein Hauptprogramm (Applikation). Speichert als Attribute den Namen und die Implementation. Überschreibt - wie alle Derivate von `CItem` - die `Validate`-Methode.

`CSubsystem`
kapselt ein Subsystem. Speichert als Attribute den Namen und die Karte, auf welcher es sich befindet. Überschreibt - wie alle Derivate von `CItem` - die `Validate`-Methode. Hat für die Repräsentation der Subsystemhierarchie eine Sammlung von Zeigern zu den Programmen, die in ihm enthalten sind. Diese Zeiger werden mit der Methode `AddCall()` aufgebaut.

`CProgram`
repräsentiert Programme und ist gleichzeitig die Basisklasse für die Funktionen und Transaktionen. (Genauere Beschreibung siehe weiter unten.)

`CFunction`
repräsentiert eine Funktion.

`CTransaction`
entspricht der Klasse CProgram, erweitert lediglich die `Validate`-Methode.

`CCallArc`
widerspiegelt einen Aufrufpfeil und besitzt als einziges Attribut die Aufrufnummer.

7.5.4.1 CProgram

CProgram : public CNode

`void AddCall(CProgram* pCalledItem,int nNumber)`
Fügt unter Berücksichtigung der Aufrufnummer `nNumber` einen Aufruf zu dem Element `pCalledItem` der Sammlung der Aufrufe hinzu
`virtual void CreateImplementation(const CString& strAppName)`
Erzeugt die Implementierung
`void CreateMethodCalls(void)`
Erzeugt die Methodenaufrufe

`CString m_strDefinition`
Speichert im Falle der Funktion den Definitionsstring für den Prototypen
`CString m_strObject`
Gibt im Falle einer Methode die Klasse an, in welcher die Methode zu suchen ist
`CString m_strName`
Speichert den Namen des Programms, der Funktion, der Transaktion oder der Methode
`CString m_strType`
Speichert den Typ des Rückgabewertes
`CArray<CString,CString&> m_astrParamName`
Liste der Parameternamen
`CArray<CString,CString&> m_astrParamType`
Liste der Parametertypen

`int m_nNodeImageId`
Speichert den Identifikator des Knotenbildes für den Aufbau der Subsystemhierarchie

`CWordArray m_awCalledItemId`
Speichert die Aufrufnummern aufsteigend sortiert, damit in der Umsetzung der Aufrufe die Reihenfolge beibehalten werden kann.

`CPtrArray m_apCalledItem`
Speichert die Zeiger zu den aufgerufenen Elementen

`CArray<CFunction,CFunction&> m_aCalledMethod`
Sammlung der aufgerufenen Methoden

`CString m_strImplementation`
Implementation

Klassenbeschreibung 7-14: CProgram

```
class CProgram : public CNode {
                    public:
                         typedef enum {
                                typeProgram=3,
                                typeFunction=4,
                                typeTransaction=5,
                                typeMethod=6
                         } Type;
                    // construction / destruction
                    public:
                         CProgram() {};
                         CProgram(const CItem& Item) :
CNode(Item) {};

                    // operations
                    public:
                         CProgram& operator=(CProgram&
Program);

                    // methods
                    public:
                         virtual void Validate(void);
                         void AddCall(CProgram*
pCalledItem,int nNumber);
                         virtual void
CreateImplementation(const CString& strAppName);
                         CString GetLabel(void);
                         virtual int GetImageId();
                         void CreateMethodCalls(void);
                         void Show(HTREEITEM hItem,CTreeCtrl&
TreeCtrl,UINT nMask);

                    // members
                    public:
                         CString m_strDefinition;

                         CString m_strObject;
                         CString m_strName;
                         CString m_strType;
                         CArray<CString,CString&>
m_astrParamName;
                         CArray<CString,CString&>
m_astrParamType;

                         int m_nNodeImageId;
                         Type m_Type;

                         CString m_strCall;
                         CWordArray m_awCalledItemId;
                         CPtrArray m_apCalledItem;
                         CArray<CFunction,CFunction&>
m_aCalledMethod;

                         CString m_strImplementation;
}; //end class CProgram
```

Listing 7-44: Klassendefinition von CProgram

7.5.5 Validierungen

Nachdem die Elemente aus der Hardydatei vollständig gelesen worden sind, können bereits weiterreichende Validierungen der Diagramminformationen durchgeführt werden.
Es wird insbesondere geprüft, ob

- genau ein Hauptprogramm im Diagramm enthalten ist und dieses einen Namen besitzt.
- mindestens ein Programm und
- mindestens eine Transaktion aufgerufen wird.

Ist die erste Bedingung nicht erfüllt, handelt es sich um einen fatalen Fehler, da dem Aufrufbaum die Wurzel fehlt. Die weiteren Tests prüfen, ob das gelesene Diagramm sinnvoll ist.
Weitere Validierungen (Eindeutigkeit, Zyklenfreiheit, etc.) werden laufend während der nachfolgenden Verarbeitung vorgenommen und müssen demnach nicht speziell in diesem Abschnitt behandelt werden.

7.5.6 Aufbau der Aufrufhierarchie

Damit Code generiert weden kann, müssen die Informationen der Aufrufkanten in echte Aufrufe umgesetzt werden. Ziel ist es, dass jedes Programm und jede Funktion eine Liste von aufgerufenen Elementen erhält. Zudem sollen die vom Benutzer in die Variable `pseudo code` geschriebenen Methodenaufrufe dargestellt und in Code umgesetzt werden.
Für den Aufbau dieser Aufrufhierarchie ist in der Klasse `CTransDoc` die untenstehende Funktion `CreateCallHierarchy` zuständig:

```
void CTransDoc::CreateCallHierarchy(void) {
                        int i;
(1)                     for (i=0;i<m_aProgram.GetSize();i++) {
(2)                          CreateCallHierarchy(&m_aProgram[i]);
(3)                          m_aProgram[i].CreateMethodCalls();
                        }//endfor
                        for (i=0;i<m_aFunction.GetSize();i++) {
                             CreateCallHierarchy(&m_aFunction[i]);
                             m_aFunction[i].CreateMethodCalls();
                        }//endfor
                        for (i=0;i<m_aTransaction.GetSize();i++) {

                        m_aTransaction[i].CreateMethodCalls();
                        }//endfor
}//end CreateCallHierarchy
```

Listing 7-45: Methode CreateCallHierarchy() der Klasse CTransDoc

(1) Für alle Programme, Funktionen und Transaktionen wird erst
(2) die Methode zum Aufbau der Aufrufe von dem aktuellen Element aus und anschliessend
(3) die Methode zum Erzeugen der Methodenaufrufe durchgeführt.

Die Aufrufe werden durch die folgende Methode kreiert:

```
void CTransDoc::CreateCallHierarchy(CProgram* pItem) {
                        int nArcId=-1;
                        int nCalledItemId=0;
                        CProgram* pCalledItem;

                        // create references to called diagram
items
(1)                     while (TRUE) {
(2)                          nCalledItemId=GetCalledItemId(pItem-
>m_nId,nArcId);
                             if (nArcId==-1) break;
(3)
                        pCalledItem=GetCalledItem(nCalledItemId);
                             ASSERT(pCalledItem);
(4)                          pItem-
>AddCall(pCalledItem,m_aCallArc[nArcId].m_nNumber);
                        }//endwhile

}//end CreateCallHierarchy
```

Listing 7-46: Methode CreateCallHierarchy(...) der Klasse CTransDoc

(1) Solange noch Aufrufpfeile mit dem gegebenen Element als Quelle existieren,
(2) Suche in der Sammlung der Aufrufpfeile nach einer derartigen Kante,
(3) beschaffe das durch diese Kante referenzierte Element und
(4) kreiere einen Aufruf zu diesem Element.

Ein Aufruf wird folgendermassen erzeugt:

```
void CProgram::AddCall(CProgram* pCalledItem,int nNumber) {

                        // insert the new item in the right
position
(1)                     for (int
i=0;i<m_awCalledItemId.GetSize();i++) {
                              if (nNumber<m_awCalledItemId[i])
break;
                        }//endfor

(2)
                        m_awCalledItemId.InsertAt(i,(WORD)nNumber)
;
(3)                     m_apCalledItem.InsertAt(i,pCalledItem);

(4)                     if ((i>0)&&(m_awCalledItemId[i-
1]==nNumber))
                              WARNING("Douplicate usage of number
%u.",nNumber);
}//end AddCall
```

Listing 7-47: Methode AddCall(...) der Klasse CProgram

(1) Der neue Aufruf muss an die richtige Position der Aufrufreihenfolge, die im Attribut `awCalledItemId` gespeichert worden ist, eingefügt werden. Die Aufrufnummer wurde bereits beim Lesen des Elementes in die Variable `m_nNumber` der Aufrufklasse geschrieben und in diese Funktion übergeben.
(2) Die Aufrufnummer und
(3) das aufgerufene Element werden in die Sammlungen aufgenommen.
(4) Wurde eine Aufrufnummer mehrfach verwendet, wird eine ensprechende Warnung ausgegeben.

Die Angaben in den Feldern `'pseudo code'` oder `calls` werden mit Hilfe der folgenden Methode in Methodenaufrufe umgesetzt:

```
void CProgram::CreateMethodCalls(void) {
                        if (m_strCall=="") return;

                        CString strCalls=m_strCall;
                        CString strCall;
                        int nPos;
                        BOOL bMore=TRUE;
                        CFunction Method;
                        CFunction* pMethod;
                        CParser Parser;

(1)                     while (bMore) {
(2)                           if
((nPos=strCalls.Find(char(13)))>=0) {
(3)                                 strCall=strCalls.Left(nPos);
(4)                                 strCalls=strCalls.Mid(nPos+3);
                                    bMore=TRUE;
                                    }
                              else {
(5)                                 strCall=strCalls;
                                    strCalls="";
                                    bMore=FALSE;
                              }//endif
(6)                           m_aCalledMethod.Add(Method);

                        pMethod=&m_aCalledMethod[m_aCalledMethod.G
etUpperBound()];
                              pMethod->m_strDefinition=strCall;
```

```
(7)                          Parser.ParseCall(strCall,pMethod-
>m_strObject,
pMethod->m_strName,
pMethod->m_strType,
pMethod->m_astrParamName,
pMethod->m_astrParamType);
                             pMethod->m_Type=CProgram::typeMethod;
                         }//endwhile

}//end CreateMethodCalls
```

Listing 7-48: Methode CreateMethodCalls(...) der Klasse CProgram

(1) Solange noch weitere Zeilen vorhanden sind,

(2) Suche in der Aufrufzeichenkette nach dem end-of-line Zeichen und

(3) speichere den linken Teil als den aktuellen Aufruf und

(4) den Rest in der Aufrufzeichenkette für den nächsten Schleifendurchlauf.

(5) Ist keine weitere Zeile in der Aufrufzeichenkette enthalten, speichere den Rest als den aktuellen Aufruf.

(6) Erzeuge eine neue, leere Methode und

(7) rufe mit deren Attributen den Parser für die Analyse des Methodenaufrufes auf.

7.5.7 Aufbau der Subsystemhierarchie

Die in einer Applikation enthaltenen Programme sind üblicherweise in verschiedenen Subsystemen strukturiert. Diese Hierarchie, welche in Hardy mit mehreren Karten dargestellt ist, aber dennoch mit derselben Hardydatei auskommt, muss in diesem Schritt nun erkannt und in eine sinnvolle Datenstruktur übertragen werden.
Dies wird mit der Methode `CreateSubSystemHierarchy` in der Klasse `CTransDoc` erledigt:

```
void CTransDoc::CreateSubSystemHierarchy(void) {
                         // for each subsystem look for expansion
card
                         int i;
(1)                      for (i=0;i<m_aSubSystem.GetSize();i++) {

                         m_aSubSystem[i].m_nCardId=GetCardId("title
",
m_aSubSystem[i].m_strName);
                         }//endfor

                         // assign each program to appropriate
subsystem
                         int nCardId;
(2)                      for (i=0;i<m_aProgram.GetSize();i++) {
(3)
                         nCardId=GetCardId(m_aProgram[i].m_nNodeIma
geId);
                              if (!nCardId) continue;
                              for (int
j=0;j<m_aSubSystem.GetSize();j++) {
(4)                                    if
(m_aSubSystem[j].m_nCardId==nCardId) break;
                              }//endfor
                              if (i==m_aSubSystem.GetSize())
continue;
(5)
                         m_aSubSystem[j].AddCall(&m_aProgram[i]);
                         }//endfor

}//end CreateSubSystemHierarchy
```

Listing 7-49: Methode CreateSubSystemHierarchy(...) der Klasse CTransDoc

(1) Für jedes bereits erkannte und gelesene Subsystem wird diejenige Karte bestimmt, die seiner Expansion entspricht.

(2) Für jedes Programm

(3) wird der Identifikator derjenigen Karte geholt, auf welcher das Programm gezeichnet worden ist. Diese Information ist nicht direkt

in den Einträgen des Knotens enthalten, weshalb der Umweg über den Eintrag des entsprechenden Darstellungsknotens (`node_image`) genommen werden muss.
(4) Anschliessend wird aus der Sammlung der Subsysteme dasjenige gesucht, welches die Karte des Programmes als Expansion besitzt.
(5) Durch die Installation eines künstlichen Aufrufes zwischen diesen beiden Elementen kann der Effekt erzielt werden, dass die Programme eines Subsystemes diesem in der baumartigen Darstellung unterstellt sind.

7.5.8 Erzeugung der Implementierungen

Für jedes Element der Sammlungen der Programme, Funktionen und Transaktionen wird die Methode `CreateImplementation()` aufgerufen. Der Code für die Definition einer Applikation muss die Prototypen der Programme und Transaktionen enthalten, weshalb ihrer Methode die entsprechenden Sammlungen übergeben werden müssen.

```
void CTransDoc::CreateImplementation(void) {
                        int i;
                        for (i=0;i<m_aProgram.GetSize();i++) {

                        m_aProgram[i].CreateImplementation(...);
                        }//endfor
                        for (i=0;i<m_aFunction.GetSize();i++) {

                        m_aFunction[i].CreateImplementation(...);
                        }//endfor
                        for (i=0;i<m_aTransaction.GetSize();i++) {

                        m_aTransaction[i].CreateImplementation(...
);
                        }//endfor

                        m_MainProgram.CreateImplementation(m_aProg
ram,m_aTransaction);
}//end CreateImplementation
```

Listing 7-50: Methode CreateImplementation(...) der Klasse CTransDoc

7.5.9 Schreiben der ODL-Datei

Die ODL-Datei wird analog zum Schreiben der ODL-Datei für das Schemadiagramm ein Kopf für die spätere Erkennung vorangestellt (vgl. Listing 7-40, Seite 233)
Im Anschluss an die Kopfinformation wird die Sektion mit der Definition der Applikation in die Datei geschrieben:

```
/*////////////////////////////////////////////////////*/
/* FIS application definition */

application FIS
  program
    /* programs */
    public HireDeveloper,

    ...

    public AssignEmplToOff,

    /* transactions */
    public FireDev(refDev:Dev) : boolean,

    ...

    public DeleteOffice() : boolean
end;
```

Listing 7-51: Applikationsdefinition in der ODL-Datei

Anschliessend werden sämtliche Programme, Funktionen und Transaktionen der Datei angehängt:

```
/*////////////////////////////////////////////////////*/
/* HireDeveloper program implementation */
```

```
program body HireDeveloper in application FIS {

  ...

} /* end HireDeveloper */

...

/*////////////////////////////////////////////////////*/
/* SelectDev function implementation */

/* function prototype */
function SelectDev(strName:String,nNumber:Integer) : Dev;

/* function implementation */
function body SelectDev {

  ...

} /* end SelectDev */

...

/*////////////////////////////////////////////////////*/
/* FireDev transaction implementation */

transaction body FireDev in application FIS {

  ...

} /* end FireDev */

...
```

Listing 7-52: Programme, Funktionen und Transaktionen in der ODL-Datei

7.6 Bewertung der Implementation

Die Wahl der Entwicklungsumgebung Microsoft Visual C++ erwies sich im Nachhinein als sehr glücklich. Durch die Verwendung der reichhaltigen Bibliothek der MFC konnten verschiedene Nebenprobleme gelöst werden. Für die Programmierung der Visualisierung, der Benutzeraktionen, oder kurz des GUI musste nicht allzuviel Zeit aufgewendet. Somit blieb viel Zeit, sich auf das Kernproblem konzentrieren zu können.
Darüber hinaus konnte durch die Verwendung der Templates für Kollektionen der MFC der ganze Code ohne eine einzige `new`-Anweisung und ohne einen einzigen exliziten Aufruf eines Destruktors realisiert werden.
Das „Microsoft Developer Studio“ unterstützte dabei die Arbeit durch seine integrierten Browser, die auch mit unkompiliertem Code arbeiten können, seinem integrierten Debugger, der durch seine „Debug-on-error“-Fähigkeit die Fehlersuche erheblich erleichtert, seinen „Application-„, bzw. „Class-Wizard“ das Erstellen und Verwalten von Applikationen, bzw. Klassen sicher macht und durch seinen „Memory-Leak-Detector“ die Betriebssicherheit garantiert.
Durch den Einsatz dieser komfortablen Werkzeuge mit ihren ausgezeichneten Möglichkeiten konnte in sehr kurzer Zeit eine mächtige und benutzerfreundliche Anwendung geschaffen werden.

7.6.1 Bekannte Probleme in OSWOOD

Nachfolgend sollen die bereits bekannten Probleme aufgeführt werden. Noch sind aber mit dem Werkzeug keine weiterführenden Feldtests durchgeführt worden. Man kann also davon ausgehen, das sich zu dieser Liste noch weitere gesellen werden.
Die Probleme, welche die methodischen oder konzeptuellen Aspekte des Entwurfes betreffen sind bereits an anderer Stelle diskutiert

worden und werden aus diesem Grund nicht in die Liste aufgenommen.

Typen

Es werden nicht alle in O_2 bekannten Typen unterstützt. Definiert man z.B. `setrefName : unique set(Name)` als Parameter oder Attribut im Schemadiagramm, wird `unique set(Name)` als Klassenname und nicht als Mengentyp interpretiert.

Aufrufe von Methoden der Basisklasse

Erbt eine Klasse B von einer Klasse A, so wird beispielsweise beim Aufruf der Methode `init()` in der Klasse B die entsprechende `init`-Methode der Vaterklasse nicht automatisch aufgerufen. Dieselbe Situation trifft man bei der impliziten `Assign`-Methode der Objektevolution.

Verbindung zwischen dem Schema- und dem Transaktionsdiagramm

In den Programmen, Funktionen oder Transaktionen formulierte Methodenaufrufe werden nicht mit den entsprechenden Definitionen im Klassenschema abgeglichen. Die beiden Diagrammtypen sind also völlig ungebunden. Es wäre dabei durchaus sinnvoll, das Werkzeug so zu erweitern, dass die Diagramme, welche eigentlich gemeinsam das Schema definieren, in eine Ansicht zusammengefasst sind. Damit wären beispielsweise die Methodenimplementierungen in der Transaktionsansicht integriert und erweiterte Browsing-Funktionen möglich.

Eindeutigkeit der Namen

OSWOOD prüft die Eindeutigkeit der Namen nicht. Somit kann man zum einen zwei identische Klassennamen im selben Diagramm verwenden, welche von OSWOOD wie zwei verschiedene Klassen behandelt werden. Zum anderen werden zwei gleichnamige Klassen, die in unterschiedlichen Diagrammen definiert sind, von OSWOOD nicht zusammengefasst.

Anzeige der O_2C-Dateien

O_2C-Dateien, die grösser als 64KB sind, können nicht vollständig angezeigt werden. Es werden lediglich die ersten 64KB in den Bildschirmpuffer geschrieben.

Reihenfolge der Klassendefinitionen

Die Klassendefinitionen werden in der Reihenfolge in die O_2C-Datei geschrieben, in der sie im Komponentenbaum auftreten (vgl. “Der Komponentenbaum”, Seite 143). Darum kann es vorkommen, dass Subklassen von Vaterklassen erben (insbesondere bei abstrakten Klassen, die ganz zum Schluss in den Komponentenbaum eingefügt worden sind), die erst später im Code definiert sind.

Persistente Einstiegspunkte

Die Namen der persistenten Einstiegspunkte sollen erst definiert werden, wenn die dazugehörige Klasse bereits bekannt ist.

Verteilung des Codes auf Dateien

OSWOOD schreibt alle Definitionen des Schemadiagramms in eine Datei und diejenigen des Transaktionsdiagramms in eine andere Datei. Würde man die strikte Trennung der beiden Ansichten aufheben, könnte eine intuitivere Aufteilung implementiert werden. Man würde beispielsweise je eine Datei für die Klassen- und

Applikationen, für die Namen, für die Methodenimplementierungen und für die Transaktionsimplementierungen erstellen.

7.6.2 Mögliche Erweiterungen

Falls OSWOOD in der Praxis - z.B. im Bereich der Erstellung wissenschaftlicher Datenbankanwendungen - eingesetzt wird, werden aller Wahrscheinlichkeit nach sowohl von den Entwerfern als auch von den Entwicklern eine Reihe von Wünschen für die Weiterentwicklung geäussert werden. Diese werden aller Voraussicht nach etwa in den in der nachfolgenden Liste aufgezählten Bereichen liegen:

Aufhebung der Trennung der Diagramme
In der Zeichnungsumgebung werden eine Reihe Diagramme gezeichnet, welche aber in OSWOOD nicht mehr getrennt betrachtet werden sollen.

Integration der restlichen Diagrammtypen
Heute können Schemadiagramme und Transaktionsdiagramme gezeichnet und in OSWOOD interpretiert werden. DEIMOS enthält aber auch die restlichen Diagrammtypen wie Zustandsübergangsdiagramme, etc. Diese sollten auch von OSWOOD in die Definition des Schemas einbezogen werden.

Zusammenfassung von Hardy und OSWOOD
Zum Zeichnen von Diagrammen und zum Generieren von Code sind zwei unterschiedliche Programme zu verwenden. Zwar kann von OSWOOD in den Zeichenmodus verzweigt werden, die dort getätigten Änderungen hingegen werden nicht automatisch in OSWOOD sichtbar. Der Entwerfer wünscht sich jedoch eine einzige Anwednung, die alle seine Wünsche abdeckt.

Änderungen in OSWOOD
In OSWOOD kann nur gelesen werden. OSWOOD zeigt also nur eine andere Ansicht der Daten an. Wünschenswert wäre die Möglichkeit der Editierung von Klassenstrukturen, Methodensignaturen oder Implmentierungen. Diese Änderungen sollten anschliessend in die Zeichnungen von Hardy zurückgeführt werden.

Zusammenführen der Plattformen
OSWOOD ist ausschliesslich für die Betriebssysteme Windows 95, bzw. Windows NT verfügbar. Um die Schemadefinitionen in das Datenbanksystem einfügen zu können, muss also die Plattform gewechselt werden. Aus diesem Grund wäre eine Portierung von OSWOOD auf die für O_2 verfügbaren Plattformen durchaus sinnvoll.

„Reverse Engineering“ und Schemaevolution
Befinden sich die beiden Systeme OSWOOD und O_2 auf denselben Plattformen, könnten auch Funktionen für das Rückführen von bestehenden Schemata in die Entwurfsumgebung („Reverse Engineering“) und für die Schemaevolution realisiert werden. Damit wäre OSWOOD so nahe ans Datenbanksystem angelehnt, dass es eigentlich in die grafische Benutzeroberfläche des O_2-Tools integriert werden sollte.

7.6.3 Fazit

OSWOOD ist in seiner heutigen Implementierung ein gutes Werkzeug für die Erstellung von Schemagerüsten für die Entwickler. Die verschiedenen Schwachstellen und Erweiterungen müssen aber gegebenenfalls beseitigt, bzw. eingebaut werden, damit OSWOOD eine gewisse Akzeptanz als Hilfsmittel für die Erstellung von praxisrelevanten Datenbankanwendungen erhalten kann.

ANHÄNGE

Unified Modeling Language (UML)

Einleitung

Weil die Vereinheitlichungsanstrengungen von Booch, Jacobsen und Rumbaugh während der Ausarbeitung der vorliegenden Diplomarbeit in der Version 1.0 des Papiers "Unified Modeling Language (UML)" gipfelten, und weil diese neuen Methode gute Chancen hat, zur Standardmethode für den objektorientierten Entwurf zu avancieren, soll ihr dieser Anhang gewidmet sein.
Zuerst sollen einige wichtige Eckdaten in der Geschichte der UML genannt werden. Anschliessend soll kurz beschrieben werden, was die UML ist und inwiefern sie sich von der betrachteten Methode von Booch unterscheidet. Zum Schluss soll untersucht werden, wie die Schwachstellen ausgesehen hätten, wäre man von UML ausgegangen und wie DEIMOS angepasst werden müsste, damit eine Konsistenz zur UML erreicht werden könnte.
Dem Anhang liegt vor allem die Beschreibung der UML in [Booch et. al. 97] zu Grunde. Da bis zum Ende dieser Arbeit keine deutschen Übersetzungen vorliegen und auch weitere deutsche Publikationen fehlen ([Oestereich 97] basiert auf [Booch et. al. 96]), wurde entsprechend auf die Übersetzung der neu eingeführten Begriffe verzichtet und an deren Stelle die englischen Originale verwendet.

Geschichte von UML

Bevor die Entwicklung der vereinheitlichten Methode UML begann, waren im Bereich der objektorientierten Analyse- und Entwurfsmethoden hauptsächlich drei Autoren federführend: Grady Booch [Booch 94], Jim Rumbaugh mit OMT [Rumbaugh et. al. 91] und Jacobsen mit OOSE [Jacobsen 92]. Alle diese Methoden sind komplett, aber auch bekannt dafür, dass sie eine gewisse Strenge besitzen.
Booch's Methode zeichnet sich dadurch aus, dass sie speziell ausdrucksstark während dem Entwurf und den Konstruktionsphasen in Projekten mit grossem Engineering Aufwand sind, während sich OMT speziell für die Analyse von datenintensiven Informationssystemen eignet. OOSE hingegen ist ein anwendungsfall-orientierter Ansatz (use case) und unterstützt daher exzellent die Modellierung der Geschäftsprozesse („Business Engineering") und die Anforderungsanalyse („Requirement Analysis").

Booch, Rumbaugh und Jacobsen vereinen ihre Kräfte

Die Entwicklung der UML begann 1994 als Booch und Rumbaugh die Vereinheitlichung ihrer Methoden, die unabhängig voneinander bereits zusammengewachsen waren, in Angriff nahmen. Daraus resultierte 1995 die Version 0.9 der Unified Method (UM), welche in [Booch et. al. 95] beschreiben ist. Zu diesem Zeitpunkt stiess Jacobsen zu der Gruppe hinzu, um die Aspekte seiner Methode

OOSE in eine noch grössere Vereinheitlichung einbringen zu können.
Das nun dreiköpfige Gespann wurde in ihrer Arbeit durch drei Aspekte motiviert:
1. Die verschiedenen Methoden haben sich ohnehin unabhängig zu einander entwickelt. Sie erachteten es deshalb als sinnvoll, die Entwicklung gemeinsam fortzusetzen, damit die Benutzer nicht weiter durch unnötige und ungewollte Differenzen verwirrt werden.
2. Durch die Vereinigung der Semantik der Notationen kann eine Stabilität im Markt der objektorientierten Methoden erreicht werden. Damit können einerseits die Systementwicklungen auf einer Hauptmethode basieren und andererseits können sich die Werkzeughersteller auf die Unterstützung einer einheitlichen Methode konzentrieren.
3. Sie erhofften, durch ihre Zusammenarbeit diejenigen Probleme lösen zu können, die keine der drei Methoden in der Vergangenheit richtig bewältigen konnte.
Während der Entwicklung sollte darüber hinaus auf verschiedene Ziele hin gearbeitet werden:
1. Modellierung von Systemen (nicht nur von Software) unter der Verwendung von objektorientierten Konzepten.
2. Erreichung einer expliziten Koppelung sowohl der konzeptuellen als auch der ausführbaren Artefakten.
3. Erzeugung einer Modellierungssprache sowohl für den Menschen als auch für die Maschine.
Die Erarbeitung einer Methode für die objektorientierte Analyse und den objektorientierten Entwurf ist nicht mit dem Entwurf einer Programmiersprache vergleichbar. Erstens muss das Problem eingegrenzt werden (soll die Notation die Spezifikation der Anforderungen umschliessen? Soll die Notation zu einer visuellen Programmiersprache ausgebaut werden?) und zweitens muss man einen Ausgleich zwischen Ausdrucksstärke und Einfachheit schaffen. Ist die Methode zu einfach, wird die Breite der Probleme, die damit gelöst werden können, zu stark eingeschränkt. Gestaltet sich die Methode hingegen zu kompliziert, überwältigt sie die Möglichkeiten der sterblichen Entwickler.
Die Arbeit der Vereinheitlichung muss zudem sensitiv zu der bestehenden Basis sein. Werden zu viele Änderungen vorgenommen, werden die existierenden Benutzer verwirrt, was sich entsprechend negativ auf die Akzeptanz auswirkt. Wird die Notation aber nicht entscheidend erweitert, entgeht der Methode die Gelegenheit, den Benutzerkreis zu verbreitern. UML versucht bei diesem Balanceakt den geeigneten Kompromiss zu finden.

Zukunft von UML

Die Beschreibung der UML ist Anfang 1997 in seiner Version 1.0 der OMG als Vorschlag für eine Standardmethode unterbreitet worden (vlg [Booch et. al. 97]). In der ersten Hälfte des Jahres 1997 wird nun eine entsprechende Stellungnahme des Gremiums erwartet, damit die Voraussetzungen erfüllt werden können, um UML ab Mitte 1997 als Standardmethode für die objektorientierte Modellierung proklamieren zu können. Unabhängig davon werden im Verlauf des

Jahres 1997 verschiedene Publikationen (u.a. auch die dritte Auflage von [Booch 94], welche die Aspekte der UML berücksichtigt) erwartet.

Was ist UML?

Die Eigenschaften der UML können folgendermassen zusammengefasst werden:

- UML ist eine Sprache für die Spezifikation, Konstruktion, Visualisierung, und Dokumentation der Artefakte eines softwareintensiven Systems.
- UML vereint die Konzepte von Booch, OMT und OOSE. Das Resultat ist eine einzige und einfach anwendbare Modellierungssprache für Benutzer, welche in der Vergangenheit bereits eine der genannten oder eine andere objektorientierte Methoden eingesetzt haben.
- UML zeigt, was mit ihr erreicht werden kann. Insbesondere wurden Konstrukte für die Modellierung von konkurrenzierenden, verteilten Systemen eingeführt.
- UML definiert eine Standardsprache für die Modellierung und verzichtet auf die Beschreibung eines Standardprozesses. Erfahrungen aus der Anwendung der verschiedentlich vorgeschlagenen Vorgehensrichtlininen im Projektkontext haben gezeigt, dass verschiedene Organisationen und Problembereiche unterschiedliche Prozesse benötigen. Deshalb wurde zuerst ein Metamodell geschaffen, welches die Semantik vereint, und anschliessend eine Notation definiert, welche diese Semantik umsetzt. Die Autoren standardisieren also keinen Prozess, dennoch werden in den prozessspezifischen Extensionen (vgl. [UML Extensions 97]) Entwicklungsprozesse vorgeschlagen, die anwendungsfallgesteuert (use-case-driven), bezüglich der Architektur generisch, iterativ und inkrementell sind.

Die Beschreibung der UML ist eine Sammlung von verschiedenen Einzeldokumenten. Im einzelnen sind dies:

- UML Summary [UML Summary 97]: fasst die weiteren Papiere zusammen und gibt eine Einblick und Überblick über UML. Enthält geschichtliche Angaben zu UML.
- UML Semantics [UML Semantics 97]: beschreibt das präzise Modell, welches UML unterliegt.
- UML Notation Guide [UML Notation 97]: Beschreibt die UML Notation und enthält Beispiele.
- UML Process Extension [UML Extensions 97]: enthält gewisse prozessspezifische Werte der Erweiterungsmechanismen.

Das Modell der UML definiert eine Reihe von grafischen Diagrammen. In der nachfolgenden Aufzählung sollen diese benannt und deren Herkunft (in Klammern) bestimmt werden. Die Diagrammtypen werden in die Bereiche des statischen Entwurfs, des dynamischen Entwurfs und der Implementation eingeteilt, wobei auf eine genauere Beschreibung verzichtet wird.

Statischer Entwurf

- use case diagrams (OOSE)
- class diagrams (Booch, OMT, et. al.)

Dynamischer Entwurf

- state diagrams (David Harel [Harel 87])
- activity diagrams (Work-Flow)
- sequence diagrams (Interaktionsdiagramme von Booch)
- collaboration diagrams (Objektdiagramme von Booch)

Implementation

- component diagrams (Moduldiagramme von Booch)
- deployment diagrams (Prozessdiagramme von Booch)

Als echt neue Konzepte der UML können die nachfolgend zusammengefassten angesehen werden:

- stereotypes
- responsibilities
- extension mechanisms: stereotypes, tagged values, constraints
- threads and processes
- distribution and concurrency (ActiveX/DCOM and CORBA)
- patterns/collaborations
- activity diagrams
- Klare Trennung von Typen, Klassen und Instanzen
- Verfeinerungen
- Schnittstellen und Komponenten

Unterschiede zu Booch

Die UML ist keine Neuerfindung, sondern eine natürliche Weiterentwicklung z.B. der Methode von Booch. Das über die Jahre angeeignete Wissen und die Erfahrungen mit der Booch'schen Notation können weiter verwendet werden.
UML ist jedoch wesentlich ausdrucksstärker geworden. Es können nun Gegebenheiten abgebildet werden, die ehemals ausserhalb der erfassbaren Sachverhalte lagen. Die Notation in UML ist klarer und uniformer und bildet eine echte Obermenge der zugrunde liegenden Notationen.
Damit ist es ohne weiteres möglich, die Diagramme, die basierend auf der Booch'schen Methode erstellt worden sind, ohne Verlust von Information in die UML zu übertragen. Aus diesem Grund sei nachfolgend aufgezeigt, wie eine derartige Abbildung in Angriff genommen werden könnte (oder wie die Diagramme von Booch in die UML eingeflossen sind).
Klassendiagramme
sind in die „class diagrams" eingeflossen, die Notation ist aber stark an OMT angelehnt, insbesondere sind die Booch'schen Wolken verschwunden.
Zustandsübergangsdiagramme
sind schon in der Booch'schen Methode an die Entsprechungen von Harel [Harel 87] angelehnt. In der UML werden nun die exakten Harel-Diagramme verwendet und unter dem Namen „state diagrams" geführt.
Objektdiagramme
sind ist die „collaboration diagrams" übergegangen.
Interaktionsdiagramme
sind wesentlich erweitert worden (Rekursion, Beginn und Ende der

Lebenszeit von Objekten, aktive und inaktive Objekte, bedingte Verzweigungen) und heissen nun „sequence diagrams“.
Moduldiagramme
sind erweitert als „component diagrams“ verfügbar.
Prozessdiagramme
sind erweitert als „deployment diagrams“ aufgenommen worden.
Aus der obigen Gegenüberstellung kann man ablesen, dass sämtliche Diagramme von Booch ihre Entsprechung in UML gefunden haben. Sie sind dabei zum Teil direkt und zum Teil durch wesentliche Erweiterungen in UML übernommen worden.
Diese Abbildung lässt sich aber nicht ohne weiteres umkehren, da in UML Diagramme enthalten sind, die in Booch nicht vorkommen (z.B. use case diagrams).

Implikationen für DEIMOS

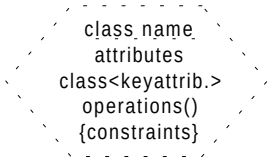
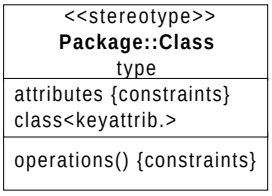
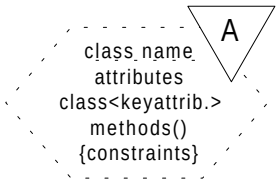
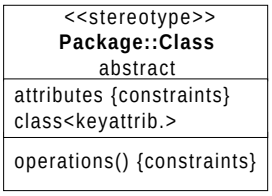
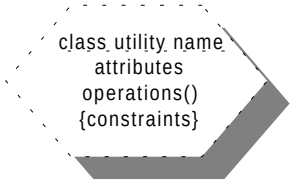
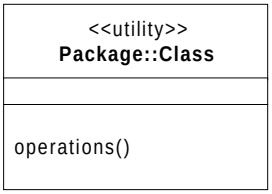
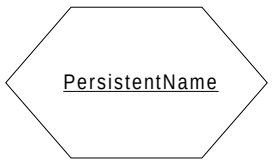
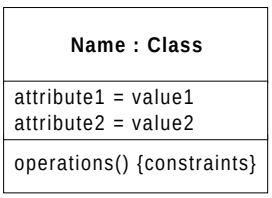
DEIMOS ist eine Erweiterung der Methode von Booch. Die Erweiterungen wurden aufgrund verschiedener Schwachstellen bei der Anwendungen für den konzeptionellen Datenbankentwurf eingeführt. Die Frage also lautet, wären die Schwachstellen nicht aufgetreten, wenn von der UML ausgegangen worden wäre?
Persistenz: Keines der Diagramme der UML - insbesondere auch das Klassendiagramm - enthält Konstrukte für die Unterscheidung zwischen persistenten und transienten Objekten. Die Schwachstelle wäre also gleichsam aufgetreten.
Transaktionen: Auch in der UML sind keine Konstrukte für den Transaktionsentwurf enthalten.
Physische Aspekte: Die im Gegensatz zu Booch’s Prozessdiagramme wesentlich erweiterten Verteilungsdiagramme erlauben einen verbesserten Entwurf der physischen Aspekte. Die Diagramme enthalten Konstrukte für die Modellierung von physischen Knoten, denen beispielsweise Cluster zugewiesen werden können.
Applikationen: Da in der UML nicht zwischen transienten und persistenten Instanzen unterschieden werden kann, sind auch im Bereich der Applikationen und Applikationsklassen Schwachstellen zu bemängeln. Mittels der neuen Konstrukte für die Darstellung von Packungen und Schnittstellendefinitionen hingegen, wird der Applikationsentwurf wesentlich besser unterstützt.
Inverse Beziehungen: Auch in den neuen Klassendiagrammen können keine „echten“ inversen Beziehungen abgebildet werden.
Extensionen und Schlüssel: In der UML stehen nach wie vor keine Konstrukte für die Definition von Schlüssel und den entsprechenden Extensionen zur Verfügung.
Fortpflanzung und Konsistenzsicherung: zwar werden in den neuen Klassenkonstrukten zusätzliche Möglichkeiten für die Formulierung von Zusicherungen angeboten (Bedingungen, welche zu Beginn und am Ende von Methodenausführungen erfüllt sein müssen), doch vermögen diese nicht alle Aspekte der Erhaltung der Datenintegrität abzudecken.
Objektevolution: Weder in den statischen noch in den dynamischen Diagrammen sind Konstrukte für die Migration von Instanzen vorgesehen. Auch diese Schwachstelle bleibt also vollständig erhalten.

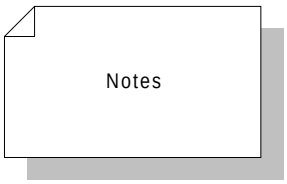
Zusammenfassend lässt sich also feststellen, dass auch in der UML die Aspekte des konzeptuellen Datenbankentwurfs ungenügend berücksichtigt worden sind. Die Schwachstellen treten also grösstenteils auch beim Einsatz der UML auf. Durch die neuen Möglichkeiten (Komponentendiagramme oder Verteilungsdiagramme) hätten sich diese jedoch vielerorts eleganter beheben lassen. Zudem hätte gegebenenfalls eine grössere Menge von Schwachstellen (physische Aspekte) in die weitere Untersuchung aufgenommen und behoben werden können.

Anpassungen an DEIMOS für die Konsistenz zur UML

Die Schwachstellen wären also grösstenteils auch aufgetreten, hätte man die Erweiterungen basierend auf der UML definiert. Würde man also von der UML ausgehen, würden sich die neu eingeführten Konstrukte vor allem optisch unterschieden.
In der nachfolgenden Gegenüberstellung wird eine derartige Abbildung von DEIMOS als Erweiterung von Booch und von DEIMOS als Erweiterung von UML dargestellt. Mit dieser einfachen Umsetzung könnte also eine Konsistenz zur UML erreicht werden.

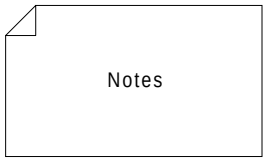
Schemadiagramme

Booch-basiert		**UML-basiert**
	Datenbankklassen Für die Darstellung der Datenbankklassen kann das Standardkonstrukt für die Klassen übernommen werden.	
	Abstrakte Klassen In UML werden für die Darstellung der abstrakten Klassen die Standardkonstrukte verwendet und mit dem Schlüsselwort „abstract“ versehen.	
	Hilfsklassen Die Hilfsklassen werden durch das Stereotyp „utility“ gekennzeichnet. Derartige Klassen besitzen keine Attribute, weshalb der mittlere Teil der Ikone frei bleibt.	
	Objekte Objekte und Klassen werden in UML nicht mehr streng getrennt. Entsprechend wird für die Darstellung ein Klassenkonstrukt verwendet, welches einen Namen hat und dessen Attribute Werte tragen.	



Notizen

Die Darstellung der Notizen hat sich nur geringfügig geändert. Der Schatten ist weggefallen und das „Eselsohr“ kann in einer beliebigen Ecke positioniert sein.

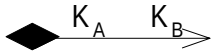


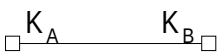
Vererbungsbeziehung

Der Vererbungspfeil ist neu mit einer etwas anderen Pfeilspitze dargestellt.

Komponentenbeziehung

Auf der Seite der Aggregationsklasse steht eine Raute und auf der Seite der Komponente ein Pfeil.





Inverse Beziehung

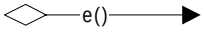
Dieses Konstrukt fehlt in der UML und muss demnach neu definiert werden. Aus Kompatibilitätsgründen wird es aus den Elementen der nicht existenzabhängigen Aggregation zusammengesetzt.

Referenzen

Für die Darstellung der Referenzen wird die Enthaltenseinsbeziehung verwendet.

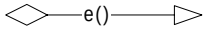
Instantiierungsbeziehung

Die Instantiierung ist neu mit einer etwas anderen Pfeilspitze dargestellt.



Objektevolution

Das Evolutionskonstrukt ist auch in UML nicht zu finden. Deshalb wird es kompatibel nachdefiniert.



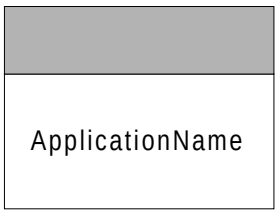
Notizbezug

Der Notizbezug ist in UML identisch.

Transaktionsdiagramme

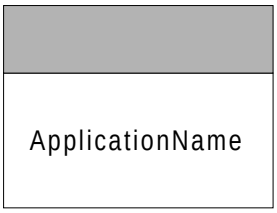
Booch-basiert

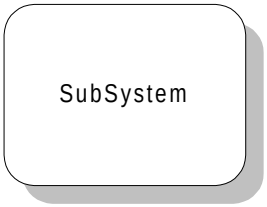
UML-basiert



Applikationen

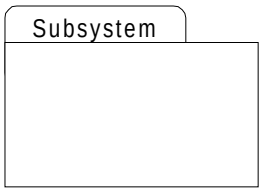
Das Applikationskonstrukt ist auch UML enthalten.

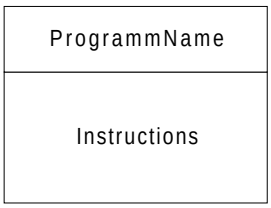




Subsysteme

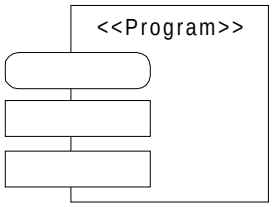
Subsysteme können in UML als sogenannte „Packungen“ dargestellt werden. In Packungen können beliebige Notationselemte enthalten sein. Sie dienen zur allgemeinen Strukturierung der Diagramme

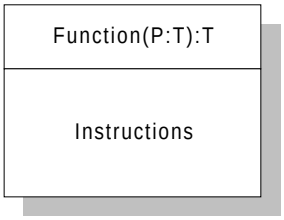




Programme

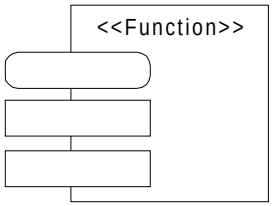
In UML wurde ein neuer Diagrammtyp, die Komponentendiagramme, eingeführt. Programme müssen dieser Semantik entsprechend als Komponenten betrachtet werden und werden demnach mit dem Komponentenkonstrukt dargestellt.

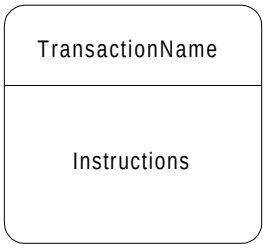




Funktionen

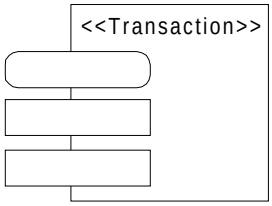
Auch Funktionen sind als Komponenten darzustellen. Sie werden durch das Stereotyp unterschieden.





Transaktionen

Die Transaktionen werden als Komponenten mit dem Stereotyp „Transaktion“ dargestellt.



Aufrufe

Abhängigkeiten zwischen den Komponenten werden mit dem ausgezogenen Pfeil dargestellt, weshalb dieses Konstrukt für die Aufrufe herangezogen werden soll. Beziehungen zwischen Packungen jedoch werden mit dem gestrichelten Pfeil definiert. Aus diesem Grund muss auch das entsprechende Konstrukt für die Darstellung der Aufrufe zugelassen sein.

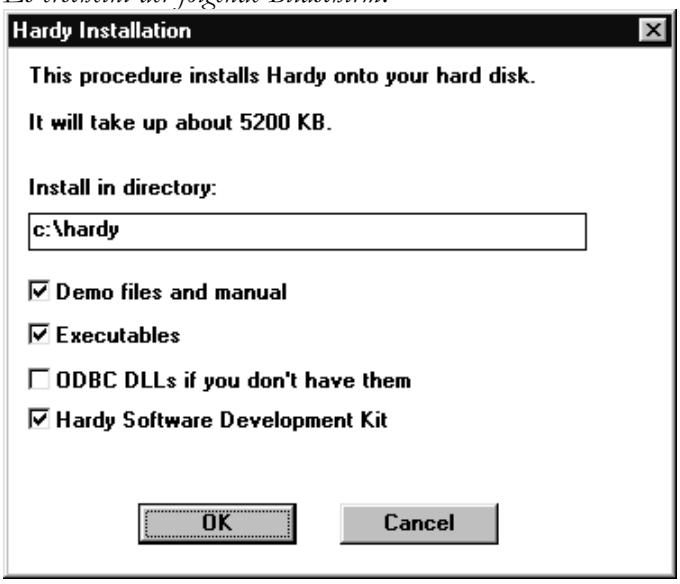
Installation

Installation von Hardy vom Internet

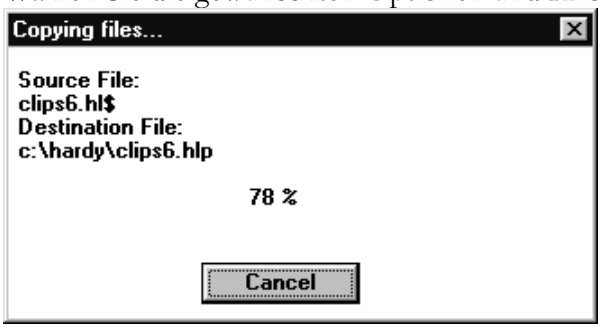
Die Entwickler des Zeichenwerkzeuges Hardy betreiben eine Seite auf dem Internet, welche über die Adresse http://rhum-ego.aiai.ed.ac.uk/~hardy abgerufen werden kann.
Auf dieser Seite ist neben den umfangreichen Informationen über Hardy auch eine „Download Area“ zu finden. Heruntergeladen werden kann insbesondere eine Demoversion von Hardy, Version 1.86 (Instruktionen zum Herunterladen sind ebenfalls auf dieser Seite zu finden).

Installation auf Windows 95

Nachdem Sie die Archivdatei vom Internet bezogen haben, extrahieren Sie es in einen speziellen temporären Ordner (z.B. C:\TEMP\INSTALL). Im Archiv ist ein Installationsprogramm INSTALL.EXE enthalten, das Sie durch einen Doppelklick starten können (vgl. [Hardy Installation 95]).
Es erscheint der folgende Bildschirm:

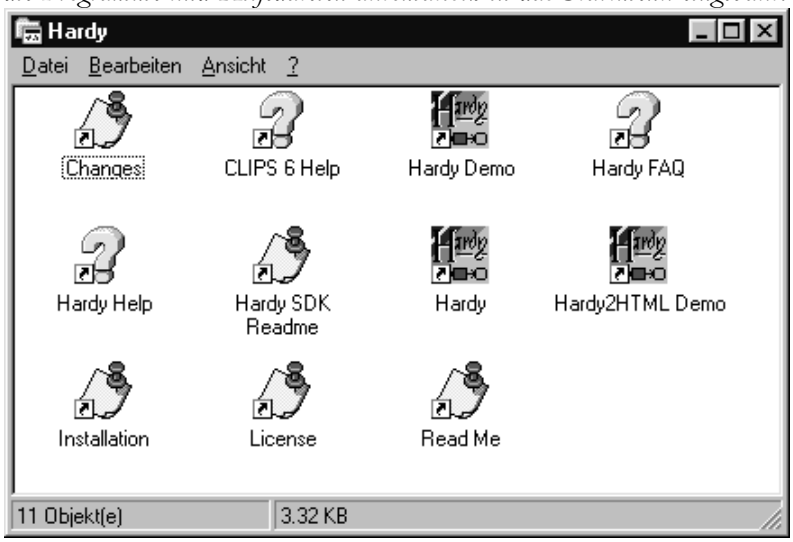


Achten Sie darauf, dass Hardy im Verzeichnis C:\HARDY installiert wird, damit das spätere Zusammenspiel von Hardy und OSWOOD einwandfrei unterstützt werden kann.
Wählen Sie die gewünschten Optionen und Klicken Sie auf OK



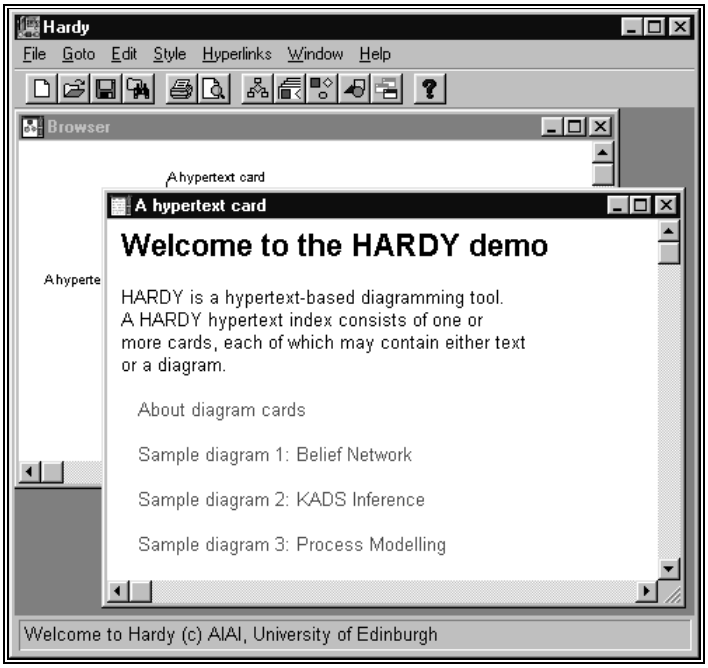
Die benötigten Dateien werden kopiert

Anschliessend wird eine DOS-Shell für das Extrahieren der Binärdateien und der Hardy-SDK-Dateien geöffnet. Danach werden die Einträge der Links auf die Programme und Hilfedateien automatisch in das Startmenü eingebaut.



Am Ende des Kopierprozesses, der ca. 10 Sekunden dauert, kann Hardy direkt gestartet werden.

Hardy wird, falls die obige Frage mit ja beantwortet wurde, mit einem mitgelieferten Einführungsbeispiel gestartet.



Zu Schluss wird der Erfolg der Installation über die folgende Meldung angezeigt:

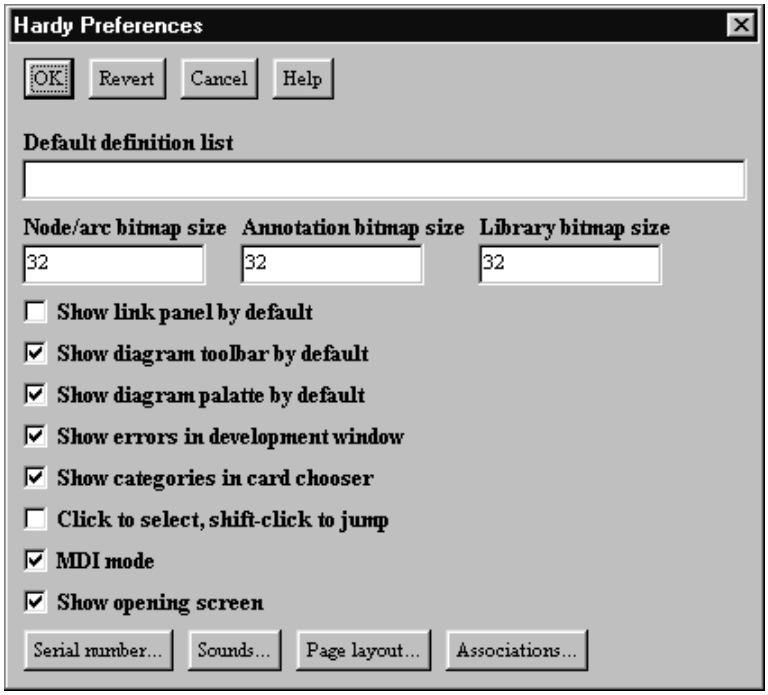
Hardy ist nun vollständig installiert und kann gemäss Beschreibung (vgl. “Starten von Hardy”, Seite 127) gestartet werden.

Registrierung

Beachten Sie bitte, dass Hardy lizenzpflichtig ist. Die von Internet bezogene Version ist lediglich eine Demoversion. Möchten Sie die Vollversion betreiben, schicken Sie den Anbietern ein E-Mail gemäss Anleitung auf der Internetseite. Haben Sie eine korrekte Lizenznummer erhalten, können Sie diese wie folgt im Hardy eingeben:

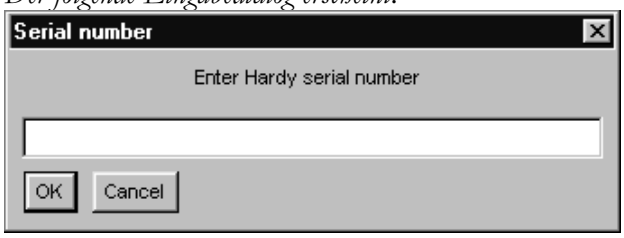
Wählen Sie aus dem Menü im Hauptfenster von Hardy den Punkt **Tools | Preferences**

Der folgende Dialog erscheint:



Klicken Sie auf die Schaltfläche „Serial number...“

Der folgende Eingabedialog erscheint:



Geben Sie ihre Seriennummer ein und bestätigen Sie die Eingabe mit „OK“.

Sie können nun Hardy ohne Einschränkungen betreiben.

Installation von Hardy mit der beiliegenden Diskette

Auf der beiliegenden Diskette „Hardy for OSWOOD, Version 1.86“ ist eine minimale Installation von Hardy vorbereitet. Gegenüber der Vollversion fehlen die Hilfedateien und das Hardy-SDK.

Bei der Installation soll nach den folgenden Schritten vorgegangen werden:

1. Erstellen Sie ein Verzeichnis C:\HARDY
2. Kopieren Sie die Datei HARDY.ZIP in das Verzeichnis C:\HARDY
3. Kopieren Sie die Datei PKUNZIP.EXE in das Verzeichnis C:\HARDY
4. Öffnen sie eine DOS-Shell (CMD.EXE oder COMMAND.EXE)
5. Wechseln Sie in das Verzeichnis C:\HARDY ("cd \hardy")
6. Expandieren Sie das Archiv HARDY.ZIP ("pkunzip hardy.zip")
7. Schliessen Sie die DOS-Shell
8. Löschen Sie aus dem Verzeichnis C:\HARDY die Datei HARDY.ZIP
9. Löschen Sie aus dem Verzeichnis C:\HARDY die Datei PKUNZIP.EXE

Beachten Sie bitte, dass diese Version lediglich eine Demoversion ist (Lizenzierung siehe oben).

Installation von OSWOOD

Voraussetzungen für den fehlerfreien Betrieb von OSWOOD sind die Installation

- von Hardy,
- der Symbolbibliotheken und
- von OSWOOD.

Der vorliegenden Arbeit beigelegt sind zwei Disketten mit den für den Minimalbetrieb notwendigen Dateien.

Die Symbolbibliotheken von DEIMOS

Die in der Methode DEIMOS verwendeten Symbolbibliotheken sind in den folgenden Dateien gespeichert:

```
DIAGRAMS.DEF
basic.slb
ooadarcs.slb
oonodes.slb
symbols.slb
DEIMOS.slb
BCLOBJEC.DEF
BINTER.DEF
BMODULE.DEF
BPROCESS.DEF
BTRANS.DEF
SCHEMA.DEF
TRANS.DEF
CONTEXT.DEF
```

Listing 7-1: Die Dateien mit den Definitionen der Symbole und Diagramme

Wenn Sie die Installation von Hardy mit der beiliegenden Diskette erledigt haben, verfügen Sie bereits über die notwendigen Symbolbibliotheken und Diagrammdefinitionsdateien. Haben Sie jedoch Hardy direkt vom Internet bezogen, müssen Sie die nachfolgenden Schritte unternehmen, damit die Diagramme von DEIMOS in Hardy unterstützt werden können:

1. Erstellen Sie ein temporäres Verzeichnis (z.B. „C:\TEMP“)
2. Kopieren Sie die Dateien HARDY.ZIP und PKUNZIP.EXE der Diskette „Hardy for OSWOOD“ in dieses Verzeichnis
3. Öffnen sie eine DOS-Shell (CMD.EXE oder COMMAND.EXE)
4. Wechseln Sie in das Verzeichnis C:\HARDY ("cd \hardy")
5. Expandieren Sie das Archiv HARDY.ZIP ("pkunzip hardy.zip")
6. Schliessen Sie die DOS-Shell
7. Kopieren Sie die Dateien der obigen Dateiliste (Listing 7-1) in das Verzeichnis C:\HARDY
8. Löschen Sie das temporäre Verzeichnis

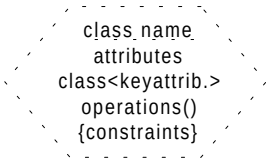
Installation von OSWOOD

Das Werkzeug OSWOOD besteht lediglich aus einer Anwendungsdatei, welche Sie auf der Diskette „OSWOOD, Version 1.0“ finden. Kopieren Sie diese in ein Verzeichnis ihrer Wahl.

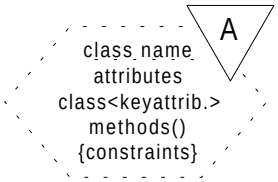
Notation

Schemadiagramm

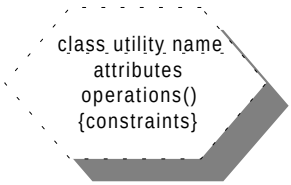
Knoten



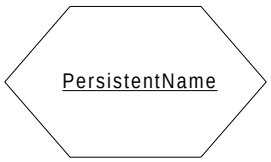
Datenbankklassen



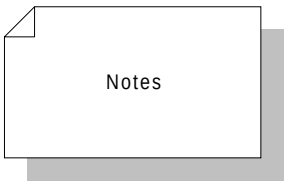
Abstrakte Klassen



Hilfsklassen

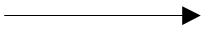


Objekte



Notizen

Kanten



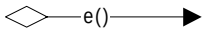
Vererbungsbeziehung

Komponentenbeziehung

Inverse Beziehung

Referenzen

Instantiierungsbeziehung

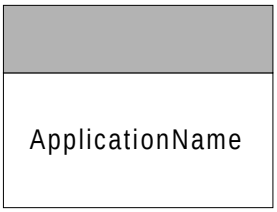


Objektevolution

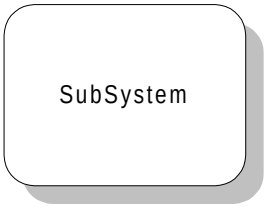
Notizbezug

Transaktionsdiagramm

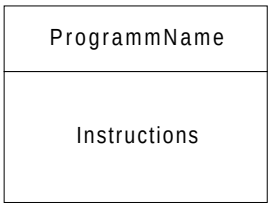
Knoten



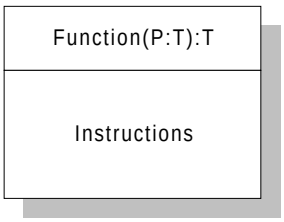
Applikationen



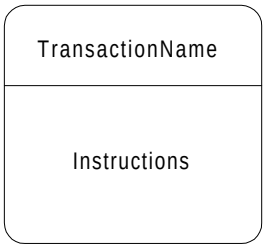
Subsysteme



Programme



Funktionen



Transaktionen

Kanten

Aufrufe

Glossar

Abstrakte Klassen

Klassen, von denen keine Instanzen gebildet werden dürfen, werden abstrakte Klassen genannt. Dienen meist für die Beschreibung des gemeinsamen abstrakten Verhaltens verschiedener Basisklassen in Superklassen. Diese Klassen finden in der Realwelt häufig keine Entsprechungen (beschreiben keine realen Objekte); sie sind abstrakt.

Assoziation

Allgemeine Beziehung zwischen zwei Objekten. Wird vor allem im Grobentwurf definiert und im Feinentwurf in eine Vererbungs-, Aggregations- oder Verwendungsbeziehung umgesetzt.

Extension

Menge aller zu einem gewissen Zeitpunkt existierenden Instanzen einer Klasse. (vgl. Intension)

Friend-Klassen

Klassen, welchen explizit spezielle Zugriffsrechte eingeräumt wurden, werden als Friend-Klassen bezeichnet. Üblicherweise werden derartigen Klassen Zugriffe auf private Attribute oder Methoden erlaubt.

IDL

Interface Definition Language. Von der OMG definierte Schnittstellendefinitionssprache für Objekte.

Intension

Menge aller denkbaren Instanzen einer Klasse. (vgl. Extension)

Inverse Beziehung

Rückbezügliche Beziehung. Ein Objekt A unterhält eine inverse Beziehung zu einem Objekt B, genau dann, wenn das Objekt B eine inverse Beziehung zum Objekt A unterhält.

Klasse

gemeinsame Beschreibung aller denkbaren, syntaktisch überhaupt möglichen Objekte einer bestimmten Charakteristik [Dittrich et. al. 95]

Nesting-Klassen

Klassen, die in einer Klasse eingebettet und demnach von ausserhalb nicht mehr sichtbar sind, werden als Nesting-Klassen bezeichnet.

MFC

Microsoft Foundation Classes. C++ Klassenbibliothek der Firma Microsoft Coorperation, Bestandteil der Entwicklungsumgebung Visual C++

Objektevolution

Steht für den Wechsel der Klassenzugehörigkeit einer Instanz.

ODL

Object Definition Language. Objektdefinitionssprache. Bestandteil des Standards der ODMG-93. Pendant zur Datendefinitionssprache DDL in relationalen Systemen.

OMG

Object Management Group. Gremium für die Standardisierungen im Bereich der objektorientierten Systeme. Prüft zur Zeit die Eignung der UML als Standard für den objektorientierten Entwurf.

OML

Object Manipulation Language. Manipulationssprache für Objektdaten. Ist kein Bestandteil des Standards der ODMG-93, sondern wird innerhalb einer bestimmten PL-ODL definiert. Pendant zur Manipulationssprache DML in relationalen Systemen.

ODMG

Object Data Management Group. Gremium für die Standardisierungen im Bereich der objektorientierten Datenbanken. Erliess im Jahre 1993 den Standard „ODMG-93“ für objektorientierte Datenbanksysteme (vgl. [Cattel 96]).

OQL

Object Query Language. Objektabfragesprache. Bestandteil des Standards der ODMG-93. Pendant zur Abfragesprache SQL in relationalen Systemen.

PL-ODL

Programming Language - ODL. Spezifische Umsetzung der Anbindung der Objektdefinitionssprache für eine bestimmte Programmiersprache.

Schemaevolution

Beschreibt die (Weiter-) Entwicklung eines Datenbankschemas (Hinzufügen/Entfernen/Verändern von Klassen, Applikationen etc.). Das Hinzufügen/Entfernen/Verändern von Objekten (Objektdaten) wird nicht als Schemaevolution angesehen.

Sicht

In relationalen Datenbanksystemen: Spezielle Ansicht einer oder mehrerer Relationen. Im Allgemeinen durch eine SELECT-Anweisung spezifiziert. Im Umfeld von objektorientierten Datenbanksystemen existiert keine Definition. Verwandter Begriff: „View“

Subsystem

Gliederungselement eines Problembereiches

Surrogat

Objekt Identifikator -> Object ID (OID)

Taxionomie

Aus der Biologie stammender Begriff für die Klassierung von Verhaltensmustern, in der objektorientierten Welt häufig durch den Begriff Vererbung ersetzt

UM

Unified Method. Erste Version der vereinheitlichten Entwurfsmethode von Grady Booch und Jim Rumbaugh (vgl. [Booch et. al. 95]).

UML (1)

Unified Method Language. Erste Version der vereinheitlichten Entwurfsmethode von Grady Booch, Jim Rumbaugh und Ivar Jacobsen (vgl. [Booch et. al. 96]).

UML (2)

Unified Modeling Language. Version 1.0 der vereinheitlichten Entwurfsmethode von Grady Booch, Jim Rumbaugh und Ivar Jacobsen (vgl. [Booch et. al. 97]).

Literaturverzeichnis

Referenzierte

Literatur

[Aksit et. al. 92] Aksit, Mehmet & Bergmans, Lodewijk, 'Obstacles in Obcjet-Oriented Software Development', in: *OOPSLA '92*, 1992, pp. 341-358.

[Atkinson et. al. 89] Atkinson, M., Bancilhon, F., DeWitt, D.J., Dittrich, K.R., Maier, D. & Zdonik, S.B., *The Object-Oriented Database Manifesto*, Kyoto, Japan, 1989.

[Booch 94] Booch, Grady, *Object-Oriented Analysis and Design with Applications*, 2nd Edition, Redwood City, California: The Benjamin/Cummings Publishing Company Inc., 1994, ISBN: 0-8053-5340-2.

[Booch et. al. 95] Booch, Grady & Rumbaugh, Jim, *Unified Method for Object-Oriented Development UM*, Documentation Set 0.8, Santa Clara, California: Rational Software Corp., 1997, URL: http://www.rational.com

[Booch et. al. 96] Booch, Grady, Jacobsen, Ivar & Rumbaugh, Jim, *Unified Method Language for Object-Oriented Development UML*, Documentation Set 0.9 Addendum, Santa Clara, California: Rational Software Corp., 1997, URL: http://www.rational.com.

[Booch et. al. 97] Booch, Grady, Jacobsen, Ivar & Rumbaugh, Jim, *Unified Modeling Language UML*, Version 1.0, Santa Clara, California: Rational Software Corp., 1997, URL: http://www.rational.com.

[Brodie 89] Brodie, M.L., 'Future intelligent information systems - AI and database technologies working together', in: Brodie, M.L. & Mylopoulos, J. (Ed.), *Artificial intelligence and databases*, San Francisco, California: Morgan Kaufmann Publishers, 1989.

[Cattel 96] Cattel, R.G.G., *The Object Database Standard: ODMG-93*, Release 1.2, San Francisco, California: Morgan Kaufmann Publishers, 1996.

[Champeaux 91] de Champeaux, D., 'Object-Oriented Analysis an Top Down Software Development', in: *European Conference on Object-Oriented Programming*, 1991, pp. 360-375.

[Coad et. al. 91a] Coad, P & Yourdon, E., *Object-Oriented Analysis*, 2nd Edition, Yourdon Press Computing Series, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Coad et. al. 91b] Coad, P & Yourdon, E., *Object-Oriented Design*, Yourdon Press Computing Series, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[Dittrich et. al. 95] Dittrich, Klaus R. & Geppert, Andreas, 'Objektorientierte Datenbanksysteme - Stand der Technik', in: *HMD*, 1995, 183, pp. 8-23.

[Foot et. al. 88] Foot, B. & Johnson, R., 'Designing Classes', in: *Journal of Object-Oriented Programming*, 1988, June/July, pp. 23-35.

[Geppert 96] Geppert, Andreas, *Objektorientierte Datenbanksystene: Ein Praktikum*, Heidelberg: dpunkt, Verlag für digitale Technologie GmbH, 1996, ISBN: 3-920993-62-4.

[Gingrich et. al. 88] Gingrich, P. & Kleyn, M., 'GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views', in: *SIGPLAN Notices*, 1988, 23(11), pp. 192.
[Grotehen 95] Grothen, Thomas, 'Auf dem Weg zum Standard mit ODMG-93', in: *HMD*, 1995, 183, pp. 41-57.
[Guilfoyle et. al. 91] Guilfoyle, C. & Jeffocate, J., *Databases for objects - the market opportunity*, London: Ovum Ltd., 1991.
[Harel 87] Harel, D., 'Statecharts: A Visual Formalism for Complex Systems', in: *Science of Computer Programming*, 1987, 8.
[Heuer 93] Heuer, Andreas, 'Objektoerientierter Datenbankentwurf', in: *Informatik Spektrum*, 1993, 16, pp. 96-97.
[Holland et. al. 89] Holland, I. & Lieberherr, K., 'Assuring Good Style for Object-Oriented Programs', in: *IEEE Software*, 1989, September, pp. 38-48.
[Humphrey 89] Humphrey, W., *Managing the Software Process*, Reading, MA: Addison-Wesley, 1989.
[Jacobsen 92] Jacobsen, Ivar, *Object-Oriented Software Engineering: A Use Case Driver Approach*, Workingham: Addison-Wesley, 1992.
[Johnson et. al. 90] Johnson, R. & Wirfs-Brock, R. , 'Surveying Current Research in Object-Oriented Design', in: *Communications of the ACM*, 1990, Vol. 33, No. 9, pp. 104-124.
[Kappel et. al. 91a] Kappel, G. & Scherfl, M., 'Using an Object-Oriented Diagram Technique for the Design of Information Systems', in: Sol, H.G. & van Hee, K.M. (Ed.), *Dynamic Modelling of Informations Systems*, North-Holland: Elsevier Science Publishers B.V., 1991, pp. 121-164.
[Kappel et. al. 91b] Kappel, G. & Scherfl, M., 'Object/Behavior Diagrams', in: **, pp. 530-538.
[Lieberherr et. al. 91] Lieberherr, K. et. al., 'Graph-Based Software Engineering: Concise Specifications of Cooperative Behavior', in: *Hortheastern University, Tech. Report: NU-CCS-91-14*, 1991.
[Meier et. al. 95] Meier, Andreas & Wüst, Thomas, 'Objektorientierte Dantebanksysteme - Ein Produktvergleich', in: *HMD*, 1995, 183, pp. 24-40.
[Mellor 88] Mellor, S. & Shlaer, S., *Object-Oriented Systems Analysis Modling the World in Data*, Youtdon Press, 1988.
[Oestereich 97] Oestereich, B., *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Method Language*, 2., aktualisierte Auflage, München: R. Oldenbourg Verlag, 1997, ISBN: 3-486-23999-6.
[Peterich 58] Peterich, Eckart, *Götter und Helden der Griechen*, Frankfurt a.M.: Fischer Bücherei, 1958.
[Rumbaugh et. al. 91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W., *Object-Oriented Modeling and Design*, Englewood Cliffs, NJ: Prentice Hall, 1991.
[Shear 88] Shear, D., 'CASE Shows Promise, but Confusion Still Exists', in: *EDN*, 1988, 33(25), pp. 168.
[Whitehead 58] Whitehead, A., *An Introduction to Mathematics*, New York, NY: Oxford University Press, 1958.
[Wirfs-Brock 90] Wirfs-Brock, R., *Responsability Driven Approach*, Englewood Cliffs, NJ: Prentice Hall, 1990.

Dokumentationen

[Hardy Installation 95] *Hardy Installation Instructions*, Artificial Intelligence Applications Institute, University of Edinburgh, Edinburgh, UK, April 1995, URL: http://rhum-ego.aiai.ed.ac.uk/~hardy.

[Hardy User 96] *Hardy: User Guide*, Version 1.3, Smart, Julian & Rae, Robert, Artificial Intelligence Applications Institute, University of Edinburgh, Edinburgh, UK, Februar 1996, URL: http://rhum-ego.aiai.ed.ac.uk/~hardy.

[O_2Admin 95] *The O_2 System Administration Guide*, Version 4.5, O_2 Technology, Versailles, Frankreich, April 1995.

[O_2Tools 95] *O_2 Tools User Manual*, Version 4.5, O_2 Technology, Versailles, Frankreich, April 1995.

[UML Summary 97] UML Summary

[UML Notation 97] UML Notation Guide

[UML Semantics 97] UML Semantics

[UML Extensions 97] UML Process-Specific Extensions

Weitere

Literatur

Cattell, R.G.G., *Object Data Management*, Menlo-Park, California: Addison-Wesley Publishing Company, 1994, ISBN: 0-201-54748-1.

Cattel, R.G.G., *The Object Database Standard: ODMG-93*, Release 1.2, San Francisco, California: Morgan Kaufmann Publishers, 1996.

Dittrich, Klaus R., 'Object-Oriented Data Model Concepts', in: *Computer and System Sciences*, Berlin: Springer-Verlag, 1994, 130.

Dittrich, Klaus R. & Geppert, Andreas, 'Spcification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming by Contract', in: *GI-Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, Dresden, 1994.

Dittrich, Klaus R. & Geppert, Andreas, 'Objektstrukturen in Datenbanksystemen oder: Auf der Suche nach voller Objektorientierung', in: ***, ***, pp.421-429.

Fischer, W.E., Küspert, K. & Puppe, F., 'Objektorientierte Datenbanksysteme', in: *Informatik Spektrum*, 1993, 16, pp. 67-68.

Huges, J.G, Objektorientierte Datenbanken, München/Englewood Cliffs: Hanser/Prentice Hall, 1992, ISBN: 3-446-16583-5.

Kemper, A. & Moerkotte, G., 'Basiskonzepte objektorientierter Datenbanksysteme', in: *Informatik Spektrum*, 1993, 16, pp. 69-80.

Matytschak, Mirko, 'UML - Eine Notation für das Software-Design', in: *Microsoft System Journal*, 1997, 4, pp. 112-117.

Meyer, B., *Object-Oriented Software Construction*, Englewood Cliffs, NJ: Prentice Hall, 1988.

Park, E.K. & Song, Il-Yeol, 'Object-Oriented Database Design Methodologies: A Survey', in: **, **, pp. 115-142.

Schmidt, D., *Persistente Objekte und objektorientierte Datenbanken: Konzepte, Architekturen, Implementierung und Anwendung*, München: Hanser, 1991, ISBN: 3-446-16411-1.

Wüst, Thomas, 'OODBMS-Einsatz bei der CSS Versicherung', in: *Informatik/Informatique*, 1996, 2, pp. 3-7.

Dokumentationen

O_2 C Reference Manual, O_2 Technology, Versailles, Frankreich, März 1995.
O_2 Kit, Version 4.5, O_2 Technology, Versailles, Frankreich, April 1995.
O_2 Look, Version 4.5, O_2 Technology, Versailles, Frankreich, April 1995.
The O_2 User Manual, Version 4.4, O_2 Technology, Versailles, Frankreich, Dezember 1993.

Verzeichnisse

Abbildungen

Tabellen

Listings

Syntaxdiagramme

Klassendiagramme

Klassenbeschreibungen

Index