Migration einer relationalen Datenbank in eine objektorientierte Datenbank

Diplomarbeit von Christina Kipp

Universität Osnabrück

16. Oktober 1996

Vorwort

Die vorliegende Diplomarbeit wurde für den schriftlichen Teil der Diplomprüfung im Studiengang Mathematik im Fachgebiet Informatik an der Universität Osnabrück erstellt.

Es handelt sich hierbei um eine praxisorientierte Arbeit, die in Zusammenarbeit der mit Firma *bcas* (Beratung für Computer-Anwendungen und Software Rolfes und Vogel GmbH, Osnabrück) und der Universität Osnabrück angefertigt wurde. Die Betreuung wurde an der Universität von Prof. Dr. Oliver Vornberger und Dipl.-Math. Frank M. Thiesing, bei *bcas* von Dr. Heiko Pape übernommen.

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich beim Erstellen dieser Arbeit unterstützt haben.

Insbesondere danke ich Herrn Prof. Dr. Oliver Vornberger für eine Reihe von Anregungen und Vorschlägen zum Inhalt und zur Ausführung der Arbeit.

Ebenso bedanke ich mich bei Herrn Dipl.-Math Frank M. Thiesing für eine intensive Betreuung und viele nützliche Tips bei der Entwicklung der Arbeit.

Ferner möchte ich mich bei Herrn Dr. Heiko Pape bedanken, ohne dessen Initiative diese Diplomarbeit nicht zustandegekommen wäre. Ebenfalls danke ich der Firma *bcas*, die mir ermöglichte, die Idee für diese Diplomarbeit in die Tat umzusetzen.

Bei Doris Schumacher und Friedrich Kipp bedanke ich mich für das Korrekturlesen.

Erklärung

Ich versichere, die vorliegende Arbeit selbständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Osnabrück, den 16. Oktober 1996

Christina Kipp

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

> *I have a cat named Trash. In the current political climate, it would seem that if I were trying to sell him (at least to a computer scientist), I would not stress that he is gentle to humans and is self-sufficient, living mostly on field mice. Rather, I would argue that he is object-oriented.*
>
> *R.King,1989,[Kin89]*

Der Begriff 'Objektorientierung' gehört zu *den* Schlagwörtern innerhalb der Informatik. Objektorientierte Konzepte stellen ein besseres Werkzeug zur Bewältigung einiger Problemstellungen dar als herkömmliche Konzepte. Im Bereich der Datenbanken besteht das Hauptanliegen darin, Anwendungsdaten umfassender und exakter in der Datenbank darstellen zu können, als dies beispielsweise mit dem relationalen Datenmodell möglich ist [Dit90]. Die Entwicklung der objektorientierten Konzepte vollzog sich nacheinander in verschiedenen Teilgebieten der Informatik. Die ersten objektorientierten Konzepte wurden im Bereich der Programmiersprachen entwickelt. Die Entwicklung objektorientierter Datenbanksysteme begann Mitte der 80er Jahre.

Der Einsatz von Datenbanksystemen zur Lösung betriebswirtschaftlich-administrativer Problemstellungen hat sich in den letzten Jahren bewährt und am Markt durchgesetzt [Beh91]. Dabei hat das relationale Datenmodell von Codd das hierarchische und das Netzwerkmodell in der Datenbanktheorie und -entwicklung als Standard abgelöst [Heu92]. Allerdings beinhaltet das RDM (Relationale Datenmodell) eine Reihe von Problemen, die insbesondere den Datenbankentwurf und die Modellierung der Anwendungsdaten betreffen. Ein Kritikpunkt ist die Schwierigkeit der Darstellung komplexer Datenobjekte, der in Anwendungsbereichen wie CAD (Computer Aided Design), CIM (Computer Integrated Manufacturing) und CASE (Computer Aided Software Engineering) besonders gravierend ist. Außerdem wird häufig die geringe Information über die inhaltliche Bedeutung und Anwendung der Daten (Semantik) kritisiert. Diese beiden Hauptkritikpunkte und einige weitere Schwächen des RDM führten zur Weiterentwicklung in semantische und objektorientierte Datenmodelle.

Es existieren bereits einige objektorientierte Datenbanksysteme (ooDBS), die zur Zeit jedoch noch kaum in kommerziellen Bereichen eingesetzt werden. Ihre Einsatzgebiete sind vielmehr in den Bereichen der Benutzeroberflächen, Multimedia-Anwendungen und des Systems Engineering zu finden. Die Möglichkeit, in objektorientierten Datenmodellen (ooDM) komplexe Objekte als eine Einheit darzustellen, ist in diesen Anwendungen von besonderer Bedeutung. Aber auch in anderen Bereichen ist durchaus ein Anwendungspotential vorhanden.

Da sich ein Großteil der Probleme, die im Zusammenhang mit dem relationalen Modell auftreten, durch die Einführung eines objektorientierten Datenmodells beheben läßt, werden sich objektorientierte Datenbanken in Zukunft mehr und mehr durchsetzen. Für Firmen, die eine solche Datenbank einführen wollen, wird es daher nötig sein, Daten, die sich bis zu dem Zeitpunkt in einer relationalen Datenbank befanden, in die objektorientierte Datenbank zu übernehmen.

In der vorliegenden Arbeit wird ein Verfahren entwickelt, mit dessen Hilfe Daten aus einer relationalen Datenbank in ein objektorientiertes Datenbanksystem überführt werden können. In Kapitel 2 werden zunächst die Aufgabenstellung der Arbeit, sowie das Migrationsumfeld und der Ablauf der Migration präzisiert.

Die Migration soll am Beispiel einer realen Datenbank durchgeführt werden. Hierfür wurde eine Datenbank ausgewählt, die einem sogenannten PPS-System zugrunde liegt. In Kapitel 3 wird diese Datenbank genauer spezifiziert.

Kapitel 4 beschreibt das Generieren von Tabellenklassen, in denen die Daten der relationalen Datenbank zwischengespeichert werden.

Bei der Definition der Klassen der objektorientierten Datenbank geht man nach bestimmten Regeln vor, damit das Übertragen der Daten aus den Tabellenobjekten in die Objekte dieser Klassen automatisch von statten gehen kann. Diese Regeln werden in Kapitel 5 genau beschrieben.

Kapitel 6 beschäftigt sich mit dem Erzeugen der Objekte und dem automatischen Übertragen der Daten aus den Tabellenobjekten. Außerdem wird das Herstellen der zwischen den Objekten bestehenden Beziehungen beschrieben.

Im Rahmen der Arbeit sind drei O_2-Applikationen entstanden, die zum Migrieren und zum Anzeigen lassen der in der Datenbank vorhandenen Daten genutzt werden können. Diese Applikationen werden in Kapitel 7 beschrieben.

In Kapitel 8 werden einige Vor- und Nachteile des Migrationskonzeptes diskutiert und einige weiterführende Überlegungen angestellt.

Die Arbeit schließt mit einer Zusammenfassung in Kapitel 9.

Kapitel 2

Aufgabenstellung

2.1 Spezifikation

Die vorliegende Arbeit beschäftigt sich mit der Migration einer relationalen Datenbank in eine objektorientierte Datenbank. Diese Migration soll am Beispiel eines „realen“ Systems durchgeführt werden.

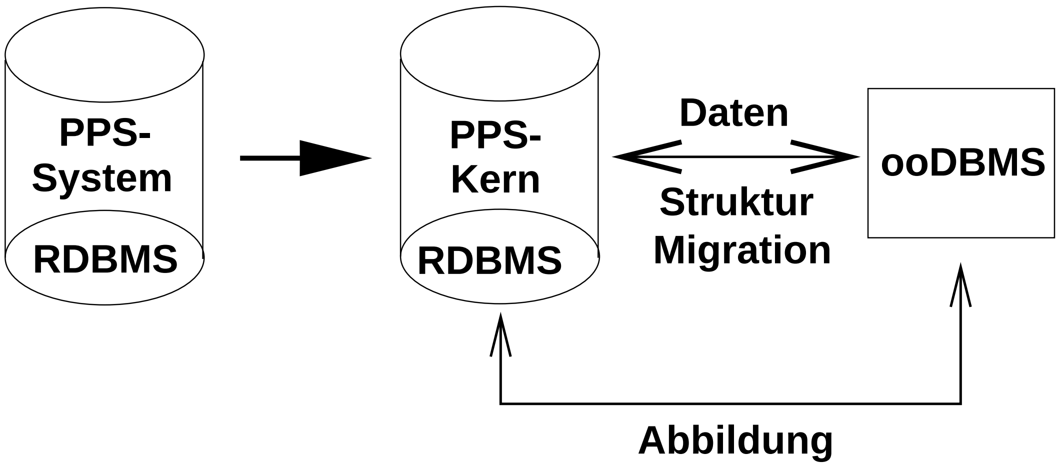

Abbildung 2.1: Die Aufgabenstellung

Als „reales“ System ist ein PPS-System (Produktionsplanungs- und Steuerungssystem) ausgewählt worden, das auf einer relationalen Datenbank basiert. Da die zu dem PPS-System gehörige Datenbank sehr umfangreich ist, wird zunächst der Teil extrahiert, der die Essenz eines PPS-Systems bildet. Dieser „Kern“ des PPS-Systems wird anschließend objektorientiert modelliert und implementiert. Bei einer Umstellung von einem relationalen Datenbankmanagementsystem (RDBMS) auf ein objektorientiertes ist es erforderlich, die Daten, die sich bis zu diesem Zeitpunkt in dem RDBMS befanden, neu zu strukturieren und diese in dem objektorientierten Datenbankmanagementsystem (ooDBMS) abzulegen. Dieser Vorgang wird als *Migration* bezeichnet. Um zu vermeiden, daß die Daten, die sich in der relationalen Datenbank befinden, alle neu eingegeben werden müssen, soll ein weitgehend automatisches Verfahren entwickelt werden, das es ermöglicht, die Daten aus dem RDBMS zu extrahieren und sie in das ooDBMS einzufügen.

Bevor eine Übertragung der Daten aus der relationalen Datenbank in die objektorientierte Datenbank stattfinden kann, muß das relationale Datenbankschema in ein objektorientiertes überführt werden. Ein Ziel der Diplomarbeit ist es, dies halbautomatisch durchzuführen, d.h., bei der Erzeugung der Klassen nach bestimmten Regeln vorzugehen. Die anschließende Migration der Daten in die ooDB soll vollautomatisch stattfinden. Als relationales Datenbankmanagementsystem wird hier *Informix* verwendet.

Bei der Übertragung der Daten wird hier keine Online-Verbindung aufgebaut, sondern die Daten werden in Dateien zwischengespeichert. Eine Online-Verbindung ist in diesem Fall nicht erforderlich, da hier davon ausgegangen wird, daß die Migration nur einmal stattfindet und anschließend mit dem objektorientierten Datenbankmanagementsystem weitergearbeitet wird.

Für den auf der objektorientierten Datenbank entstandenen Kern des PPS-Systems soll ein User-Interface geschaffen werden, welches dem Benutzer die vernetzte Struktur der Objekte visualisiert.

2.2 Migrationsumfeld

Wie Abbildung 2.2 zeigt, gibt es zwei verschiedene Systemumfelder, die räumlich voneinander getrennt sind. D.h., zwischen diesen beiden Systemumfeldern besteht keine Netzverbindung.

Die Datenbank, die die Grundlage des PPS-Systems darstellt, befindet sich auf einer IBM RS6000. IBM verwendet als Betriebssytem AIX. Als relationales Datenbankmanagementsystem ist auf dieser Maschine *Informix* 6.0 installiert. *Informix* verwaltet mehrere Datenbanken, die die Basis für das PPS-System bilden. Aus einer dieser Datenbanken wurden die Daten extrahiert und wieder in eine *Informix*-Datenbank auf demselben Rechner integriert. Diese neu entstandene Datenbank, der das Entity-Relationship-Diagramm zugrunde liegt, bildet die Essenz des PPS-Systems ab.

Die Tabellenstruktur der Datenbank ist in dem ER-Diagramm festgehalten. Die SQL-Befehle zur Erstellung der Tabellen in *Informix* werden mit Hilfe eines Skripts in eine Datei geschrieben, die in *Informix* gelesen werden kann. Nachdem die Tabellen mit Hilfe dieser Datei entstanden sind, müssen diese mit Daten aus der „realen“ Datenbank gefüllt werden. Dazu werden mit Hilfe des

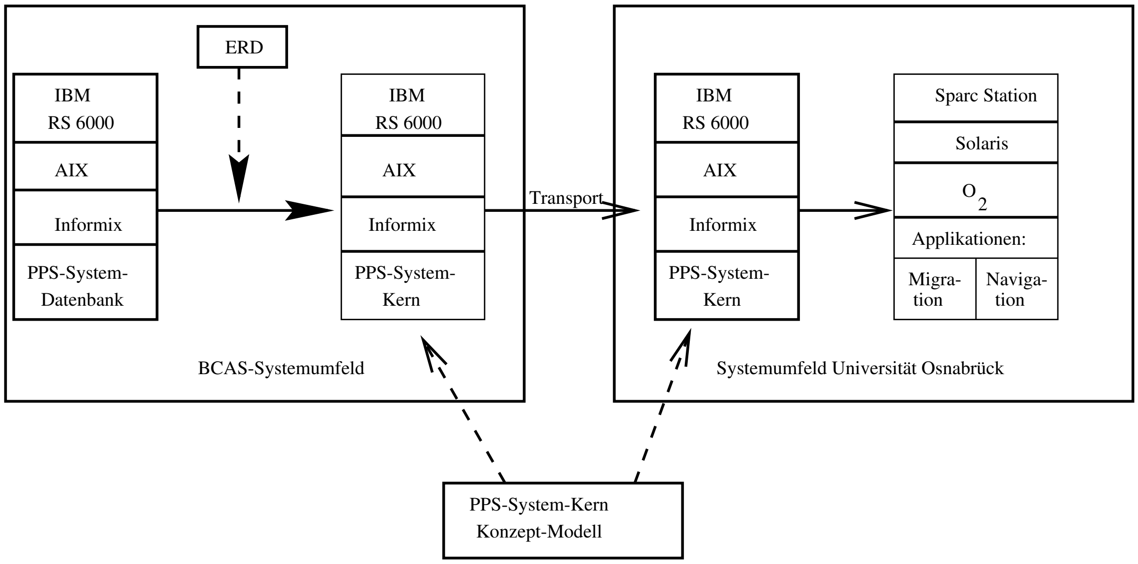

Abbildung 2.2: Systemumfeld bei der Migration

`unload`-Befehls Daten aus der Datenbank selektiert und in eine Datei geschrieben. Diese Dateien werden anschließend in der „neuen" Datenbank wieder gelesen und die Tabellen auf diese Weise mit Daten gefüllt.

Nachdem die Datenbank mit dem PPS-Kern bei der Firma *bcas* auf einer *Informix*-Datenbank erstellt und mit Daten gefüllt worden ist, wird wiederum ein `unload`-File erzeugt, das sowohl die Daten als auch die Datenstruktur enthält. Mit Hilfe dieser Datei wird im Rechenzentrum der Universität Osnabrück auf einer RS6000 wiederum eine *Informix*-Datenbank erzeugt, die nun den PPS-Kern mit den zugehörigen Daten enthält.

Als objektorientiertes System, auf dem die objektorientierte Datenbank des PPS-Systems entstehen soll, wird O_2 eingesetzt. O_2 ist ein objektorientiertes Datenbankmanagementsystem, das im Rahmen des 5-Jahres Projekts Altaïr von 1986 an entworfen und implementiert worden ist. An dem Projekt waren INRIA (Institut National de Recherche en Informatique et Automatique), IN2 (einer Tochtergesellschaft von Siemens) und die Universität Paris-Sud beteiligt [BDK92]. Ein Prototyp einer objektorientierten Datenbank wurde im September des Jahres 1989 fertiggestellt. Zu Beginn des Jahres 1991 kam O_2 als kommerzielles Produkt der Firma O_2Technologie auf den Markt.

Das objektorientierte Datenbankmanagementsystem O_2 befindet sich auf einer SPARCstation im Fachbereich Informatik der Universität. Dieser Rechner ist über Netz mit der RS6000, auf der sich die *Informix*-Datenbank befindet, verbunden. Die Migration der relationalen Datenbank auf die objektorientierte erfolgt wiederum über Dateien, da eine Online-Verbindung zwischen den Datenbanksystemen aus den genannten Gründen nicht gefordert wurde.

2.3 Ablauf der Migration

Die Migration ist ein zweigeteilter Vorgang. Einerseits ist die Datenstruktur der relationalen Datenbank zu migrieren und andererseits müssen die Daten in die objektorientierte Datenbank übertragen werden. Abbildung 2.3 veranschaulicht den Ablauf der Migration.

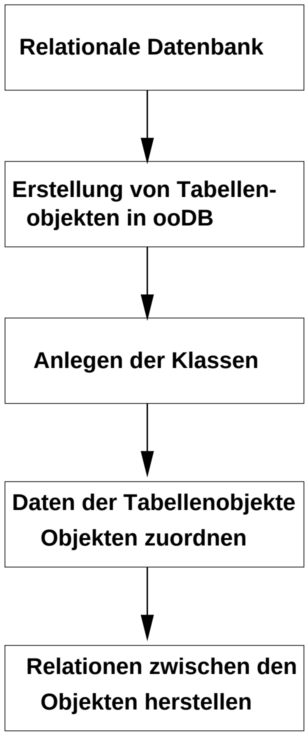

Abbildung 2.3: Ablaufdiagramm der Migration

Um das Operieren auf den Daten zu vereinfachen, werden diese zuerst nach O_2 portiert und in temporären Objekten zwischengespeichert. Die zu diesen Objekten gehörigen Klassen werden zur Laufzeit erzeugt und bilden die Tabellenstruktur der relationalen Datenbank ab.

Anschließend erfolgt die Migration der Datenstruktur, d.h. die Klassen für die objektorientierte Datenbank werden implementiert.

Nachdem die Klassendefinition abgeschlossen ist, werden Objekte dieser Klassen erzeugt und mit den Daten der temporären Objekte gefüllt.

Kapitel 3

PPS-Kern

Grundlage der Diplomarbeit ist ein auf einer relationalen Datenbank beruhendes Produktionsplanungs- und Steuerungssystem (PPS-System). Ein PPS-System beschäftigt sich mit der Planung und Ausführung von Kundenaufträgen von der Auftragsannahme bis hin zur Auslieferung. Dabei umfaßt ein solches System die Bereiche Materialwirtschaft, Fertigungsplanung und Fertigungssteuerung.

3.1 Extraktion des PPS-Kerns aus einer „realen“ PPS-Datenbank

Um zu prüfen, ob sich ein PPS-System, das auf einer relationalen Datenbank beruht, auch in eine objektorientierte Datenbank migrieren läßt, genügt es, die Essenz des PPS-Systems zu extrahieren und diese zu migrieren. Die Datenstruktur des zu extrahierenden PPS-Kerns muß zunächst festgelegt werden. Dazu wird ein logisches Datenmodell in Form eines Entity-Relationship-Diagramms (ER-Diagramm) erstellt, das die Datenstruktur der für den PPS-Kern benötigten Daten beschreibt.

3.2 PPS-Kern

Die Datenstruktur des PPS-Kerns, der sowohl in *Informix* als auch in O_2 realisiert werden soll, wird zunächst anhand eines ER-Diagramms definiert. Abbildung 3.1 zeigt das entstandene ER-Diagramm.

Für die Auftragsbearbeitung benötigt jeder PPS-Kern einen `Kunden` als Entity. Da Kunden und `Lieferanten` einige gemeinsame Attribute haben, werden diese Attribute unter dem En-

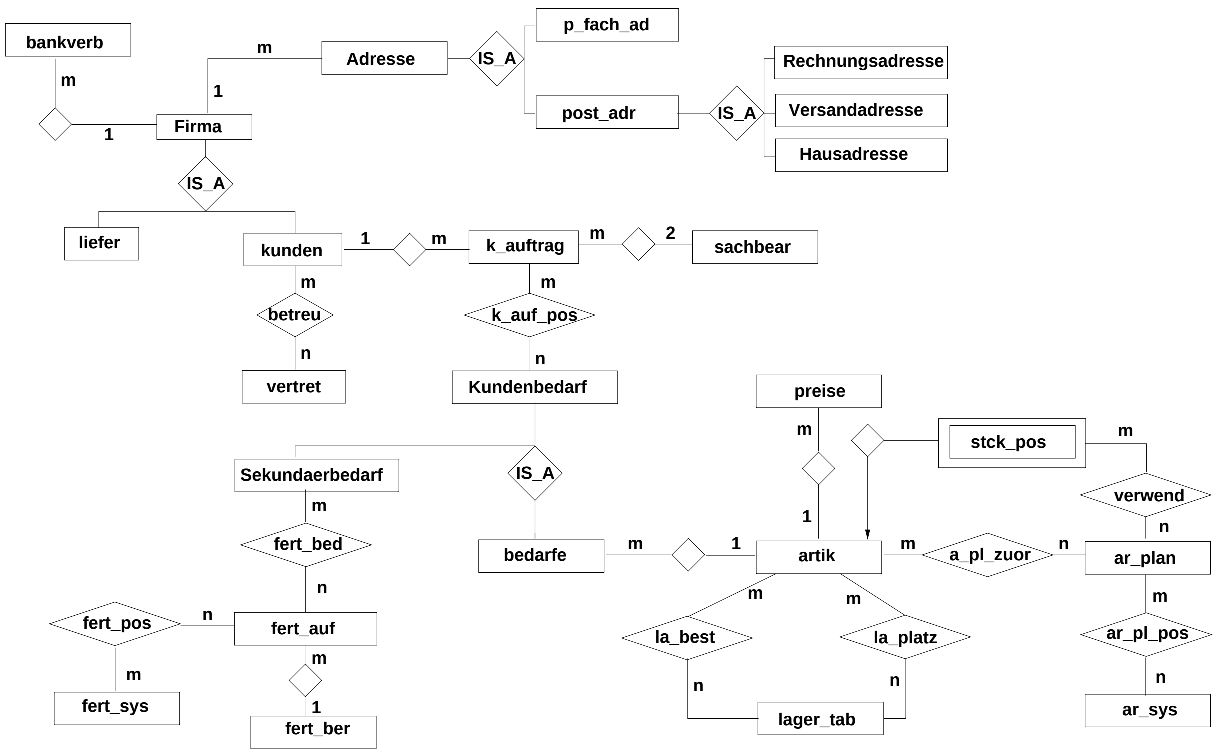

Abbildung 3.1: PPS-Kern

tity `Firma` zusammengefaßt. Jeder Firma ist eine `Adresse` zugeordnet. Eine Firma kann mehrere Adressen haben, z.B. eine Hausadresse und eine Rechnungsanschrift. Deshalb wurde die `Postadresse` in `Rechnungsadresse`, `Versandadresse` und `Hausadresse` unterteilt. Außerdem kann eine Firma über ein Postfach und somit über eine `Postfachadresse` verfügen.

Jeder Kunde wird von `Vertretern` betreut.

Durch einen Kunden wird ein `Kundenauftrag` definiert. Dabei kann ein Kunde mehrere Kundenaufträge definieren, ein Kundenauftrag ist jedoch immer genau einem Kunden zugeordnet. Zu einem `Kundenauftrag` gehört ein `Sachbearbeiter` und ein stellvertretender Sachbearbeiter. D.h. auftragsbezogen wird ein Kunde immer von einem Sachbearbeiter betreut, ein Sachbearbeiter ist aber für mehrere Kundenaufträge verantwortlich.

Ein Kundenauftrag besteht aus mehreren `Kundenauftragspositionen`, die fortlaufend numeriert werden. Durch jede `Kundenauftragsposition` wird ein `Kundenbedarf` (Primärbedarf) definiert.

Ein Kundenbedarf legt den Bedarf an Artikeln fest, die für den Kundenauftrag benötigt werden.

Einem `Artikel` ist immer mindestens ein `Preis` zugeordnet. Es kann aber auch mehrere unterschiedliche Preise für einen Artikel geben. Daher handelt es sich hier um eine $1 : n$-Beziehung.

Ein Artikel kann einen `Lagerplatz` belegen. Handelt es sich bei dem Artikel um ein Kaufteil, welches direkt in die Fertigung übernommen wird, so belegt dieser `Artikel` keinen Lagerplatz.

Ein Lagerplatz befindet sich in genau einem `Lager`. Ein Artikel kann mehrere Lagerplätze in unterschiedlichen Lagern belegen.

Bei einem Artikel kann es sich um ein Oberteil oder ein Unterteil einer Stückliste handeln. Eine Stückliste legt fest, aus welchen Einzelteilen (sog. Unterteilen) sich ein Artikel (ein Oberteil) zusammensetzt. Außerdem gibt eine Stückliste Auskunft darüber, an welcher Position und in welcher Menge ein Unterteil in ein Oberteil eingeht. Eine Stückliste besteht aus einer Menge von Stücklistenpositionen, wobei jede Stücklistenposition zwei Artikel zueinander in Beziehung setzt. Die Stücklistenpositionen werden fortlaufend numeriert. Da eine Stückliste in einem `Arbeitsplan` verwendet werden kann, wird aus dem Beziehungstyp `stck_pos` ein sogenanntes weak-Entity, da es eine Beziehung zu einer Beziehung im ER-Diagramm nicht geben kann.

Für jedes Oberteil ist ein `Arbeitsplan` definiert. Ein Arbeitsplan hat eine Beziehung zu dem Entity `Arbeitssystem`. Ein Arbeitssystem beschreibt einen Arbeitsgang, der zur Fertigung eines Oberteils erforderlich ist. Die Beziehung, die zwischen den beiden Entity-Typen besteht, ist die `Arbeitsplanposition`. Durch die Arbeitsplanposition werden die Arbeitssysteme, die in einem Arbeitsplan verwendet werden, fortlaufend numeriert.

Bei der Fertigung eines Artikels entsteht ein `Sekundaerbedarf`, d.h. es entsteht ein Bedarf an Artikeln, die für die Fertigung benötigt werden, aber nicht direkt verkauft werden. Sowohl der Kundenbedarf, als auch der Sekundärbedarf, stellen einen `Bedarf` dar. Aus diesen Bedarfen entsteht, nachdem eine Losgrößenbildung erfolgt ist, ein `Fertigungsauftrag`. Ein Fertigungsauftrag ist in `Fertigungsauftragspostionen` unterteilt, die wiederum einen Sekundärbedarf an Artikeln auslösen. Durch die Fertigungsauftragspostion erfolgt eine fortlaufende Numerierung der Fertigungssysteme, die zur Fertigung eines bestimmten Artikels erforderlich sind. Der Fertigungsauftrag kann auf diese Weise als konkrete Ausprägung eines Arbeitsplanes angesehen werden.

3.3 Die relationale Datenbank des PPS-Kerns

Aus dem oben beschriebenen logischen Datenmodell entsteht nun eine relationale Datenbank. In diesem Fall handelt es sich dabei um eine *Informix*-Datenbank.

Beim Datenbankdesign werden zunächst aus allen bis auf die spezialisierten Entity-Typen (IS_A-Beziehungen) Tabellen erzeugt. Hat ein Entity-Typ eine $m : n$-Beziehung zu einem anderen Entity-Typ, so wird für diese Beziehung wiederum eine Tabelle erzeugt, die die Primärschlüssel der beiden anderen Tabellen enthält. Mittels dieser Tabelle kann später die Beziehung zwischen den Daten hergestellt werden. Bei einer $1 : n$-Beziehung zwischen zwei Tabellen wird dagegen keine neue Tabelle erzeugt, sondern in einer der beiden Tabellen wird der Primärschlüssel der anderen Tabelle als sogenannter Fremdschlüssel mit aufgenommen. Dabei ist es nicht beliebig, in welcher der beiden Tabellen der Schlüssel aufgenommen wird.

Nachfolgendes Beispiel zeigt, daß der Schlüssel in der Tabelle als Attribut aufgenommen werden muß, bei der mehrere Einträge einen Eintrag der anderen Tabelle referenzieren. Ein Kunde kann z.B. mehrere Konten bei verschiedenen Banken haben. D.h. zwischen den Entity-Typen bzw. den Tabellen `kunde` und `bankverb` besteht eine $1 : n$-Beziehung. In diesem Fall muß der Primärschlüssel der Tabelle `kunde` in die Tabelle `bankverb` aufgenommen werden. Wenn der Primärschlüssel der Tabelle umgekehrt in die Tabelle `kunde` aufgenommen würde, könnte ein Kunde nur ein Konto haben. Außerdem könnten dann zwei oder mehrere Kunden dasselbe Konto haben, was nicht sein darf.

Bei der Modellierung der spezialisierten Entity-Typen gibt es zwei Möglichkeiten:

i) Es wird für jeden Entity-Typ eine eigene Tabelle erzeugt, die jeweils auch die Attribute des allgemeineren Typs enthält, für den keine Tabelle erzeugt wird.

ii) Es wird nur für den allgemeineren Entity-Typ eine Tabelle erzeugt und die unterschiedlichen Spezialisierungen werden anhand eines zusätzlichen Attributs unterschieden.

Enthalten die spezialisierten Entity-Typen zusätzliche Attribute, so wird das erste Verfahren angewendet. Handelt es sich bei den spezialisierten Entity-Typen dagegen nur um logische Unterscheidungen, so verfährt man wie unter Punkt zwei beschrieben.

3.4 Übernahme von Realdaten aus der PPS-Datenbank in die PPS-Kern-Datenbank

Die „reale“ Datenbank besteht aus etwa 500 Tabellen, aus denen die Daten extrahiert werden müssen. Da die Daten der realen Datenbank anders strukturiert sind als in dem entstandenen PPS-Kern, werden mit Hilfe des in SQL vorgesehenen `unload`-Kommandos Daten aus der realen Datenbank herausgefiltert, die über die passende Struktur verfügen. Diese Daten werden in eine Datei geschrieben und später in der *Informix*-Datenbank des PPS-Kerns wieder eingelesen. Die Daten, die in diesen sogenannten `unload`-Dateien stehen, können anschließend direkt in die Tabellen der *Informix*-Datenbank eingefügt werden.

Für die Artikeltabelle sieht ein solches Kommando folgendermaßen aus:

```
unload to "artik.unl"
      select tst_nr, artbez1, me_lager
      from mbi_tstbs;
```

Nach dem `unload`-Befehl kann eine Datei angegeben werden, in die die entladenen Daten geschrieben werden sollen. Der anschließende `select`-Befehl sucht nur die gewünschten Daten aus der Tabelle heraus. Auf diese Weise können auch Daten aus mehreren Tabellen selektiert werden. In dem Fall müssen in einer `where`-Klausel die Schlüssel der beiden Tabellen verglichen werden. Dies ist z.B. bei den Kundendaten erforderlich:

```
unload to "kunden.unl"
      select unique mbi_debit.debitor_nr, mbi_adres.name1,
             mbi_brncd.branche_bez, mbi_skont.skonto_proz1,
             mbi_skont.skonto_tage1, mbi_debit.vers_limit,
             mbi_debit.versich_nr, mbi_debit.kreditlimit, saldo
      from mbi_debit, mbi_adres,  mbi_brncd, mbi_debzu, mbi_skont
      where mbi_debit.adress_nr = mbi_adres.adress_nr
            and mbi_debit.branche_nr = mbi_brncd.branche_nr
            and mbi_debzu.debitor_nr=mbi_debit.debitor_nr
            and mbi_debzu.skonto_nr = mbi_skont.skonto_nr;
```

Die Datei, in die die Daten geschrieben werden, heißt in diesem Fall `kunden.unl`. Jede Zeile dieser Datei repräsentiert ein Kundendatum. Die Einträge der einzelnen Spalten sind jeweils durch das Zeichen "`|`" voneinander getrennt.

```
5033|Richard Bauer|Möbelhandel|5,0|30|0,0| |0,0|0,0|
6004|Ernst Bergemann GmbH|Möbelhandel|5,0|30|0,0| |0,0|0,0|
6103|Schreinerei Heinz Becker|Möbelhandel|5,0|8|0,0| |0,0|0,0|
```

Mit dem `load`-Kommando können die Daten aus dieser Datei wieder in die Tabelle geladen werden:

```
load from kunden.unl insert into kunden;
```

Auf diese Weise wird die Tabelle `kunden` der PPS-Kern-Datenbank mit Daten gefüllt.

So entsteht eine Datenbank, die nur noch über 27 Tabellen verfügt. Die Anzahl der Einträge in den Tabellen ist sehr unterschiedlich, wie die nachfolgende Tabelle zeigt.

Tabellenname	Anzahl Einträge
artik	2765
ar_plan	114
ar_pl_pos	795
ar_sys	129
a_pl_zuor	150
bankverb	3
bedarf	2135
betreu	1384
fert_auf	23
fert_bed	25
fert_ber	17
fert_pos	79
fert_sys	33
k_auftrag	1072

Tabellenname	Anzahl Einträge
kunden	88
k_auf_pos	1980
lager_tab	20
la_best	42
la_platz	26
liefer	2
post_adr	42
preise	1821
p_fach_ad	15
sachbear	6
stck_pos	6655
vertret	40
verwend	3

Summe: 19556
Mittelwert: 724

Kapitel 4

Tabellenobjekte

Migration ist abgeleitet vom lateinischen Wort *migratio* („Umzug“) und bedeutet Wanderung. In diesem Fall „wandern“ Daten von einer relationalen Datenbank in eine objektorientierte Datenbank.

Beim Migrationskonzept muß zwischen der Migration der Datenstruktur und der Migration der Daten unterschieden werden. Letztlich sollen die Daten aus den Tabellen einer relationalen Datenbank in Objekte einer objektorientierten Datenbank überführt werden. Objekte werden durch Klassen definiert, die in einer Vererbungshierarchie organisiert sind. Sollen Daten in Objekte migriert werden, muß zunächst eine Klassenhierarchie geschaffen werden. Die Erzeugung dieser Klassenhierarchie stellt die Migration der Datenstruktur dar.

Um die Übertragung der Daten in das objektorientierte Modell möglichst einfach zu gestalten, wird ein „Zwischenschritt“ durchgeführt. In diesem Zwischenschritt werden nur die Daten nach O_2 portiert, d.h. es werden auf objektorientierter Seite Klassen angelegt, die die Daten einer Tabelle aufnehmen können. Auf diese Weise erzeugt man sozusagen eine relationale Datenbank auf objektorientierter Seite.

4.1 Migration der Daten

Für jede in *Informix* vorhandene Tabelle wird in O_2 temporär eine Klasse erzeugt, die den kompletten Inhalt der Tabelle aufnehmen kann. Diese Klassen werden dynamisch erzeugt und später wieder gelöscht. Die temporären Klassen haben folgende Eigenschaften:

- Für jede relationale Tabelle wird eine Klasse erzeugt.
- Ein Objekt der jeweiligen Klasse enthält alle Einträge der zugehörigen Tabelle.

- Die Klassen unterscheiden sich in der Struktur, haben aber viele gleiche Methoden.
  $\Longrightarrow$ Einführung einer gemeinsamen Superklasse.

- Die Superklasse enthält auch Attribute, um die Primär- und die Fremdschlüssel sowie den Klassennamen der Tabelle zu speichern.

- Die Klassen unterscheiden sich in der Anzahl der Attribute und in den Datentypen der Attribute, deshalb kann es keine gemeinsame Methode zum Laden der Daten geben.
  $\Longrightarrow$ `load` gehört zu den Subklassen.

Beim Generieren der Klassen wird das von O_2 zur Verfügung gestellte Metaklassen-Konzept verwendet, das später noch näher erläutert wird.

Die Struktur der Klassen, die erzeugt werden, kann aus der Datei ausgelesen werden, die auch *Informix* zum Anlegen der Tabellen der relationalen Datenbank verwendet. Der Name der Datei ist beliebig. Um Verwechslungen mit anderen Dateien auszuschließen soll die Datei mit `tables.sql` bezeichnet werden. Für die Tabelle `kunden` sieht der Eintrag in der Datei `tables.sql` beispielsweise folgendermaßen aus:

```
create table kunden
(       Kundennr integer,
        Firmenname char(20),
        Branche char(30),
        Skontoproz float,
        Skontotage integer,
        Vers_limit money,
        Versich_nr char(30),
        Kreditlimit money,
        saldo money,
        primary key (Kundennr)
        );
```

Aufgrund dieses Befehls wird in Informix eine Tabelle `kunden` angelegt, in der Kundendaten abgelegt werden können:

<u>Kundennr</u>	Firmenname	Branche	...

In O_2 wird folgende Klasse `T_kunden` erzeugt:

```
class T_kunden inherit TabellenObj
public type
    tuple(tabelle: set(tuple(Kundennr: integer,
                             Firmenname: string,
                             Branche: string,
                             Skontoproz: real,
                             Skontotage: integer,
                             Vers_limit: real,
                             Versich_nr: string,
                             Kreditlimit: real,
                             saldo: real)))
method
    public load(path: string)
end;
```

Die Klasse `T_kunde` wird von der Klasse `TabellenObj` abgeleitet. Diese Klasse kann als abstrakte Basisklasse angesehen werden, da von dieser Klasse selbst nie ein Objekt erzeugt wird. Objekte werden nur von den temporären Subklassen erzeugt. Die Subklassen erben alle Attribute und Methoden der Klasse `TabellenObj`. Zusätzlich erhält jede generierte Klasse ein Attribut mit dem Namen tabelle. Dieses Attribut besteht aus einer Menge von Tupeln, wobei jedes Tupel einen Eintrag der relationalen Tabelle repräsentiert. Das obige Beispiel zeigt, daß das Attribut tabelle der Klasse `T_kunden` aus einer Menge von Tupeln besteht, wobei jedes Tupel ein Kundendatum darstellt. Ein Objekt der Klasse `T_kunden` kann also den kompletten Inhalt der relationalen Tabelle `kunden` in dem Attribut tabelle speichern.

Da für die Realisierung auf objektorientierter Seite das objektorientierte Datenbankmanagementsystem O_2 verwendet wird, können zur Modellierung nur Datentypen verwendet werden, die O_2 anbietet.

4.2 O_2-Datentypen

Neben den elementaren Datentypen, wie `integer`, `real`, `string` etc., die mit den elementaren Datentypen, die C verwendet, vergleichbar sind, stellt O_2 die folgenden strukturierten Datentypen zur Verfügung:

- `tuple()`
- `set()`
- `list()`

- `unique set()`

Der Datentyp `tuple()` kann Objekte unterschiedlicher Klassen enthalten. Ein `set()` ist dagegen eine Menge von Objekten desselben Typs. Dies trifft auch auf die Datentypen `unique set()` und `list()` zu. Ein `unique set()` ist eine Menge, in der jedes Objekt nur einmal enthalten sein kann. Der Datentyp `list()` repräsentiert eine geordnete Menge.

Beim Anlegen der Klassen müssen die Datentypen der Attribute von *Informix* auf die Datentypen in O_2 abgebildet werden. Da es in *Informix* keine geschachtelten, sondern nur elementare Datentypen gibt, müssen bei der Abbildung der Datentypen nur die elementaren Datentypen berücksichtigt werden. Diese Abbildung sieht folgendermaßen aus:

Informix	$\rightarrow$	O_2
`integer`	$\rightarrow$	`integer`
`money`	$\rightarrow$	`real`
`char()`	$\rightarrow$	`string`
`float`	$\rightarrow$	`real`
`date`	$\rightarrow$	`Date`

Zu dem in *Informix* verwendeten Datentyp `date` gibt es in O_2 kein Analogon. Das Schema O_2Kit enthält eine Klasse `Date`. Zum Ablegen eines Datums wird in O_2 ein Objekt dieser Klasse erzeugt.

4.3 Die Klasse `TabellenObj`

Alle neu entstehenden Klassen sind Subklassen einer Klasse, die hier mit `TabellenObj` bezeichnet wird. Diese Klasse hat folgende Struktur:

```
class TabellenObj inherit Object public type
    tuple(primary_key: list(string),
          foreign_keys: list(tuple(name: string,
                                   table: string,
                                   reference: string,
                                   obj_ref: TabellenObj)),
          classname: string)
method
    private get_attributes: list(tuple(name: string,
                                       type: Meta_definition)),
    public set_primary(keys: list(string)),
```

```
    public get_primary_keys(classname: string): list(string),
    public define_primary(path: string): list(string),
    public define_foreign(path: string): list(tuple(name: string,
                                                    table: string,
                                                    reference: string)),
    public is_no_class: boolean,
    public define_class,
    public create_ObjectClass(classname: string,
                              attrlist: string),
    public find_referenced_table(tablename: string): TabellenObj,
    public attach,
    public create_named_Object,
    public set_classname(name: string),
    public set_foreign_reference,
    public initialize(path: string),
    public define_set_attr,
    public define_insert_references: integer,
    public def_insert_for_class,
    public redefine_set_attr,
    public return_param: list(string),
    public create_subclass_objects,
    public create_get_tuple,
    public menu: list(string),
    public create_get_one_entry,
    public delete_set_attr
end;
```

Alle Tabellenklassen besitzen Attribute, um Primär- und Fremdschlüssel sowie den Namen der Klasse abzulegen, der die Daten später zugeordnet werden sollen. Diese Attribute werden daher in der Klasse `TabellenObj` angelegt.

Das Attribut primary_key ist eine Liste vom Datentyp `string`, da eine Tabelle mehr als einen Primärschlüssel haben kann.

Für die referentielle Integrität ist es notwendig, daß bei den Fremdschlüsseln die Tabelle und der Name der Spalte mit angegeben werden, auf die der Fremdschlüssel verweist. Bei der Liste der Fremdschlüssel wird für jeden Schlüssel ein `tuple()` angelegt. In jedem `tuple()` bezeichnet das Attribut name den Namen des Fremdschlüssels in dieser Tabelle. Unter dem Attribut table wird der Name der referenzierten Tabelle abgespeichert. reference bezeichnet den Namen der Spalte in der referenzierten Tabelle. Diese Informationen werden bereits in *Informix* benötigt. Da die Schlüssel auch in der *Informix*-Datenbank definiert sein müssen, können die Schlüssel ebenfalls aus der Datei tables.sql ausgelesen werden. Zusätzlich enthält jedes Tupel des Attributs foreign_keys einen Zeiger auf das referenzierte Tabellenobjekt, der unter obj_ref eingetragen wird.

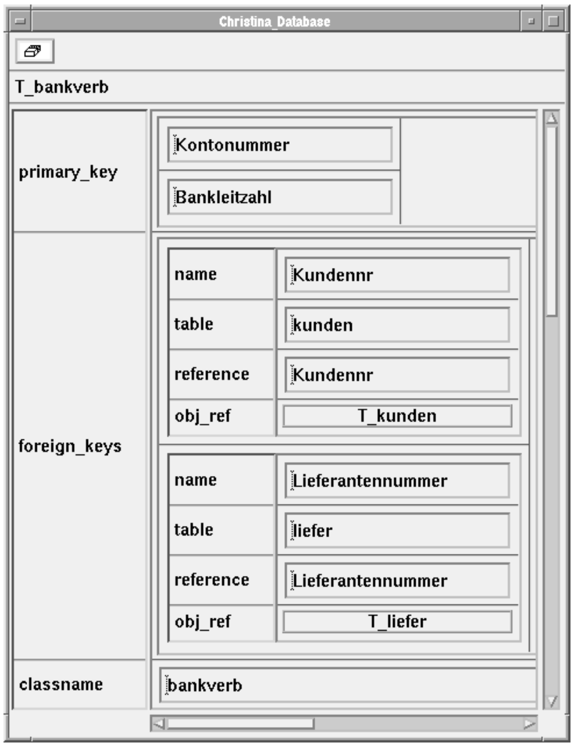

Abbildung 4.1: Objekt der Klasse T_bankverb

Bei der Tabelle bankverb handelt es sich um eine Tabelle, die die Kundennr und die Lieferantennummer als Fremdschlüssel besitzt.

```
create table bankverb
(       Kontonummer     integer,
        Bankleitzahl    integer,
        Kreditinstitut  char(20),
        Kundennr        integer,
        Lieferantennummer       integer,
        primary key (Kontonummer,Bankleitzahl),
        foreign key (Kundennr) references kunden(Kundennr),
        foreign key (Lieferantennummer) references liefer(Lieferantennummer)
        );
```

Die Klasse `TabellenObj` verfügt über Methoden, um die Primärschlüssel und die Fremdschlüssel zu definieren. Die Schlüssel können allerdings erst bei den Objekten eingetragen werden, während die Struktur der Tabelle direkt in der Klasse abgelegt wird.

Nachdem ein Objekt der Subklasse `T_bankverb` angelegt worden ist, sorgt die Methode `initialize` der Superklasse `TabellenObj` dafür, daß Primär- und Fremdschlüssel und der Klassenname bei dem Objekt eingetragen werden. Abbildung 4.1 zeigt ein Objekt der Klasse `T_bankverb`, bei dem die Primär- und Fremdschlüssel eingetragen sind.

Mit Hilfe der Liste der Primärschlüssel und der Liste der Fremdschlüssel kann, nachdem die Objekte erzeugt worden sind, dafür gesorgt werden, daß die Beziehungen eingetragen werden.

Für jede dieser neu entstehenden Klassen wird eine Methode generiert, die die zugehörigen Daten aus der `unload`-Datei von *Informix* laden kann. Die Methode kann nicht allgemein als Methode der Klasse `TabellenObj` geschrieben werden, da sich die Einträge der Tabellen sowohl in der Anzahl als auch im Datentyp unterschieden.

4.4 O_2-Metaklassen

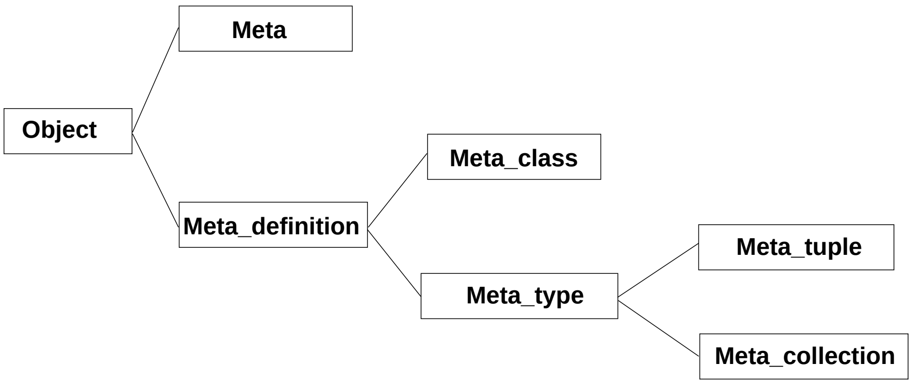

Abbildung 4.2: Metaklassen von O_2

Bei der technischen Umsetzung des Konzeptes wird das von O_2 zur Verfügung gestellte Metaklassen-Konzept verwendet. Mit Hilfe dieser Metaklassen ist es möglich, zur Laufzeit einer Applikation eine Klasse zu erzeugen oder zu löschen. Außerdem können Programme oder Methoden erzeugt und auch anschließend ausgeführt werden.

O_2 verwendet die Metaklassen: `Meta`, `Meta_definition`, `Meta_class`, `Meta_type`, `Meta_collection` und `Meta_tuple`. Abbildung 4.2 zeigt die Vererbungshierarchie, in der die Klassen angeordnet sind.

O_2 erzeugt für jede Klasse, die in der Klassenhierarchie vorhanden ist, und jeden Typ ein sogenanntes *Metaklassenobjekt*. Dieses Objekt kennt die Struktur der angelegten Klasse bzw. des Typs. Dabei wird für eine Klasse ein Objekt der Klasse `Meta_class` angelegt, während für einen Typ ein Objekt der Klasse `Meta_type` angelegt wird. Für den Datentyp `tuple` wird jeweils ein Objekt der Klasse `Meta_tuple` erzeugt. Die Klasse `Meta_collection` wird für die Datentypen `set()` und `list()` verwendet. Mit Hilfe der Methoden, die in den verschiedenen Klassen des `Meta_Schemas` implementiert sind, kann man z.B. die Attribute einer bestimmten Klasse abfragen oder eine Liste aller in einer Klasse vorhandenen Methoden erhalten. Mit Hilfe des `Meta_Schemas` ist es möglich, die Tabellenobjekte, bzw. zunächst die zugehörigen Klassen, zur Laufzeit der Applikation zu erzeugen und wieder zu löschen.

Eine Klasse wird wie folgt erzeugt:

1. Erzeugen eines Objekts der Klasse `Meta`
2. Instantiieren des Objekts, z.B.
   `run body{ Schema = new Meta; }`
3. Methodenaufruf:
   `Schema->command(`<*commandstring*>`)`

Beim dynamischen Erzeugen einer Klasse wird ein <*commandstring*> generiert, der auf der O_2-Shell zum Erzeugen einer Klasse eingegeben werden muß. Anschließend wird bei dem Objekt `Schema` die Methode `command` mit diesem String aufgerufen. Für die Tabelle `kunden` wird folgender string generiert:

```
create class T_kunden inherit TabellenObj
public type
    tuple(tabelle: set(tuple(Kundennr: integer,
                             Firmenname: string,
                             Branche: string,
                             Skontoproz: real,
                             Skontotage: integer,
                             Vers_limit: real,
                             Versich_nr: string,
                             Kreditlimit: real,
                             saldo: real)))
end;
```

Auf diese Weise wird für jede in *Informix* vorhandene Tabelle eine temporäre Klasse in O_2 geschaffen. Der Name der Klasse setzt sich zusammen aus dem String „`T_`“ und dem Namen der zugehörigen Tabelle in *Informix*. Da es sich um temporäre Klassen handelt, wird dem Namen der Klasse der String „T_“ hinzugefügt. Außerdem soll beim Entwurf der endgültigen Klassen der Name der Tabelle verwendet werden. Abbildung 4.3 zeigt die so entstehende Klassenhierarchie.

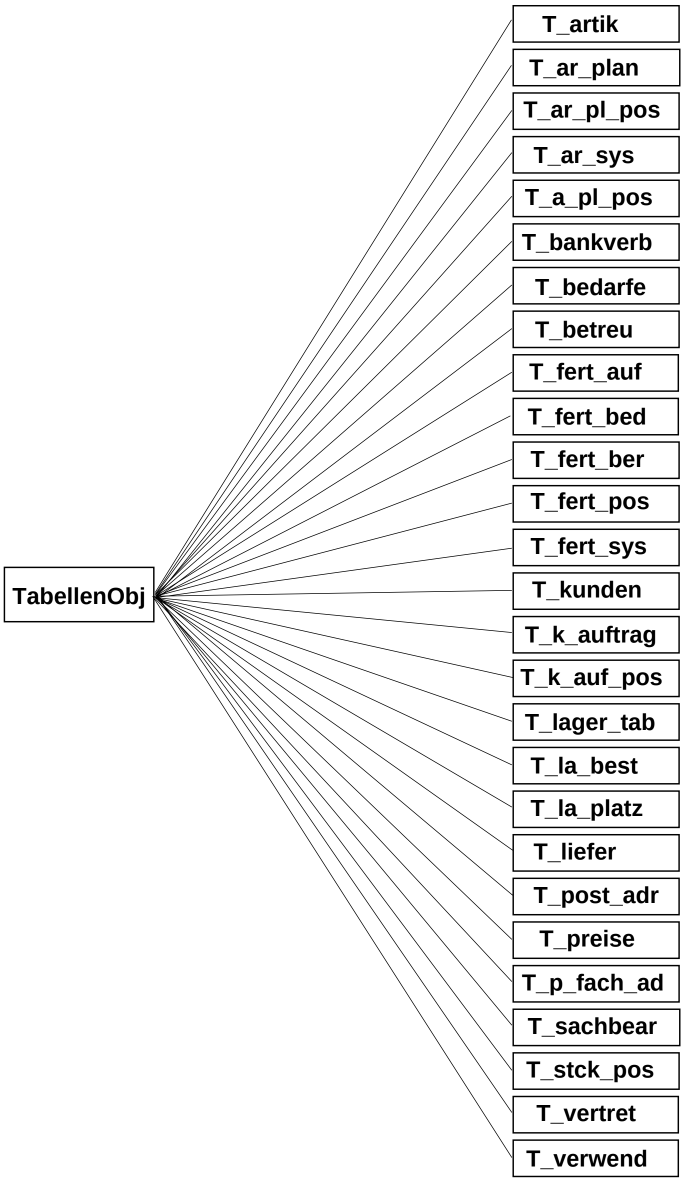

Abbildung 4.3: Klassenhierarchie der temporären Klassen

In einem zweiten Schritt wird für jede dieser Klassen eine Methode zum Laden der Daten aus der Datei erzeugt.

Methoden sind Programme, die einer Klasse zugeordnet sind. Sie beschreiben das Verhalten aller Objekte dieser Klasse. Eine Methode wird entweder innerhalb eines `create class`-Kommandos oder mit dem Kommando `create method` erzeugt. Methoden bestehen in O_2 aus zwei Teilen:

- der Signatur: `method signature`
- und dem Rumpf: `method body`.

Die Signatur beschreibt die Parameter, die die Methode erwartet bzw. zurückliefert. Der Rumpf beinhaltet die Implementierung der Methode. Die Signatur einer Methode muß definiert werden, bevor der Rumpf implementiert werden kann.

Das Generieren der `load`-Methode geschieht auf dieselbe Weise wie das Generieren einer Klasse. D.h., zum dynamischen Erzeugen einer Methode wird nur ein anderer *<commandstring>* generiert. Allerdings muß die Methode `command` des Objekts `Schema` zweimal für jede Klasse aufgerufen werden. Beim ersten Aufruf wird nur die Signatur der Methode erzeugt.

```
cmd = "create method public load(path:string) in class T_kunden;";
Schema->command( cmd );
```

Beim zweiten Aufruf der `command`-Methode wird der Rumpf der `load`-Methode erzeugt.

Die Methode `load` der Klasse `T_kunden` liest die Daten aus der Datei *kunden.unl*. Der Pfad, unter dem die Datei gesucht werden soll, wird der Methode als Parameter übergeben. Eine Kundendatei, die Daten zweier Kunden umfaßt, könnte folgendermaßen aussehen:

```
5033|Richard Bauer|Möbelhandel|5,0|30|0,0| |0,0|0,0|
6004|Ernst Bergemann GmbH|Möbelhandel|5,0|30|0,0| |0,0|0,0|
```

Eine Zeile in einer `unload`-Datei entspricht einem Kundendatum. Die verschiedenen Attributausprägungen sind durch das Symbol „`|`“ voneinander getrennt. Die Methode `load` durchläuft die Datei zeilenweise und erzeugt pro Zeile ein neues Tupel, das, nachdem es mit Daten gefüllt worden ist, dem Attribut `self->tabelle` hinzugefügt wird.

4.5 Das Tabellenobjekt T_kunden

Um die Daten, die die Tabelle `kunden` in *Informix* enthält, zu speichern, wird *ein* Objekt der Klasse `T_kunden` in O_2 erzeugt und unter dem Namen `tmp_T_kunden` persistent gespeichert. Anschließend werden mit Hilfe der Methoden die Primär- und Fremdschlüssel eingetragen. Bei der Tabelle `kunden` wird nur ein Primärschlüssel eingetragen, nämlich die Kundennummer. Die

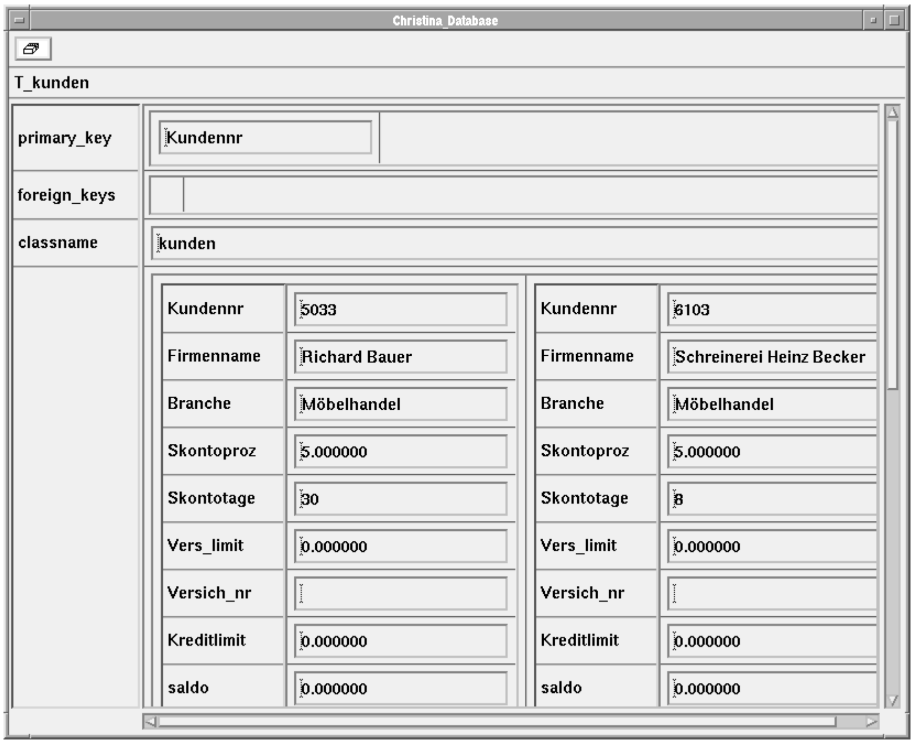

Abbildung 4.4: TabellenObjekt von `T_kunden`

Tabelle `kunden` hat zwar Beziehungen zu anderen Tabellen, da es sich aber immer um $1 : n$-Beziehungen handelt, d.h. ein Kunde immer Beziehungen zu mehreren Einträgen in der anderen Tabelle hat, wird die Kundennummer in der jeweils anderen Tabelle als Fremdschlüssel aufgenommen. Nachdem das Objekt `tmp_T_kunden` erzeugt und initialisiert worden ist, wird es mit Daten gefüllt. Dies geschieht durch den Aufruf der Methode `load`.

Abbildung 4.4 zeigt das Tabellenobjekt `tmp_T_kunden` der Klassen `T_kunden`. Dieses Objekt enthält nun alle Einträge der Tabelle `kunden` in *Informix*.

Kapitel 5

Das objektorientierte Datenmodell

Nachdem die Migration der Daten von der relationalen Datenbank in die Tabellenobjekte erfolgt ist, befinden sich alle Einträge einer Tabelle als Tupel in einem Objekt. Die objektorientierte Datenbank soll für *einen* Eintrag einer Tabelle *ein* Objekt enthalten, es sei denn, es handelt sich um eine Tabelle, die nur die Zuordnung zwischen zwei Tabellen abbildet. Dazu müssen auf objektorientierter Seite Klassen geschaffen werden, die jeweils ein solches Datum aufnehmen können.

Nachdem das objektorientierte Datenmodell bzw. die Klassenhierarchie erzeugt worden ist, werden die Daten in die vorgesehenen Objekte portiert, und die Tabellenobjekte werden einschließlich der zugehörigen Klassen gelöscht.

5.1 Migration der Datenstruktur

Da die Daten aus den Tabellenobjekten automatisch übertragen werden sollen, erfolgt der Entwurf der Klassen kanonisch. Bei der objektorientierten Datenmodellierung liegt das logische Datenmodell zugrunde, d.h. ausgehend vom logischen Entity-Relationship-Diagramm des PPS-Kerns wird ein objektorientiertes Datenmodell erstellt.

Für jedes Entity im logischen Datenmodell entsteht eine Klasse. Die Attribute der Klasse entsprechen den Attributen des Entities. Auch die Schlüsselattribute der Entities bleiben erhalten, obwohl sie im objektorientierten Modell nicht mehr benötigt werden.

Um die Relationen abbilden zu können, müssen in den verschiedenen Klassen zusätzliche Attribute aufgenommen werden. Im Gegensatz zum relationalen Modell werden im objektorientierten allerdings immer in beiden Klassen zusätzliche Attribute aufgenommen, unabhängig davon, ob es sich um eine $1:n$-Beziehung oder eine $m:n$-Beziehung handelt.

5.2 Wie werden Relationen abgebildet?

Da sich Relationen in ihrer Kardinalität unterscheiden, müssen sie unterschiedlich auf das objektorientierte Modell übertragen werden, damit die Kardinalität „erhalten“ bleibt. In dem vorliegenden ER-Modell treten vier verschiedene Arten von Relationen auf. Wie die Abbildung der hier auftretenden Relationen auf die Klassen erfolgt, wird nachfolgend beschrieben.

5.2.1 $1 : n$-Beziehung

Bei der Beziehung zwischen Kunden und Kundenaufträgen handelt es sich, wie Abbildung 5.1 zeigt, um eine $1 : n$-Beziehung. Ein Kunde kann mehrere Kundenaufträge erteilen, aber jedem Kundenauftrag kann nur genau ein Kunde zugeordnet werden.

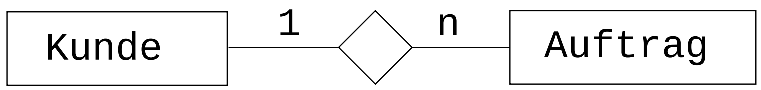

Abbildung 5.1: Beispiel einer $1 : n$-Beziehung

Im relationalen Modell wird bei einer solchen Beziehung der Fremdschlüssel der Tabelle `Kunden` in die Tabelle `Kundenauftrag` aufgenommen.

Tabelle `Kunden`:

<u>Kundennr</u>	Firmenname	Branche	...
5033	Richard Bauer	Möbelhandel	

Tabelle `Kundenauftrag`:

<u>Kundenauftragsnr</u>	<u>Kundennr</u>	Erfassungsdatum	Bestelldatum	...
20577	5033	...		

Im objektorientierten Modell entsteht in beiden Klassen ein zusätzliches Attribut, das auf ein oder mehrere Objekte der anderen Klasse verweisen kann.

Die zugehörige Klassendefinition sieht folgendermaßen aus:

```
class Kunden inherit Firma public type
    tuple(Kundennr: integer,
          Firmaname: char(20),
          Branche: char(30),
          saldo: real,
          Auftrag: set(Auftrag))
end;

class Auftrag inherit Object public type
    tuple(Kundenauftragsnr: integer,
          Kundennr: integer,
          Erfassungsdatum: Date,
          Bestelldatum: Date,
          .
          .
          Kunden: Kunden)
end;
```

Da ein Auftrag immer genau zu einem Kunden gehört, wird in der Klasse `Auftrag` ein Attribut eingeführt, das auf *genau einen* `Kunden` zeigen kann. Bei der Klasse `Kunden` wird dagegen ein Attribut erzeugt, das auf eine Menge von Aufträgen verweisen kann.

Als Namen der entstehenden Attribute werden die Namen der zueinander in Beziehung stehenden Klassen verwendet.

5.2.2 $m : n$-Beziehung

Für eine $m : n$-Beziehung wird im relationalen Modell eine zusätzliche Tabelle eingeführt, die die Schlüsselattribute der beiden zueinander in Beziehung stehenden Tabellen aufnimmt.

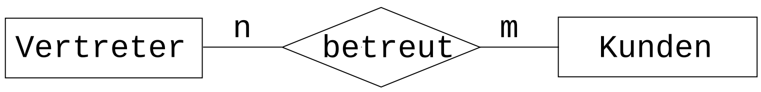

Abbildung 5.2: Beispiel einer $m : n$-Beziehung

Im relationalen Ansatz wird für das in Abbildung 5.2 gezeigte Beispiel neben den beiden Tabellen `Vertreter` und `Kunden` eine dritte Tabelle `betreut` erzeugt. Diese Tabelle enthält die

Primärschlüssel der beiden Tabellen als Fremdschlüssel und bildet auf diese Weise die Beziehung, die zwischen den beiden Tabellen besteht, ab.

Tabelle `Kunden`:

Kundennr	Firmenname	Branche	...
5033	Richard Bauer	Möbelhandel	...

Tabelle `Vertreter`:

Personalnummer
2

Tabelle `betreut`:

Kundennr	Personalnummer
5033	2

Im objektorientierten Ansatz entsteht keine zusätzliche Klasse. Hier entscheidet der Datentyp des eingeführten Attributs über die Kardinalität der Relation. Daher werden bei einer $m:n$-Beziehung auf beiden Seiten Attribute eingeführt, die eine Menge von Objekten der jeweils anderen Klasse aufnehmen können. Der Name des eingeführten Attributes entspricht dem Namen der Relation im ER-Diagramm.

Auf diese Weise entsteht folgende Klassendefinition:

```
class Kunden inherit Firma public type
    tuple(Kundennr: integer,
          Firmenname: char(20),
          Branche: char(30),
          saldo: real,
          betreut: set(Vertreter),
          Auftrag: set(Auftrag))
end;

class Vertreter inherit Object public type
    tuple(Personalnummer: integer,
          betreut: set(Kunden))
end;
```

5.2.3 $m : n$-Beziehung mit zusätzlichem Attribut

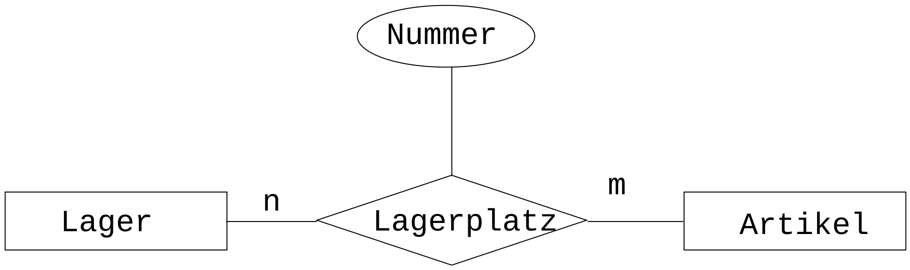

Abbildung 5.3: Beispiel einer $m : n$-Beziehung mit zusätzlichem Attribut

Bei der zwischen den Entities `Artikel` und `Lager` bestehenden Beziehung `Lagerplatz`, handelt es sich um eine $m : n$-Beziehung mit zusätzlichem Attribut. D.h., jeder Lagerplatz in einem Lager erhält eine Nummer. Im relationalen Ansatz sind die folgenden drei Tabellen entstanden:

Tabelle `Artikel`:

Artikelnummer	Artikelbez	Me_einheit_kbez
11	Billy 2-60	

Tabelle `Lager`:

Lagernummer	Lagerbez	Lagertyp
12	Bestellager, Fremdfert. anonym	Eigenes Lager

Tabelle `Lagerplatz`:

Artikelnummer	Lagernummer	Nummer
11	12	4

Die Abbildung zeigt, daß das Attribut in die Zuordnungstabelle `Lagerplatz` aufgenommen wurde. Um Datenredundanz zu vermeiden, wird das Attribut Lagerplatz im objektorientierten Datenmodell nicht in beiden Klassen als Attribut aufgenommen, sondern hier entsteht nur bei der Klasse `Lager` ein Attribut vom Typ `tuple()`, das ein Objekt der Klasse `Artikel` und eine Nummer umfaßt. Da es sich um eine $m : n$-Beziehung handelt, muß das Attribut Lagerplatz eine Menge solcher `tuple()` aufnehmen können. Bei der Klasse `Artikel` wird ein Attribut eingeführt, das auf eine Menge von Objekten der Klasse `Lager` verweisen kann. D.h., ein Artikel „weiß" in welchen Lagern er sich befindet. Das zugehörige Lager verwaltet die Information darüber, auf welchen Lagerplätzen sich der Artikel befindet.

Auf diese Weise entstehen folgende Klassen:

```
class Artikel inherit Object public type
    tuple(Artikelnummer: integer,
          Artikelbez: string,
          Me_einheit_kbez: string,
          Lagerplatz: set(Lager))
end;

class Lager inherit Object public type
    tuple(Lagernummer: integer,
          Lagerbez: string,
          Lagertyp: string,
          Lagerplatz: set(tuple(Nummer: integer,
                                Artikel: Artikel)))
end;
```

5.2.4 Sonderfall: Stücklisten

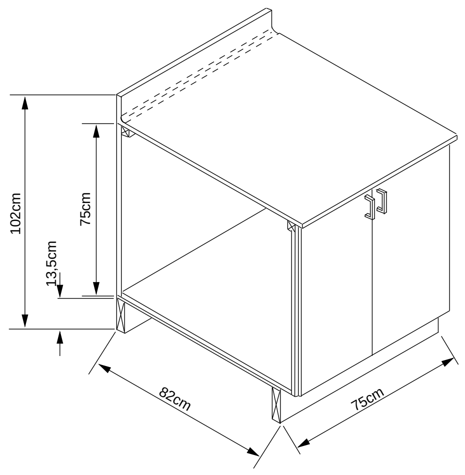

Abbildung 5.4: Konstruktionszeichnung eines Küchenschranks

Eine *Stückliste* zerlegt Erzeugnisse schrittweise in Baugruppen und Einzelteile. Die Stückliste gehört zu einem Artikel, einem sogenannten *Oberteil*. Ein Oberteil ist die Klassifizierung eines Artikels, in den *Unterteile* eingehen. Ein Oberteil setzt sich also aus verarbeiteten Unterteilen zusammen. Bei dem Unterteil handelt es sich wiederum um einen Artikel. Die Stückliste legt fest, in welcher Reihenfolge und in welcher Menge Unterteile in Oberteile eingehen.

Ein solches Oberteil ist beispielsweise ein Küchenschrank, wie ihn Abbildung 5.4 zeigt. Der Küchenschrank besteht aus den Unterteilen: Unterschrank, Tür und Arbeitsplatte.

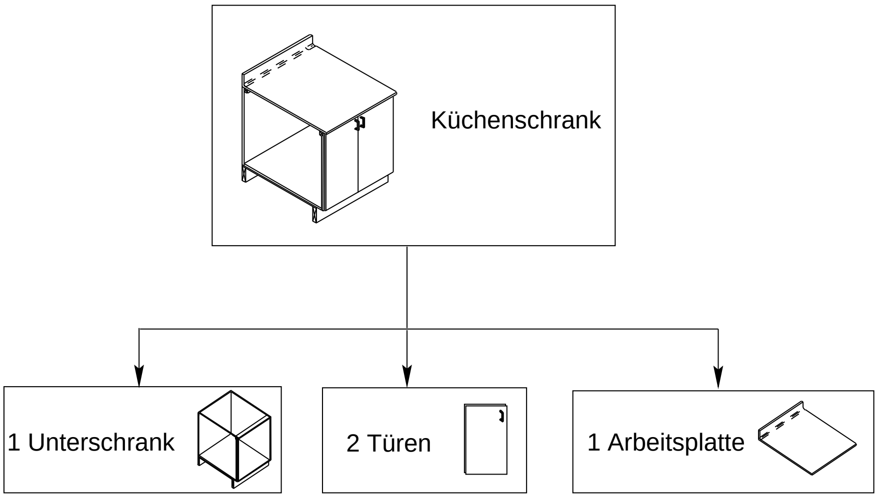

Abbildung 5.5: Baukastenstückliste für einen Küchenschrank

Die Unterteile eines Artikels können wiederum Oberteile sein, wenn sie aus weiteren Unterteilen zusammengesetzt sind. Eine Tür des obigen Küchenschranks besteht beispielsweise aus einer Holzplatte, einem Griff und zwei Scharnieren.

Eine Stückliste ist eine Liste, die aus Stücklistenpositionen besteht. Jede Stücklistenposition enthält das Unterteil, die Position und die Menge, in der das Unterteil in das Oberteil eingeht.

Stückliste des Küchenschranks:

`Oberteil`	`Unterteil`	`Positionsnummer`	`Menge`
`311 Küchenschrank`	`300 Unterschrank`	`1`	`1`
	`400 Tür`	`2`	`2`
	`200 Arbeitsplatte`	`3`	`1`

Bei der Stücklistenposition (`stck_pos`) handelt es sich um eine Beziehung auf dem Entity `artik`, wie Abbildung 5.6 zeigt.

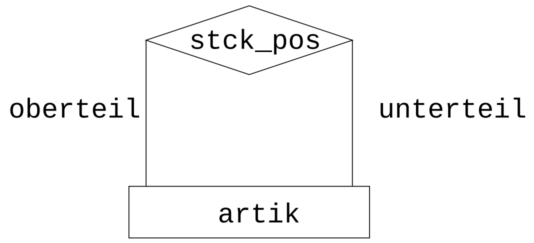

Abbildung 5.6: Modellierung von Stücklisten

Im relationalen Modell existiert *eine* Tabelle `stck_pos`, in der die einzelnen Stücklistenpositionen abgelegt sind. Die zu einem Artikel gehörige Stückliste wird im Bedarfsfall durch Selektion generiert.

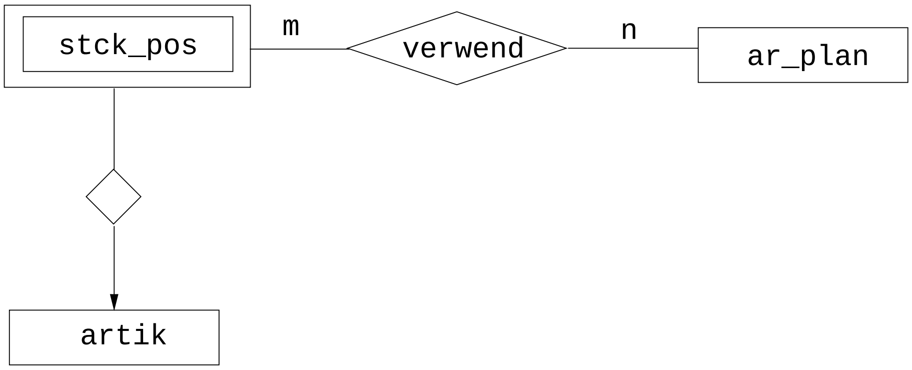

Abbildung 5.7: Weak-Entity `stck_pos`

Da eine Stückliste in einem oder mehreren Arbeitsplänen zum Einsatz kommt, hat die Stücklistenposition eine Beziehung zum Arbeitsplan. Da es eine Beziehung zu einer Beziehung im ER-Modell nicht gibt, wird aus dem Beziehungstyp `stck_pos` ein sogenanntes weak-entity (siehe Abbildung 5.7). Ein solcher schwacher Entitytyp ist immer an einen starken Entitytyp gebunden. In diesem Fall bedeutet das, daß es keine Stücklistenposition ohne den zugehörigen Artikel geben kann.

Um diesen Sachverhalt abzubilden, sind in der relationalen Datenbank folgenden vier Tabellen vorhanden. Die Tabellen enthalten hier die Daten aus dem Beispiel des Küchenschranks:

Tabelle `artik`:

Artikelnummer	Artikelbez	...
...		
311	Schrank	
300	Unterschrank	
400	Tür	
200	Arbeitsplatte	
12	Holzplatte	
25	Griff	
4711	Scharnier	
...		

Tabelle `stck_pos`:

Stueckli_pos_ID	Unterteilnummer	Oberteilnummer	Positionsnummer	...
...				
50	300	311	1	
51	400	311	2	
52	200	311	3	
53	12	400	1	
54	25	400	2	
55	4711	400	3	
...				

Tabelle `ar_plan`:

Arbeitsplan_nr	Arbeitsplanbez	Arbeitsplanbesch	...
...			

Tabelle `verwend`:

Stueckli_pos_ID	Arbeitsplan_nr
...	

Das zugehörige objektorientierte Klassenmodell besteht ebenfalls aus vier Klassen. Für die beiden Entities `ar_plan` und `artik` wird jeweils eine Klasse erzeugt. Ebenso für das schwache Entity `stck_pos`. Die zwischen den beiden Entities `stck_pos` und `ar_plan` bestehende $m : n$-Beziehung wird, wie oben beschrieben, abgebildet, indem in beiden Klassen ein zusätzliches Attribut **verwend** eingeführt wird, das jeweils auf eine Menge von Objekten der anderen Klasse verweist.

Um zu vermeiden, daß die Stückliste für einen Artikel jedesmal bei Bedarf generiert werden muß, wird eine zusätzliche Klasse `Stueckliste` angelegt. Diese besteht aus einem Oberteil, also einem Artikel, und einer Liste von Stücklistenpositionen: `list(stck_pos)`.

Der Klasse `artik` wird nur ein Attribut vom Typ `Stueckliste` hinzugefügt, da diese Klasse die Beziehung zur Klasse `stck_pos` herstellt. Die Klasse `stck_pos` erhält dagegen eine direkte Beziehung zu der Klasse `artik`. Hier wird ein Attribut unterteil eingeführt, das auf den zu der Stücklistenposition gehörigen Artikel verweist.

Die Klassendefinition in O_2 sieht demnach folgendermaßen aus:

```
class stck_pos inherit Object public type
    tuple(Stueckli_pos_ID: integer,
          Unterteilnummer: integer,
          Oberteilnr: integer,
          Positionsnummer: integer,
          Mengenwert: real,
          Vorlaufzeit: Date,
          verwend: set(ar_plan),
          unterteil: artik)
end;

class ar_plan inherit Object public type
    tuple(Arbeitsplan_nr: integer,
          Arbeitsplanbez: string,
          Arbeitsplanart: string,
          Erstellungsdatum: Date,
          Aenderungsdatum: Date,
          Login_Name: string,
          verwend: set(stck_pos))
end;

class artik inherit Object public type
    tuple(Artikelnummer: integer,
          Artikelbez: string,
          Me_einheit_kbez: string,
          Stueckliste: Stueckliste,
          ...
end;

class Stueckliste inherit Object public type
    tuple(Oberteil: artik,
```

```
        Positionen: list(stck_pos))
end;
```

5.3 Die entstandene Klassenhierarchie

Aufgrund dieser Regeln ist eine neue Klassenhierarchie für die zugrundeliegende Datenbank des PPS-Kerns entstanden. Diese wird in Abbildung 5.8 veranschaulicht.

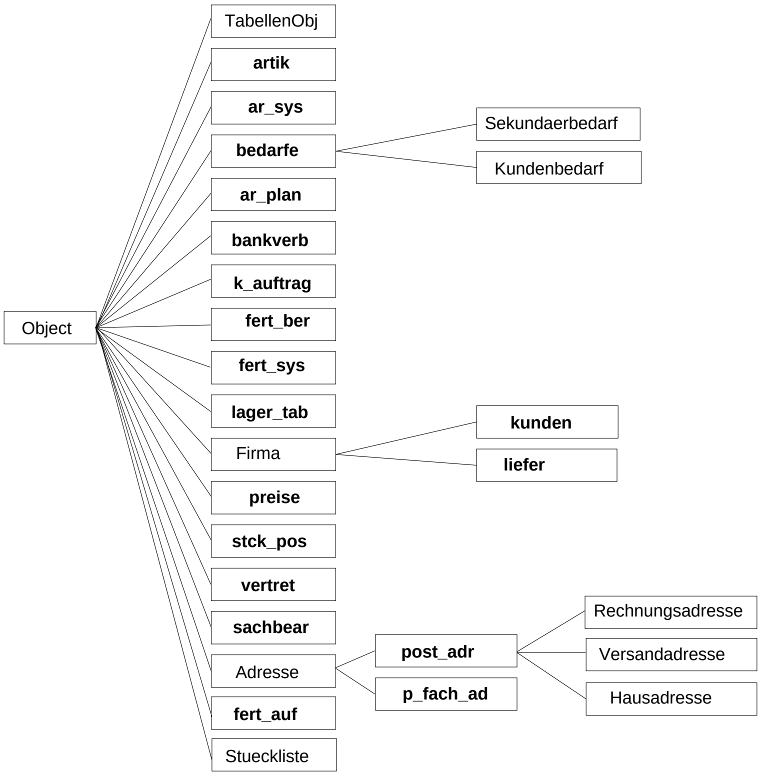

Abbildung 5.8: Klassenhierarchie

Die Subklassen der Klasse `TabellenObj` sind hier nicht mit aufgeführt, da sie nur temporär existieren. Die 27 relationalen Tabellen sind zunächst auf die 18 Klassen abgebildet worden, die in der Abbildung 5.8 hervorgehoben sind. Diese haben denselben Namen wie die zugehörigen relationalen Tabellen.

Abgesehen von den Klassen `TabellenObj` und `Stueckliste` sind sieben weitere Klassen hinzugekommen, die durch *Generalisierung* und *Spezialisierung* entstanden sind. Spezialisierung und

Generalisierung werden im ER-Diagramm als IS_A-Beziehungen dargestellt. Im objektorientierten Modell entsteht eine Vererbungshierarchie.

5.3.1 Was ist Generalisierung?

Bei der Beziehung zwischen den Entities `Kunden`, `Lieferanten` und `Firma`, wie sie in Abbildung 5.9 dargestellt ist, handelt es sich um eine *Generalisierung*.

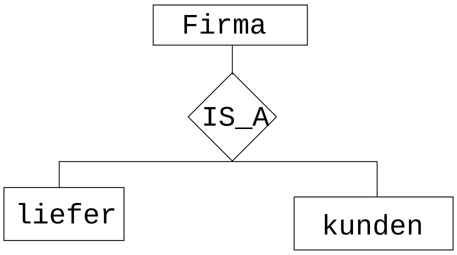

Abbildung 5.9: Beispiel einer Generalisierung

Kunden und Lieferanten haben viele gemeinsame Daten, die unter dem Entity `Firma` zusammengefaßt werden können. Im relationalen Modell existiert keine zu dem Entity `Firma` gehörige Tabelle. Im objektorientierten Modell wird eine Klasse `Firma` angelegt, die aus den gemeinsamen Attributen der Klassen `Lieferant` und `Kunde` besteht. Die Attribute, in denen sich die Klassen unterscheiden, werden den Subklassen zugeordnet.

Die Klasse `Firma` ist wie folgt definiert:

```
class Firma inherit Object public type
    tuple(Firmenname: string,
          Branche: string,
          Vers_limit: real,
          Versich_nr: string,
          Kreditlimit: real,
          bankverb: set(bankverb),
          Adresse: set(Adresse))
end;
```

Da sowohl Kunden als auch Lieferanten in Relation zu den Klassen `Adresse` und `bankverb` stehen, wird diese Beziehung ebenfalls über die Superklasse `Firma` abgebildet. Es handelt sich in beiden Fällen um eine $1:n$-Beziehung, daher wird bei der Klasse `Firma` jeweils ein Attribut vom Datentyp `set()` erzeugt.

```
class kunden inherit Firma public type
    tuple(Kundennr: integer,
          Skontoproz: real,
          Skontotage: integer,
          saldo: real,
          betreu: set(vertret),
          k_auftrag: set(k_auftrag))
end;

class liefer inherit Firma public type
    tuple(Lieferantennummer: integer)
end;
```

5.3.2 Was ist Spezialisierung?

Spezialisierung bedeutet Aufsplitten einer Klasse in mehrere Subklassen.

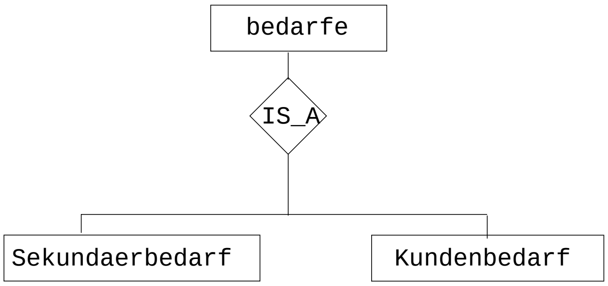

Abbildung 5.10: Beispiel einer Spezialisierung

Bedarfe werden unterschieden in Kundenbedarfe und Sekundärbedarfe. Da sich diese Bedarfe in ihren Attributen nicht unterscheiden, existiert im relationalen Modell nur *eine* Tabelle, in der beide Bedarfe abgelegt werden. Unterschieden werden die Bedarfe anhand eines zusätzlich eingeführten Attributs `Bedarfstyp`.

Im objektorientierten Modell wird eine Klasse erzeugt, die dieselben Attribute hat wie die relationale Tabelle. Diese Klasse `bedarfe` sieht wie folgt aus:

```
class bedarfe inherit Object public type
    tuple(Bedarfs_nr: integer,
          Artikelnummer: integer,
          Termin: Date,
```

```
          Mengenwert: real,
          Bedarfstyp: string,
          artik: artik)
end;
```

Zusätzlich werden zwei Subklassen `Kundenbedarf` und `Sekundaerbedarf` angelegt. Das Attribut, anhand dessen die Bedarfe im relationalen Modell unterschieden werden, wird in diesem Modell zwar nicht mehr benötigt, bleibt aber, genau wie die Schlüsselattribute der Tabellen, erhalten.

Sekundärbedarfe und Kundenbedarfe unterscheiden sich in ihren Beziehungen zu anderen Entities, daher unterscheiden sich die beiden Klassen in ihren Attributen.

```
class Kundenbedarf inherit bedarfe public type
    tuple(k_auf_pos: set(k_auftrag))
end;

class Sekundaerbedarf inherit bedarfe public type
    tuple(fert_bed: set(fert_auf))
end;
```

Ein Kundenbedarf hat eine Beziehung zu einem oder mehreren Kundenaufträgen, während ein Sekundärbedarf in Relation zu verschiedenen Fertigungsaufträgen stehen kann. Alle anderen Attribute sind gleich und werden in der Klasse `bedarfe` abgebildet.

5.3.3 Beispiel einer Generalisierung und Spezialisierung

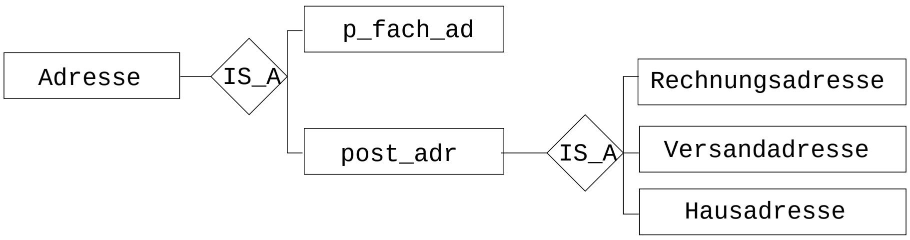

Abbildung 5.11: Beispiel einer Generalisierung und Spezialisierung

Bei den Adressen findet, wie Abbildung 5.11 zeigt, sowohl eine Generalisierung als auch eine Spezialisierung statt. Im Zuge der Generalisierung werden die gemeinsamen Attribute der Klassen `post_adr` und `p_fach_ad` in der Superklasse `Adresse` zusammengefaßt.

```
class Adresse inherit Object public type
    tuple(Adress_nr: integer,
          Kundennr: integer,
          Lieferantennummer: integer,
          Ort: string,
          Firma: Firma)
end;
```

Da sowohl Postadressen als auch Postfachadressen in Relation zu Kunden und Lieferanten stehen, wird diese Beziehung ebenfalls über die Superklasse `Adresse` abgebildet. Eine Adresse gehört immer zu einem Kunden oder einem Lieferanten. Da jedes Objekt der Klassen `kunden` und `liefer` auch ein Objekt der Klasse `Firma` ist, wird die Relation über einen Zeiger auf ein Objekt der Klasse `Firma` realisiert.

Die Attribute, in denen sich die Postadressen und die Postfachadressen unterscheiden, werden den Subklassen zugeordnet.

```
class p_fach_ad inherit Adresse public type
    tuple(Postfach: integer,
          Postfachleitzahl: integer)
end;


class post_adr inherit Adresse public type
    tuple(adress_typ: string,
          Strasse: string,
          Postleitzahl: integer,
          Telefonnummer: string,
          Telefaxnummer: string)
end;
```

Postadressen werden unterschieden in Hausadressen, Versandadressen und Rechnungsadressen. Hierbei handelt es sich um logische Unterscheidungen, d.h. die Entities unterscheiden sich nicht in ihren Attributen. Relational wird diese Unterscheidung daher anhand des Attributs `Adresstyp` vorgenommen, während im objektorientierten Modell drei unterschiedliche Subklassen eingeführt werden:

```
class Hausadresse inherit post_adr public
end;

class Versandadresse inherit post_adr public
end;
```

```
class Rechnungsadresse inherit post_adr public
end;
```

In diesem Beispiel erhalten die erzeugten Subklassen kein zusätzliches Attribut.

Kapitel 6

Die objektorientierte Datenbank

Bis hierher befinden sich die Daten noch in den Tabellenobjekten. Nachdem die Klassenhierarchie festgelegt worden ist und die Klassen in O_2 halbautomatisch angelegt worden sind, müssen die Daten aus den Tabellenobjekten nun in Objekte der jeweiligen Klasse überführt werden. Dabei wird für jeden Eintrag in dem Tabellenobjekt ein Objekt der neuen Klasse erzeugt und mit den Daten aus dem Tabellenobjekt gefüllt. Abbildung 6.1 veranschaulicht diesen Sachverhalt: Hier entstehen aus zwei Tupeln des Tabellenobjekts der Klasse `T_kunden` zwei Objekte der Klasse `kunden`.

Die Beziehungen, die zwischen den einzelnen Objekten bestehen, können erst hergestellt werden, wenn alle Objekte erzeugt worden sind. Eine Relation, die beispielsweise zwischen einem Kunden und einem Auftrag besteht, kann erst eingetragen werden, wenn beide Objekte existieren. Daher werden zunächst alle Objekte ohne Relationen erzeugt und in der Datenbank abgelegt. In einem zweiten Schritt werden dann die Beziehungen, die zwischen diesen Objekten bestehen, hergestellt.

6.1 Erzeugen der Objekte

Damit das Erzeugen der Objekte vollautomatisch funktioniert, müssen folgende Voraussetzungen erfüllt sein:

1. Die Attribute der Klassen haben dieselben Namen und denselben Datentyp wie die Attribute der Tabellen.
2. Die Anzahl der Attribute stimmt überein.

Beides wurde beim Anlegen der Klassen berücksichtigt.

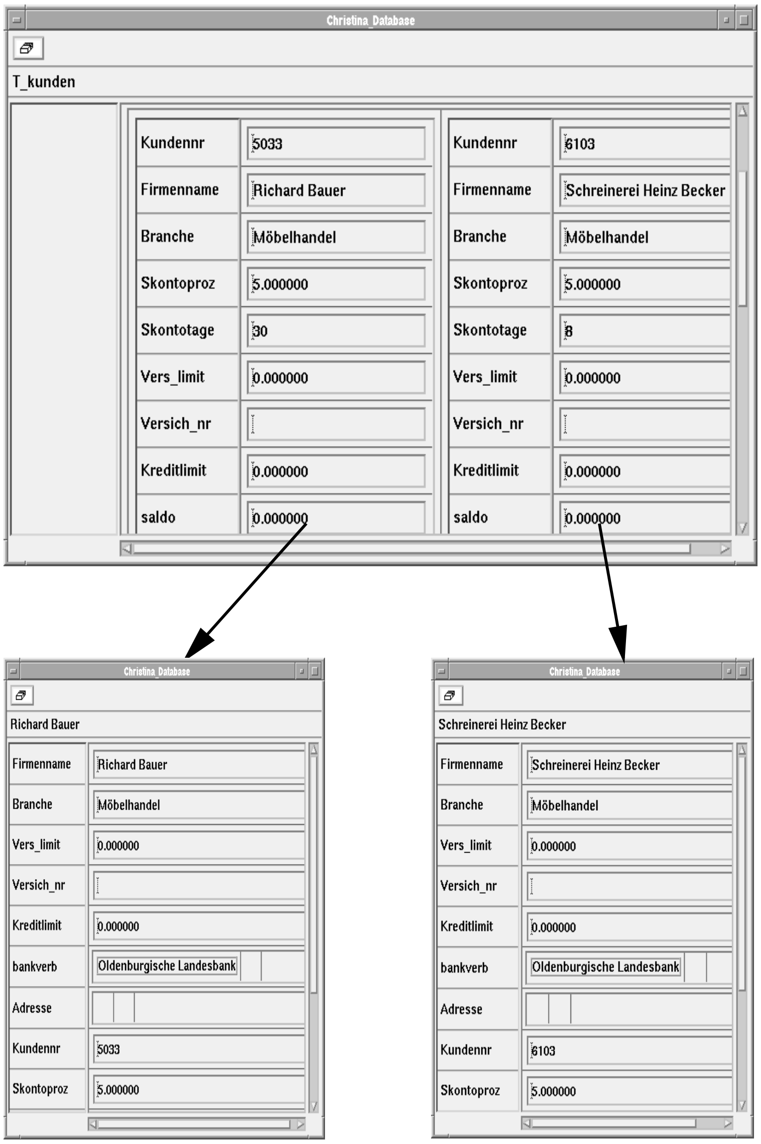

Abbildung 6.1: Aus den Tupeln eines Tabellenobjekts Objekte erzeugen

6.1.1 Persistente Objekte anlegen

Die Lebensdauer eines Objekts endet, wenn das Programm, von dem es erzeugt wurde, terminiert. Die Objekte, die hier entstehen, sollen jedoch über das Ende des Erzeugerprogramms hinaus Bestand haben.

Die Möglichkeit, Objekte dauerhaft in einer Datenbank zu speichern, wird als *Persistenz* bezeichnet. O_2 erlaubt dem Benutzer, Objekte oder Werte *persistent* zu machen, indem er ihnen einen *Namen* gibt. Sie werden in O_2 als *named objects* bzw. *named values* bezeichnet.

Allgemein gelten in O_2 die folgenden Persistenzregeln [DLR95]:

1. Objekte oder Werte, die einen Namen besitzen, sind persistent.
2. Objekte oder Werte, die von einem persistenten Objekt referenziert werden, sind persistent.

Ein solcher Name ist weder eine Klasse noch ein Objekt. Es ist nur ein Name, der mit einem Objekt oder einer Menge von Objekten assoziiert wird. Namen können z.B. folgendermaßen erzeugt werden:

```
create name Bauer: kunden;
create name PPS_kunden: set(kunden);
```

Im ersten Fall existiert *höchstens ein* persistentes Objekt der Klasse `kunden`. Im zweiten Fall wird jedes Objekt dieser Klasse persistent, wenn es der Menge `PPS_kunden` hinzugefügt wird. In diesem Beispiel gilt für den Namen `Bauer` die erste Persistenzregel, während der Name `PPS_kunden` unter die zweite Regel fällt.

Vor dem Erzeugen der einzelnen Objekte müssen Namen existieren, mit deren Hilfe die Objekte persistent gemacht werden können. Für jede Klasse wird ein Name erzeugt, der eine Menge von Objekten der jeweiligen Klasse persistent speichern kann. Es entstehen Namen vom Datentyp `set()` analog zu dem Namen `PPS_kunden` in obigem Beispiel.

Für die Klassen, die im objektorientierten Datenmodell durch Spezialisierung bzw. Generalisierung hinzugekommen sind, wird kein solcher Name erzeugt, da diese Klassen beim Erzeugen der Objekte zunächst unberücksichtigt bleiben.

Das Erzeugen der persistenten Objekte übernimmt die Methode `create_named_Object` der Klasse `TabellenObj`. Die Methode wird für alle persistenten Objekte der Subklassen von `TabellenObj` aufgerufen. Zunächst wird überprüft, ob zu dem in `self->classname` abgelegten String eine zugehörige Klasse definiert ist. Existiert eine Klasse dieses Namens, so wird ein *named object* erzeugt. Dieser Name setzt sich aus dem String „`PPS_`" und dem Namen der Klasse zusammen.

Die Methode `create_named_Object` ist wie folgt implementiert:

```
method body create_named_Object in class TabellenObj
{
        o2 string cmd;          /* string, der auf der Shell ausgefuehrt wird */
        o2 Meta_definition def; /* Objekt des Meta-Schemas */
```

```
        transaction;
        /* ueberpruefen, ob es sich bei diesem Tabellenobjekt um
           eine Zuordnungstabelle handelt */
        def =Schema->definition(self->classname);
        if (def != nil)
        {
                /* Es existiert eine zugehoerige Klasse, fuer die
                   ein "named object" erzeugt werden kann.
                   Der Name des Objekts wird aus dem string "PPS_" und dem
                   Klassennamen gebildet. Der Datentyp ist set().*/
                cmd = "create name PPS_" + self->classname
                         + ": set(" + self->classname + ");";

                /* string zur Kontrolle auf der Shell ausgeben. */
                printf("%s\n",cmd);
                transaction;

                /* Kommando ausfuehren mit Hilfe des Meta-Schemas */
                Schema->command( cmd );
                validate;
        }
};
```

6.1.2 Objekte erzeugen und Daten übertragen

Nun können die Objekte erzeugt und persistent gemacht werden. Beim Erzeugen der Objekte wird folgendermaßen vorgegangen:

- Durchlaufen aller persistenten Tabellenobjekte.
- Falls es sich nicht um eine Zuordnungstabelle handelt, für jeden Eintrag in der Tabelle ein Objekt erzeugen und persistent ablegen.

Bei jedem persistenten Tabellenobjekt wird die Methode `attach` (siehe Anhang B.1) aufgerufen. Diese Methode überprüft zunächst, ob die vorliegende Tabelle nur Zuordnungen zwischen anderen Tabellen vornimmt. Dazu wird die Methode `is_class` aufgerufen. Zuordnungstabellen werden erst beim Eintragen der Relationen benötigt. Sie spielen zunächst keine Rolle.

Die Methode `is_class` überprüft die *Existenz* einer Klasse mit dem Namen, der in dem Attribut **self->classname** abgelegt ist. Existiert zu diesem Namen keine Klasse, so handelt es sich um eine Zuordnungstabelle. Die Methode liefert in diesem Fall `false` zurück. Das Überprüfen der Existenz einer Klasse wird von einem Objekt der Klasse `Meta_definition` übernommen. Der `definition`-Methode dieser Klasse wird ein String, beispielsweise ein Klassenname, übergeben. Sie liefert ein Metaklassenobjekt zurück, falls eine Klasse oder ein Typ dieses Namens definiert ist, sonst wird `nil` zurückgegeben.

Handelt es sich um eine Zuordnungstabelle, terminiert die Methode `attach`. Andernfalls wird ein Programm generiert, welches die Tabelle abarbeitet und für jedes Tupel ein Objekt erzeugt. Das Programm wird anschließend mit Hilfe der `command`-Methode der Klasse `Meta` auf der O_2-Shell ausgeführt. Für ein Objekt der Klasse `T_kunden` sieht das generierte Programm wie folgt aus:

```
run body{
o2 kunden obj;
o2 tuple(Kundennr:integer,Firmenname:string,
 Branche:string,Skontoproz:real,
 Skontotage:integer,Vers_limit:real,
 Versich_nr:string,Kreditlimit:real,
 saldo:real) eintrag;

for(eintrag in tmp_T_kunden->tabelle)
{ obj = new kunden;
        obj->Firmenname=eintrag.Firmenname;
        obj->Branche=eintrag.Branche;
        obj->Vers_limit=eintrag.Vers_limit;
        obj->Versich_nr=eintrag.Versich_nr;
        obj->Kreditlimit=eintrag.Kreditlimit;
        obj->Kundennr=eintrag.Kundennr;
        obj->Skontoproz=eintrag.Skontoproz;
        obj->Skontotage=eintrag.Skontotage;
        obj->saldo=eintrag.saldo;
PPS_kunden+= set(obj);}
}
```

Das Erzeugen der Objekte wird von den Tabellenobjekten gesteuert, d.h., ein Tabellenobjekt arbeitet „seine“ Tabelle sukzessive ab und erzeugt für jeden Eintrag ein Objekt. Auf diese Weise ist einerseits gewährleistet, daß für jeden Eintrag ein Objekt entsteht und andererseits, daß das Programm terminiert. Dies ist nicht gegeben, wenn sich jedes Objekt beim Erzeugen die Daten aus dem zugehörigen Tabellenobjekt „holt“.

6.1.3 Objekte der Subklassen erzeugen

Die Daten befinden sich jetzt in den Objekten einer der zuerst entstandenen 18 Klassen der Klassenhierarchie (siehe Abbildung 5.8). Die durch Generalisierung und Spezialisierung hinzugekommenen Klassen sind bisher unberücksichtigt geblieben. Von den durch Generalisierung entstandenen Superklassen werden auch keine Instanzen erzeugt. Um Objekte der Subklassen, die durch Spezialisierung entstanden sind, zu erzeugen, wird bei allen Tabellenobjekten die Methode `create_subclass_objects` aufgerufen. Diese Methode überprüft, ob die gegebene Klasse Subklassen besitzt. Ist dies der Fall, müssen aus den Objekten der Klasse ggf. Objekte der Subklasse erzeugt werden.

Zum Erzeugen der Objekte der Subklasse existiert bei den Klassen, die Subklassen haben, die Methode `create_subclass`. Anhand dieser Methode wird „entschieden“, von welcher Subklasse ein Objekt erzeugt werden muß. Bei der Klasse `bedarfe` sieht diese Methode wie folgt aus:

```
method body create_subclass in class bedarfe
{
/* im relationalen Modell werden Sekundaerbedarf
   und Kundenbedarf anhand des Attributs Bedarfstyp
   unterschieden */
if(self->Bedarfstyp == "k")
{
/* bei Bedarfstyp == "k" handelt es sich
   um einen Kundenbedarf */
return  new Kundenbedarf(self);
}
else
{
/* bei Bedarfstyp == "s" handelt es sich
   um einen Sekundaerbedarf */
return  new Sekundaerbedarf(self);
}
};
```

Nach dem Aufruf von `create_subclass` bei den Objekten der Klasse `bedarfe` besteht die Menge der persistenten Objekte `PPS_bedarfe` nur noch aus Objekten der Klassen `Sekundaerbedarf` und `Kundenbedarf`. Bei den Klassen `Kundenbedarf` und `Sekundaerbedarf` wurde die `init`-Methode überschrieben. Die `init`-Methode einer Klasse wird automatisch aufgerufen, wenn ein neues Objekt der Klasse mit `new` erzeugt wird. Erwartet diese Methode Parameter, so sind diese beim Aufruf von `new` mit anzugeben. In diesem Fall wird ein Objekt der Klasse `Bedarfe` als Parameter übergeben. Die Werte des Objekts, das übergeben wurde, werden bei dem neuen

Objekt eingetragen. Abbildung 6.2 zeigt die Menge `PPS_bedarfe` vor und nach dem Aufruf der Methode `create_subclass`.

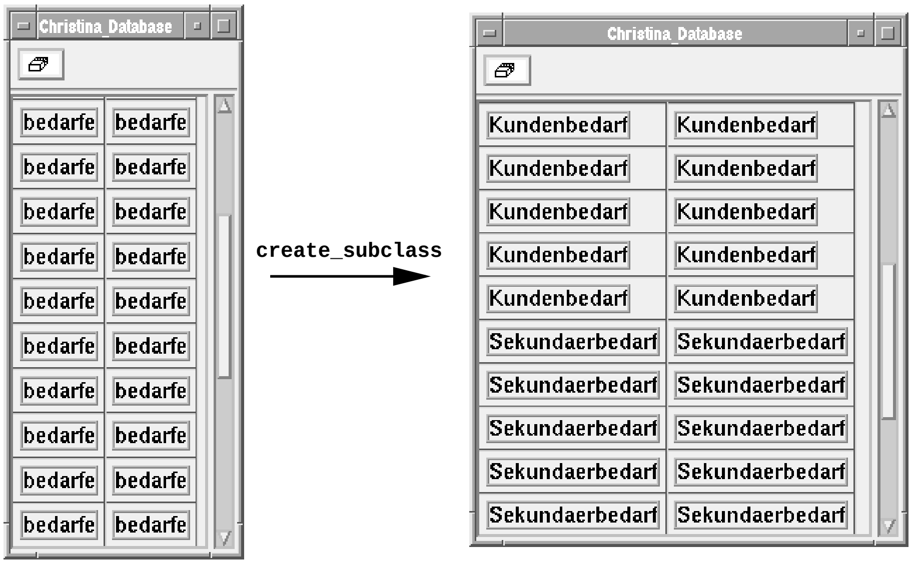

Abbildung 6.2: Erzeugung von Subklassenobjekten am Beispiel von Bedarfen

6.2 Eintragen der Relationen

Die Objekte, die sich nun in der objektorientierten Datenbank befinden, enthalten zwar die Daten der relationalen Tabellen, mit Ausnahme der Zuordnungstabellen, aber die Relationen, die zwischen den Objekten bestehen, sind noch nicht in der objektorientierten Datenbank abgebildet. Abbildung 6.3 zeigt ein Objekt der Klasse `kunden` und ein Objekt der Klasse `k_auftrag`, bei denen die Relationen noch nicht eingetragen sind.

Bei dem Objekt der Klasse `k_auftrag` ist die Kundennr 5033 eingetragen, d.h., die beiden abgebildeten Objekte stehen zueinander in Beziehung. Beim Eintragen der Relationen muß dafür gesorgt werden, daß das Attribut `kunden` der Klasse `k_auftrag` eine Referenz auf das dargestellte Objekt der Klasse `kunden` erhält. Umgekehrt ist zu der Menge der Kundenaufträge `k_auftrag` eine Referenz zu dem dargestellten Objekt der Klasse `k_auftrag` herzustellen.

Das Eintragen der Relationen wird ebenfalls von den Tabellenobjekten gesteuert. Ein Tabellenobjekt kann anhand der Liste der Fremdschlüssel feststellen, welche Tabellen in Relation zueinander stehen. Beim Eintragen der Relationen läuft daher folgendermaßen ab:

1. Tabellen mit Fremdschlüsseln noch einmal bearbeiten.

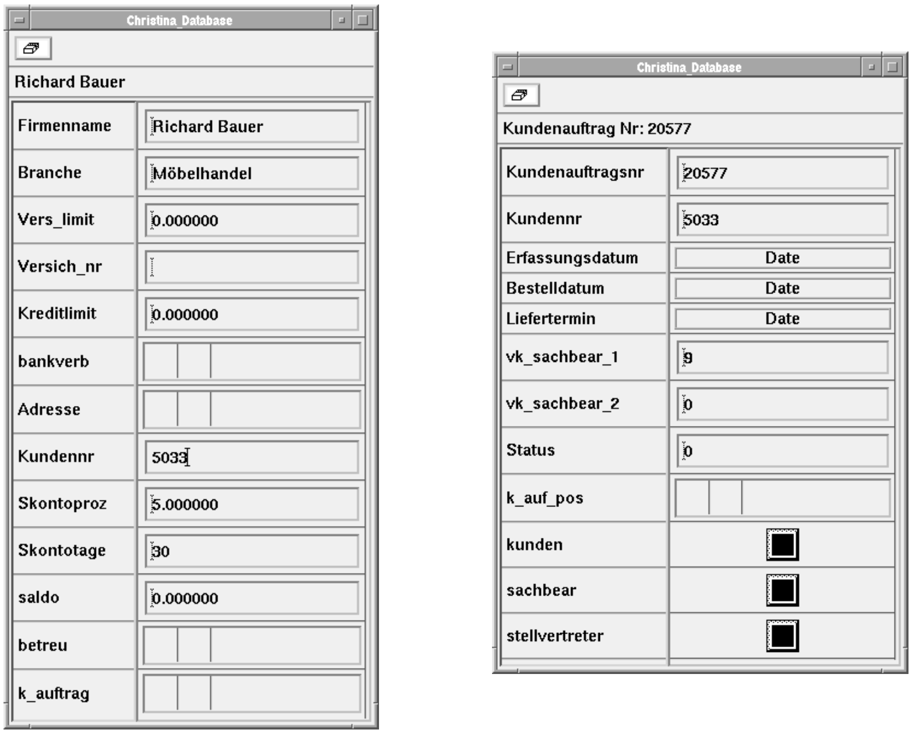

Abbildung 6.3: Relationen eintragen bei `kunden` und `k_auftrag`

2. Für jeden Eintrag in der Tabelle die zugehörigen Objekte suchen.
   (Die Objekte werden dabei über die erhalten gebliebenen Schlüsselattribute identifiziert.)
3. Durch Methodenaufruf bei den Objekten die Beziehungen herstellen.

Zum Eintragen der Relationen wird bei den Tabellenobjekten, deren Liste der Fremdschlüssel nicht leer ist, eine Methode generiert, die später zum Erzeugen der Relationen nur aufgerufen werden muß. Zum Generieren dieser Methoden werden die in der Klasse `TabellenObj` implementierten Methoden `define_insert_references` und `def_insert_for_class` benutzt (Siehe Anhang B.2 und B.3).

6.2.1 Die Methode `insert_references` der Tabellenobjekte

Die Methode, die Relationen zwischen den Objekten der Klassen herstellen soll, wird mit `insert_references` bezeichnet.

Auch hier muß zwischen Zuordnungstabellen und anderen Tabellen unterschieden werden. Zuordnungstabellen werden einmal durchlaufen, unabhängig von der Anzahl der Fremdschlüssel.

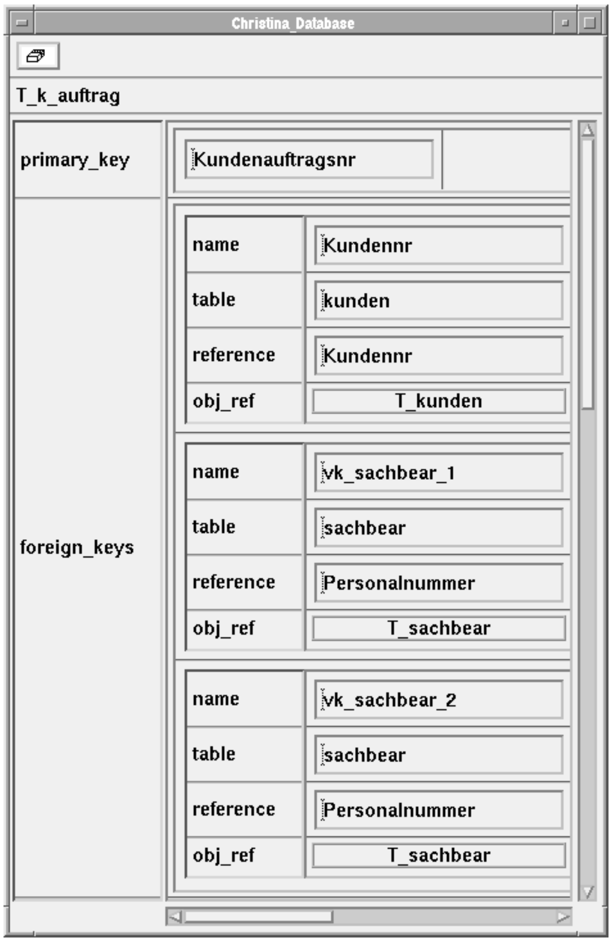

Abbildung 6.4: Fremdschlüssel der Klasse `k_auftrag`

Bei anderen Tabellen entspricht die Anzahl der Durchläufe der Anzahl der Fremdschlüssel. Die Abbildung 6.4 zeigt das Tabellenobjekt `tmp_T_k_auftrag`, das drei Fremdschlüssel enthält. Diese Tabelle wird beim Erzeugen der Relationen dreimal durchlaufen.

Bei den Tabellen `k_auftrag` und `kunden` handelt es sich nicht um Zuordnungstabellen. Da die Liste der Fremdschlüssel des Objekts `tmp_T_kunden` leer ist, wird diese Tabelle beim Erzeugen der Relationen nicht bearbeitet. Die Beziehungen, die zwischen den Objekten der beiden Klassen bestehen, werden von dem Objekt `tmp_T_k_auftrag` erzeugt. Außerdem stellt dieses Objekt die Beziehungen eines Kundenauftrags zu einem Sachbearbeiter und ggf. dessen Stellvertreter her.

Das Erzeugen dieser drei Beziehungen wird von der Methode `insert_references` übernommen, die für die Klasse `T_k_auftrag` generiert wird. Eine solche Methode wird für alle Tabellenobjekte erzeugt, deren Liste der Fremdschlüssel nicht leer ist. Bei Zuordnungstabellen wird *nur diese* Methode erzeugt, so daß diese nur einmal durchlaufen werden. Handelt es sich, wie bei `T_k_auftrag`, nicht um eine Zuordnungstabelle, wird für jeden vorhandenen Fremdschlüssel eine weitere Methode generiert, die für das Eintragen der jeweiligen Relation sorgt. Diese Methoden werden von der Methode `insert_references` aufgerufen. Die Namen dieser zusätzlich generierten Methoden setzen sich zusammen aus dem String „`insert_`" und dem Namen des Attributs, das die Relation zwischen den Klassen abbildet. Die Methode `insert_references` der Klasse `T_k_auftrag` sieht wie folgt aus:

```
method body insert_references in class T_k_auftrag
{self->insert_kunden;
self->insert_sachbear;
self->insert_stellvertreter;
};
```

Die Methode `insert_references` ruft drei weitere Methoden auf, die alle ähnlich strukturiert sind. Die Methode `insert_kunden` erzeugt die Relationen, die zwischen Kunden und deren Aufträgen bestehen:

```
method body insert_kunden in class T_k_auftrag
{
o2 k_auftrag obj1;
o2 set(k_auftrag) first_result;
o2 kunden obj2;
o2 set(kunden) second_result;
o2 tuple(Kundenauftragsnr:integer,Kundennr:integer,
 Erfassungsdatum:Date,Bestelldatum:Date,Liefertermin:Date,
 vk_sachbear_1:integer,vk_sachbear_2:integer,
 Status:integer)eintrag;

for(eintrag in self->tabelle)
{
o2query(first_result,"select x from x in PPS_k_auftrag
where x.Kundenauftragsnr=$1",eintrag.Kundenauftragsnr);

o2query(second_result,"select x from x in PPS_kunden
where x.Kundennr=$1",eintrag.Kundennr);

if((count(first_result)==1)&&(count(second_result)==1))
{
```

```
obj1 = element(first_result);
obj2 = element(second_result);

transaction;
obj1->set_kunden(obj2);
obj2->set_k_auftrag(obj1);
validate;
}
}
};
```

Diese Methode durchläuft alle Einträge eines Objekts der Klasse `T_k_auftrag`. Für jedes Tupel werden die zugehörigen Objekte der Klassen `kunden` und `k_auftrag` in der Datenbank gesucht. Die Objekte werden anhand der Kundennr bzw. der Kundenauftragsnummer identifiziert. Bei beiden Attributen handelt es sich um den Primärschlüssel aus der relationalen Datenbank. Diese Primärschlüssel werden in der objektorientierten Datenbank nicht unbedingt gebraucht, da die Objekte über eine interne Objekt_ID identifiziert werden. Zum Erzeugen der Relationen werden die Primärschlüssel allerdings benötigt. Relational wurden die Beziehungen, die zwischen den Entities bestehen, über Schlüsselattribute abgebildet. Da die Relationen nicht direkt beim Erzeugen der Objekte aus den Tabellenobjekten eingetragen werden können, müssen die Objekte, zwischen denen die Relation besteht, mit Hilfe der Primärschlüssel in der Datenbank gesucht werden.

Das Suchen von Objekten in einer Datenbank geschieht mittels einer Anfragesprache, die in O_2 den Namen *OQL* trägt. *OQL* steht für *object oriented query language* und ist eine SQL-ähnliche objektorientierte Abfragesprache. *OQL* kann als eingebettete Funktion in einer Programmiersprache und andererseits als ad-hoc Abfragesprache eingesetzt werden. Die Funktion `o2query` erlaubt es dem Benutzer, jede beliebige Anfrage an die Datenbank in den Programmcode einzubinden.

In obigem Beispiel wird die Funktion `o2query` dazu benutzt, die zusammengehörigen Objekte in der Datenbank zu suchen:

```
o2query(first_result,"select x from x in PPS_k_auftrag
where x.Kundenauftragsnr=$1",eintrag.Kundenauftragsnr);
```

Dieser erste Aufruf dient dazu, den Kundenauftrag in der Datenbank zu suchen. Dabei wird der Schlüssel des Kundenauftrags, nach dem gesucht werden soll, in `eintrag.Kundenauftragsnr` übergeben. Alle Objekte der Klasse `k_auftrag`, die in der Datenbank gefunden wurden, werden in der Variablen `first_result` zurückgeliefert. In einer zweiten `o2query` wird dann der zugehörige Kunde in der Datenbank gesucht.

Anschließend wird überprüft, ob zu jedem Schlüssel ein Objekt existiert. Existiert zu einem der Schlüssel kein Objekt oder existieren mehrere Objekte mit diesem Schlüssel, wird bei keinem der Objekte die Relation eingetragen. In diesem Fall ist die *referentielle Integrität* nicht gegeben, d.h., eigentlich dürften solche Einträge in der relationalen Datenbank nicht enthalten sein. An dieser Stelle besteht die Möglichkeit, Fehlermeldungen auszugeben und eine Fehlerkorrektur in der Datenbank durchzuführen.

Ist zu jedem Schlüssel genau ein Objekt der Klasse gefunden worden, geschieht das Eintragen der Relation durch einen Methodenaufruf bei jedem der gefundenen Objekte.

In obigem Beispiel sieht der Methodenaufruf wie folgt aus:

```
obj1->set_kunden(obj2);
obj2->set_k_auftrag(obj1);
```

`obj1` ist ein Objekt der Klasse `k_auftrag`. Bei diesem Objekt wird die Methode `set_kunden` aufgerufen, die als Parameter ein Objekt der Klasse `kunden` erwartet. Die Methode „trägt beim Kundenauftrag den Kunden ein“.

```
method body set_kunden(obj:kunden) in class k_auftrag
{
self->kunden = obj;
};
```

Da jeder Kundenauftrag zu genau einem Kunden gehört, muß hier die Relation nur eingetragen werden. Die Methode `set_k_auftrag` der Klasse `kunden` sieht etwas anders aus:

```
method body set_k_auftrag(obj: k_auftrag) in class kunden
{
if (!(obj in self->k_auftrag))
{self->k_auftrag += set(obj);
}
};
```

In diesem Fall wird der Kundenauftrag der Menge der Kundenaufträge eines Kunden hinzugefügt. Allerdings nur, wenn er nicht schon in der Menge enthalten ist.

Die Methoden `set_kunden` und `set_k_auftrag` werden ebenfalls generiert. Das Generieren dieser Methoden wird von den Tabellenobjekten `tmp_T_kunden` und `tmp_T_k_auftrag` übernommen.

6.2.2 Generieren der Methoden `set_`<*Attributname*>

Die Methode `define_set_attr` generiert, falls es zu dem Tabellenobjekt eine Klasse gibt, für jedes Attribut dieser Klasse eine Methode, die dem Attribut einen Wert zuweist. Falls es Subklassen gibt, wird auch bei den Subklassen für jedes Attribut eine solche Methode erzeugt.

Das Generieren der Methoden wird hier am Beispiel der Klasse `kunden` erläutert:

```
class kunden inherit Firma public type
    tuple(Kundennr: integer,
          Skontoproz: real,
          Skontotage: integer,
          saldo: real,
          betreu: set(vertret),
          k_auftrag: set(k_auftrag))
end;
```

Für jedes Attribut dieser Klasse wird eine Methode mit dem Namen `set_`<*Attributname*> erzeugt. Zu den Attributen der Klasse zählen auch die, die in der Klassendefinition selbst nicht enthalten sind, da sie von der Superklasse `Firma` vererbt wurden. Zu letzteren gehört beispielsweise das Attribut Firmenname. Da in der Superklasse keine Methoden zum Setzen der Attribute definiert sind, müssen sie in der Subklasse implementiert werden. Für die Klasse `kunden` werden daher die folgenden Methoden generiert:

```
method public set_Firmenname(obj: string) in class kunden;
method public set_Branche(obj: string) in class kunden;
method public set_Vers_limit(obj: real) in class kunden;
method public set_Versich_nr(obj: string) in class kunden;
method public set_Kreditlimit(obj: real) in class kunden;
method public set_Kundennr(obj: integer) in class kunden;
method public set_Skontoproz(obj: real) in class kunden;
method public set_Skontotage(obj: integer) in class kunden;
method public set_saldo(obj: real) in class kunden;
method public set_Adresse(obj: Adresse) in class kunden;
method public set_betreu(obj: vertret) in class kunden;
method public set_k_auftrag(obj: k_auftrag) in class kunden;
method public set_bankverb(obj: bankverb) in class kunden;
```

Jede dieser Methoden bekommt als Parameter eine Variable, die dem Datentyp des betreffenden Attributs entspricht. Die Signaturen der Methoden unterscheiden sich nur in dem Namen und in dem Datentyp des Parameters.

Wie der Rumpf der Methode aussieht, hängt vom Datentyp des Attributs ab. Handelt es sich um einen einfachen Datentyp, wie beispielsweise `integer` oder `string`, so wird das Attribut nur auf den übergebenen Wert gesetzt.

```
method body set_Kundennr in class kunden
{
self->Kundennr = obj;
};
```

Das gilt auch, wenn das Attribut vom Datentyp `tuple()` ist oder auf ein Objekt einer Klasse zeigt.

In obigem Beispiel sehen nur die letzten vier Methoden anders aus, da die Attribute vom Typ `set()` sind.

```
method body set_k_auftrag(obj: k_auftrag) in class kunden
{
if (!(obj in self->k_auftrag))
{self->k_auftrag += set(obj);
}
};
```

Zum Generieren dieser Methoden sind in der Klasse `TabellenObj` die Methoden `define_set_attr` und `redefine_set_attr` (siehe Anhang B.4) definiert. Für alle persistenten Tabellenobjekte wird zunächst die Methode `define_set_attr` aufgerufen. Falls zu dem Tabellenobjekt eine Klasse existiert, wird mit Hilfe des Metaklassenobjekts der Klasse die Struktur der Klasse abgefragt. Für jedes Attribut wird eine Funktion aufgerufen, die die entsprechende Methode generiert.

```
method body define_set_attr in class TabellenObj
{
    o2 Meta_definition def;
    o2 Meta_class m,sub;
    o2 tuple(name: string, visibility: string, type: Meta_definition) att;

    /* Existiert eine Klasse oder ein Typ dieses Namens? */
    def = Schema->definition(self->classname);
    if(def != nil)
    {
        /* Handelt es sich um einen Typ bricht die Methode ab */
        if(! def->is_class)
```

```
        {
            printf("Der ist Klassenname falsch !\n");
            abort;
        }
        else
        {
            /* Da es sich um eine Klasse handelt ist def ein
               Objekt der Klasse Meta_class */
            m = (o2 Meta_class) def;

            /* Fuer jedes Attribut der Klasse wird eine Methode erzeugt.*/
            for (att in m->attributes)
            {
                create_method(att, self->classname);
            }
            /* Alle Subklassen (falls vorhanden) werden durchlaufen */
            for(sub in m->subclasses)
            {
                for (att in sub->attributes)
                {
                /* Fuer jedes Attribut der Subklasse wird eine Methode erzeugt.*/
                    create_method(att, sub->name);
                }
            }
        }
    }
};
```

Wie bereits in Abschnitt 4.4 auf Seite 19 erläutert, wird zum Generieren von Methoden ein <*commandstring*> erzeugt, der mit Hilfe der `command`-Methode der Klasse `Meta` ausgeführt wird. Dies wird von der Funktion `create_method` übernommen (siehe Anhang C.1.). Diese erwartet als Parameter einerseits den Namen der Klasse, für die die Methode generiert werden soll, und andererseits eine Variable vom Typ `tuple()`, die unter anderem den Namen und den Datentyp des Attributs beinhaltet. Die Komponente `type` dieses Tupels beinhaltet den Datentyp des Attributs. Dieser bestimmt den Rumpf der zu generierenden Methode. Handelt es sich um ein Objekt einer Klasse oder einen einfachen Datentyp, wird folgender String erzeugt:

```
impl =  "self->" + att.name;
impl += " = obj;\n";
```

Dies gilt auch, wenn das Attribut vom Typ `tuple()` ist.

Anschließend wird für alle persistenten Tabellenobjekte die Methode `redefine_set_attr` aufgerufen (siehe Anhang B.4). Die Methode redefiniert eine Methode `set_`<*Attributname*> gegebenenfalls durch Aufruf der Funktion `redefine_method` (siehe Anhang C.2). Das Redefinieren einer Methode ist bei $m : n$-Beziehungen mit zusätzlichem Attribut notwendig, wie sie in Abschnitt 5.2.3 auf Seite 29 beschrieben worden sind. Bei der Klassendefinition wurde das Attribut nur in eine der beiden Klassen aufgenommen, dies muß beim Eintragen der Relationen von der Methode `set`<*Attributname*> berücksichtigt werden.

Innerhalb der Methode `insert_references` bzw. innerhalb einer der von ihr aufgerufenen Methoden, wird die Beziehung zwischen zwei Objekten einer Klasse hergestellt, indem bei beiden Objekten eine Methode aktiviert wird. Die aktivierte Methode kann entweder `set_`<*Attributname*> oder `ins_`<*Attributname*> heißen.

6.2.3 Benutzerdefinierte Methoden `ins_`<*Attributname*>

Im allgemeinen wird die Methode `set_`<*Attributname*> aufgerufen. Da diese vorher generiert worden ist, ist sie für alle Attribute definiert. Wenn ein Attribut von der generierten Methode `set_`<*Attributname*> *semantisch* falsch gesetzt wird, kann der Benutzer eine Methode `ins_`<*Attributname*> von Hand implementieren. Eine Methode `ins_`<*Attributname*> existiert nicht für jedes Attribut. Ist eine solche Methode definiert, so wird diese statt `set_`<*Attributname*> aufgerufen.

In der vorliegenden Datenbank ist dies in der Klasse `stck_pos` der Fall.

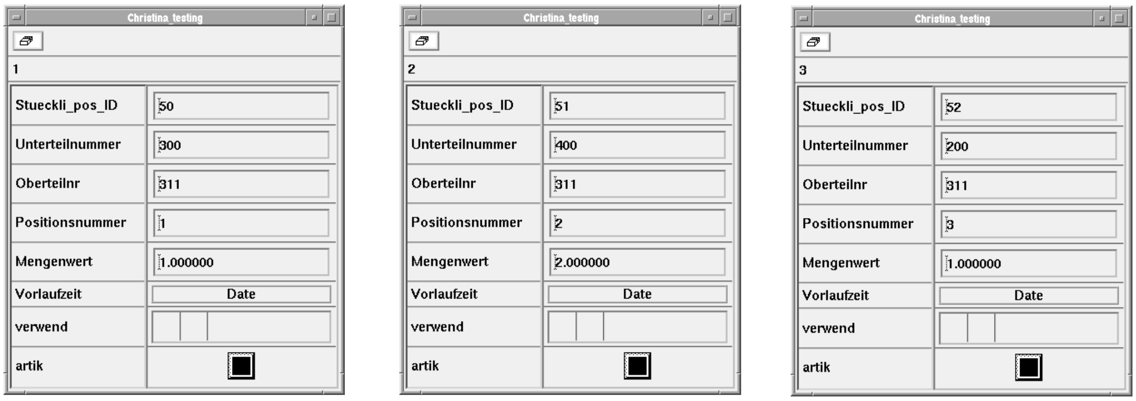

Abbildung 6.5: Objekte der Klasse `stck_pos` vor dem Eintragen der Relation zum Unterteil

Wie bereits in Abschnitt 5.2.4 auf Seite 30 beschrieben, stellt eine Stücklistenposition eine Beziehung auf dem Entity `artik` dar. Die Beziehung zum Oberteil wird über die Klasse `Stueckliste` abgebildet. Das Attribut `artik` der Klasse `stck_pos` repräsentiert die Beziehung zum Unterteil.

Für einen Artikel wie den Küchenschrank aus Abschnitt 5.2.4 sind in einer solchen Datenbank drei Objekte der Klasse `stck_pos` enthalten, da sich der Küchenschrank aus den drei Unterteilen Unterschrank, Tür und Arbeitsplatte zusammensetzt. Aus diesen drei Objekten setzt sich die Stückliste des Artikels zusammen. Abbildung 6.5 zeigt die drei Objekte der Klasse `stck_pos`, die noch keine Beziehung zum Unterteil haben.

Nach dem Eintragen der Relationen muß jedes dieser Objekte eine Referenz zum Unterteil besitzen. Bei dem Objekt mit der Unterteilnummer 400 muß das Attribut artik nach dem Eintragen der Relationen auf ein Objekt der Klasse `artik` verweisen, dessen Artikelnummer 400 ist. Bei dem Objekt handelt es sich um den Artikel Tür, der an Position 2 in die Stückliste des Küchenschranks eingeht. Analog muß bei dem ersten Objekt ein Verweis auf den Unterschrank und bei dem letzten Objekt ein Link auf die Arbeitsplatte vohanden sein.

Die Methode `set_artik`, die von der Methode `define_set_attr` für die Klasse `stck_pos` generiert wird, um diese Beziehung herzustellen, erwartet ein Objekt der Klasse `artik`, überprüft aber nicht, ob es sich bei diesem Objekt tatsächlich um ein Unterteil handelt.

```
method public set_artik(obj: artik) in class stck_pos;
method body set_artik in class stck_pos
{
self->artik = obj;
};
```

Das Tabellenobjekt `tmp_T_stck_pos` besitzt zwei Fremdschlüssel: Die Oberteilnr und die Unterteilnummer. Beide beziehen sich auf die Klasse `artik`. Anhand der Definition ist nicht klar, ob die Objekte der Klasse `artik` anhand der Oberteilnr oder der Unterteilnummer selektiert werden sollen.

Innerhalb der Methode `insert_artik` dieses Objekts wird folgende Anfrage an die Datenbank gestellt:

```
o2query(second_result,"select x from x in PPS_artik
where x.Artikelnummer=$1",eintrag.Oberteilnr);
```

Wird mit dem von `o2query` zurückgelieferten Objekt die Methode `set_artik` aufgerufen, so entsteht hier eine Beziehung zum Oberteil anstatt zum Unterteil. In dem obigen Beispiel bedeutet das, daß bei jedem Objekt eine Referenz zu einem Objekt mit der Artikelnummer 311 hergestellt wird. Das Objekt mit der Artikelnummer 311 ist aber der Küchenschrank selbst. D.h., daß in allen drei Stücklistenpositionen der Stückliste des Küchenschranks der Küchenschrank als Unterteil eingetragen würde.

Daher existiert in dieser Klasse die Methode `ins_artik`, die den Artikel nur dann bei dem Objekt der Klasse `stck_pos` einträgt, wenn es sich um ein Unterteil handelt, d.h. die Unterteilnummer mit der Artikelnummer übereinstimmt. Wird ein Objekt übergeben, bei dem die Bedingung nicht erfüllt ist, wird innerhalb der Methode `ins_artik` das richtige Objekt der Klasse `stck_pos` in der Datenbank gesucht und bei dem Attribut artik eingetragen.

Wird beispielsweise bei dem Objekt, das die zweite Stücklistenposition darstellt, also den Bezug zum Unterteil Tür herstellen soll, die Methode `ins_artik` mit dem Küchenschrank statt mit der Tür aufgerufen, so wird das Objekt der Klasse `artik`, das die Tür repräsentiert, in der Datenbank gesucht und als Unterteil eingetragen:

```
method body ins_artik(artik: artik): boolean in class stck_pos
{
   o2 set(artik) result;

   if(artik == nil)
      return false;

   /* Stimmt die Artikelnummer des Objekts artik mit der Unterteilnummer
      von self ueberein, so muss bei self->artik nur das Unterteil
      eingetragen werden */
   if(artik->Artikelnummer == self->Unterteilnummer)
   {
      self->artik = artik;
      return true;
   }
   else
   {
      /* Wurde nicht das Unterteil uebergeben, so muss das Objekt in der
         Datenbank gesucht werden, bevor es eingetragen werden kann. */
      o2query(result,"select x from x in PPS_artik where
            x.Artikelnummer=$1",self->Unterteilnummer);

      if(count(result)==1)
      {
         /* Die element-Funktion kann nur auf ein Objekt angewendet werden. */
         artik = element(result);
         /* Unterteil eintragen */
         self->artik = artik;
         return true;
      }
      else
         return false;
```

```
    }
    return true;
};
```

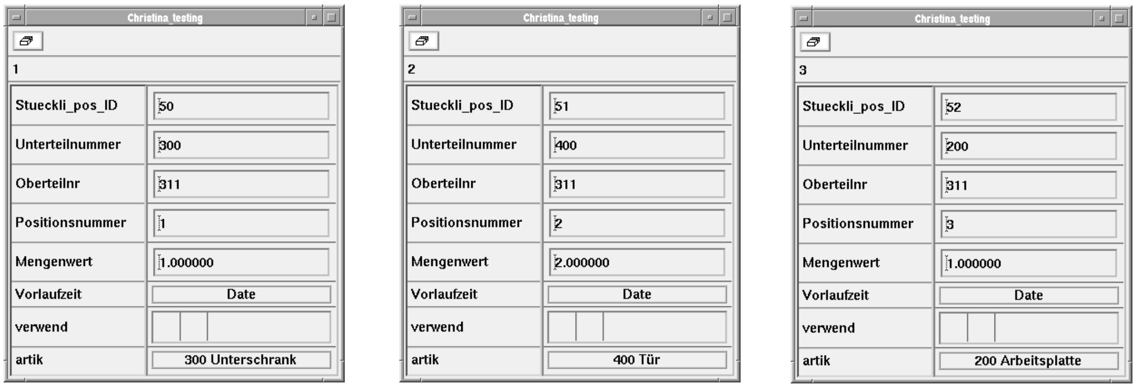

Abbildung 6.6: Objekte der Klasse `stck_pos` mit Bezug zum Unterteil

Wird in der Methode `insert_references` diese Methode an Stelle von `set_artik` aktiviert, so wird bei den Objekten aus Abbildung 6.5 das Unterteil eingetragen.

Wie Abbildung 6.6 zeigt, hat nun jedes Objekt der Klasse `stck_pos` den Bezug zu dem richtigen Objekt der Klasse `artik`. Das erste Objekt zeigt auf den Unterschrank, das zweite auf die Tür und das dritte Objekt verweist auf die Arbeitsplatte.

An diesem Punkt stellt sich die Frage: Wann ist es erforderlich eine Methode `ins_`<*Attributname*> von Hand zu schreiben? In diesem Beispiel enthält die relationale Tabelle `stck_pos` die Fremdschlüssel Oberteilnr und Unterteilnummer, die sich beide auf die Tabelle `artik` beziehen. Die objektorientierte Klasse `stck_pos` enthält nur ein Attribut, das sich auf ein Objekt der Klasse `artik` bezieht, so daß die Methode `define_insert_references` nicht entscheiden kann, aufgrund welches Schlüssels das zugehörige Objekt gesucht werden muß.

Das Problem tritt jedoch auch auf, wenn die Tabelle zwei Fremdschlüssel besitzt, die sich beide auf dieselbe Tabelle beziehen und die objektorientierte Klasse ebenfalls über zwei Attribute verfügt, die sich jeweils auf ein Objekt der entsprechenden Klasse beziehen. In der Klasse `stck_pos` könnte das zum Beispiel ein Attribut definiert sein, das die Beziehung zum Oberteil abbildet und sich ebenfalls auf ein Objekt der Klasse `artik` bezieht. Beim Generieren der Methoden `insert_references`, `insert_`<*Attributname*> sowie `set_`<*Attributname*> tritt kein Fehler auf. Auch nicht beim Eintragen der Relationen, weil die Datentypen, bzw. die Klassen übereinstimmen. Allerdings kann es beim Eintragen der Relationen passieren, das die Objekte vertauscht zugeordnet werden.

Es ist also immer dann erforderlich, eine Methode `ins_`<*Attributname*> zu schreiben, wenn die zugehörige Tabelle mehrere Fremdschlüssel enthält, die sich auf dieselbe Tabelle beziehen.

6.2.4 Aufruf der Methode `insert_references`

Nachdem alle erforderlichen Methoden generiert worden sind, kann das Eintragen der Relationen erfolgen, indem bei allen persistenten Tabellenobjekten, bei denen eine Methode mit dem Namen `insert_references` existiert, diese aufgerufen wird.

Bei Zuordnungstabellen existiert nur die Methode `insert_references`. Das Erzeugen der Relationen wird in diesem Fall direkt von dieser Methode gesteuert.

Abbildung 6.7 zeigt die bereits in Abbildung 6.3 dargestellten Objekte der Klassen `kunden` und `k_auftrag` nach dem Aufruf der Methode `insert_references`.

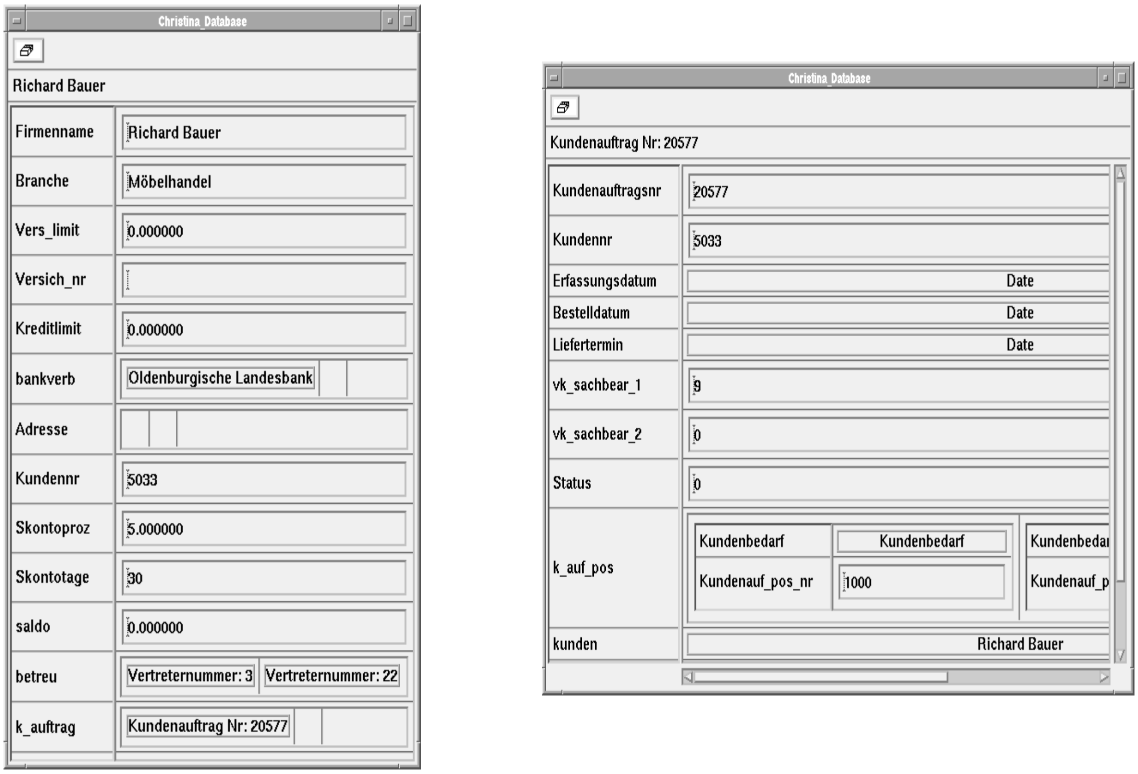

Abbildung 6.7: Objekte nach dem Eintragen der Relationen

6.2.5 Sonderfall: Stücklisten erzeugen

Zum Schluß müssen nur noch die Stücklisten der Artikel erzeugt werden. Im relationalen Modell muß die Stückliste jedesmal bei Bedarf aus der Tabelle `stck_pos` generiert werden. Im objektorientierten Modell wird die Stückliste zu einem Artikel nur einmal erzeugt. Nachdem alle Relationen erzeugt worden sind, wird für jedes Objekt der Klasse `artik` die zugehörige Stückliste generiert und unter dem Attribut stueckliste abgelegt. Über das Attribut stueckliste kann immer auf die Stückliste eines Artikels zugegriffen werden. Abbildung 6.8 zeigt einen Artikel, bei dem alle Relationen, sofern sie vorhanden sind, eingetragen sind, nur das Attribut stueckliste zeigt noch auf `nil`.

Abbildung 6.8: Artikel Küchenschrank ohne Stückliste

Jeder Artikel erzeugt „seine" Stückliste selbst. Die Methode `create_Stueckliste`, die in der Klasse `artik` implementiert ist, erzeugt ein neues Objekt der Klasse `Stueckliste` und ruft bei diesem Objekt die Methode `create` auf. Bezeichnet man das Objekt der Klasse `Stueckliste` mit `s`, kann die Methode folgendermaßen aufgerufen werden:

```
self->stueckliste = s->create(self);
```

Da die Methode `nil` zurückliefert, falls keine Stückliste zu dem Artikel existiert, wird das Attribut stueckliste des Artikels gleich auf den richtigen Wert gesetzt.

Eine Stückliste besteht aus einem Oberteil und vielen Stücklistenpositionen. Die `create`-Methode erwartet daher als Übergabeparameter ein Objekt der Klasse `artik`. Dieses Objekt wird als Oberteil in die Stückliste eingetragen. Anschließend wird die Liste der Stücklistenpositionen erzeugt. Die Stücklistenpositionen, die zu dem Artikel gehören, sind in der Menge `PPS_stck_pos` enthalten. Aus dieser Menge müssen diejenigen Stücklistenpositionen herausgefiltert werden, bei denen die Oberteilnummer mit der aktuellen Artikelnummer übereinstimmt. Die `create`-Methode enthält daher folgende `for`-Schleife:

```
for(pos in PPS_stck_pos where pos->Oberteilnr == artikel->Artikelnummer)
{
self->Positionen += list(pos);
}
```

Ist die Liste der Stücklistenpositionen nach dieser `for`-Schleife leer, wird `nil` zurückgegeben. In diesem Fall existiert zu dem gegebenen Artikel keine Stückliste. Andernfalls wird `self`, also die Stückliste selbst, zurückgeliefert.

Abbildung 6.9: Artikel Küchenschrank mit zugehöriger Stückliste

Für alle Objekte der Klasse `artik`, die in der Menge `PPS_artik` enthalten sind, wird durch Aufruf der Methode `create_Stueckliste` die zugehörige Stückliste erzeugt. Die Methode `sort` der Klasse `Stückliste` sortiert anschließend unter Verwendung des Bubblesort-Algorithmus die Elemente der Liste aufsteigend nach Positionsnummern. Abbildung 6.9 zeigt einen Artikel mit der zugehörigen Stückliste.

Kapitel 7

Die O_2-Applikationen

Im Rahmen der vorliegenden Arbeit sind drei verschiedene Applikationen entstanden:

1. Die Applikation `Tabellenmigration` erzeugt und löscht die temporären Tabellenklassen.
2. Die Applikation `Zuordnung_zu_Klassen` dient dem Erzeugen der Objekte und Eintragen der Relationen.
3. Die Applikation `Database` ermöglicht das Anzeigen der in der Datenbank vorhandenen Objekte.

7.1 Die Applikation `Tabellenmigration`

Die Applikation `Tabellenmigration` enthält Programme zum Erzeugen und Löschen von Tabellenklassen. Außerdem können persistente Objekte dieser Klassen geschaffen und mit den entsprechenden Daten aus den `unload`-Dateien der *Informix*-Datenbank gefüllt werden.

Das dynamische Anlegen und Löschen von Klassen mit Hilfe des `Meta_Schemas` funktioniert nicht in der Umgebung von O_2Tools. Daher kann die Applikation `Tabellenmigration` nur von der O_2Shell gestartet werden. Dazu ist auf der Shell folgendes Kommando abzusetzen:

```
run application Tabellenmigration
^D
```

Abbildung 7.1 zeigt das Fenster, das nach dem Start der Applikation am Bildschirm erscheint.

Zum Anlegen der Tabellenklassen ist die Datei erforderlich, die auch *Informix* zum Erzeugen der Tabellen verwendet. Die Datei wurde hier beispielhaft mit `tables.sql` bezeichnet. Der Name

Abbildung 7.1: Die Applikation Tabellenmigration

der Datei muß eingelesen werden. Die Dateien, in denen sich die Daten befinden, müssen sich in einem Unterverzeichnis `unload` des Verzeichnisses befinden, in dem sich die Datei `tables.sql` befindet, die zum Erzeugen der Tabellenklassen dient. Daher wird zuerst nur der Verzeichnisname eingelesen. (Siehe Abbildung 7.2)

Abbildung 7.2: Einlesen des Verzeichnisnamens

Beim Einlesen des Verzeichnisses muß der vollständige Pfad angegeben werden. Relative Pfade sind nicht möglich. Zum Erzeugen der Tabellenklassen wird anschließend der Dateiname eingelesen. (Siehe Abbildung 7.3).

Abbildung 7.3: Einlesen des Dateinamens

Wird die Datei oder das Verzeichnis nicht gefunden, muß beides erneut eingegeben werden.

Nachdem die Tabellenobjekte erzeugt und initialisiert worden sind, erscheint wiederum das Fenster aus Abbildung 7.1. Die Tabellenobjekte können jetzt mit Daten gefüllt werden. Dazu ist der Punkt Daten laden zu wählen. Das Verzeichnis, aus dem die Daten geladen werden sollen, muß nicht noch einmal eingegeben werden.

7.2 Die Applikation `Zuordnung_zu_Klassen`

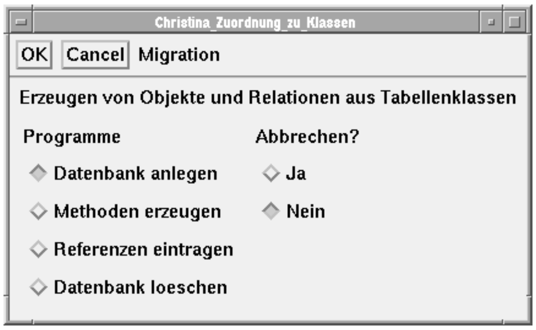

Abbildung 7.4: Die Applikation Zuordnung_zu_Klassen

Die Applikation `Zuordnung_zu_Klassen` dient dazu, aus den Tupeln der Tabellenobjekte Objekte der objektorientierten Datenbank zu erzeugen und die entstandenen Objekte zueinander in Relation zu setzen. Die Applikation aktiviert dazu die in der Klasse `TabellenObj` vorgesehenen Methoden. Außerdem werden in dieser Applikation die Stücklisten für die Artikel erzeugt.

Die Applikation kann entweder von der Shell aus mit Kommando:

```
run application Zuordnung_zu_Klassen
^D
```

oder in der Umgebung von O_2Tools im Applications-Fenster unter dem Menü-Punkt Applications/Test aufgerufen werden. Abbildung 7.4 zeigt das Fenster, das nach dem Aufruf erscheint.

Wird der Punkt Datenbank anlegen gewählt, werden zunächst Namen erzeugt, um die Objekte später persistent machen zu können. Anschließend wird die Methode `attach` bei allen Subklassen der Klasse `TabellenObj` aufgerufen, um aus den Tupeln der Tabellenobjekte einzelne Objekte zu erzeugen und persistent abzulegen. Schließlich werden bei den Klassen, die Subklassen haben, Objekte dieser Subklassen erzeugt.

Zum Generieren der Methoden `set_`<*Attributname*>, `insert_references` und `insert_`<*Attributname*> ist Methoden erzeugen vorgesehen.

Wird der Punkt Referenzen eintragen aktiviert, so wird die Methode `insert_references` bei allen Tabellenobjekten, bei denen eine solche Methode definiert ist, aufgerufen. Anschließend wird das Programm `create_Stuecklisten` gestartet. Wurde die Datenbank neu angelegt, so muß dieses Programm vorher neu übersetzt werden, da es auf den Namen `PPS_artik` zugreift. Dies kann mit dem Befehl `compile depend` auf der Shell geschehen. Allerdings ist die Applikation dazu einmal abzubrechen. Wird das Kommando `compile depend` nicht ausgeführt, werden zwar die Referenzen bei den Objekten eingetragen, aber die Stücklisten bei den Artikeln werden nicht erzeugt.

Der Menü-Punkt Datenbank loeschen löscht alle Namen, die zu der Datenbank gehören. Die persistenten Tabellenobjekte werden hier nicht gelöscht.

7.3 Die Applikation `Database`

Abbildung 7.5: Persistente Mengen der objektorientierten Datenbank

Zum Visualisieren aller in der Datenbank vorhandenen Objekte ist die Applikation `Database` vorgesehen. Diese kann in der Umgebung von O_2Tools oder durch Absetzen des Kommandos:

```
run application Database
^D
```

auf der Shell aktiviert werden.

Nach dem Start der Applikation kann der Benutzer wählen, ob er sich die Tabellenobjekte oder die in der Datenbank vorhandenen Objekte anzeigen lassen möchte. In jedem Fall erscheint anschließend ein Fenster, das entweder die Namen aller persistenten Tabellenobjekte oder die Namen aller persistenten Mengen der in der Datenbank vorhandenen Objekte enthält. Abbildung 7.5 zeigt das Fenster, das erscheint, wenn die persistenten Objekte der Datenbank angezeigt werden sollen.

Wird hier der Name `PPS_artik` ausgewählt, so wird die Menge aller persistenten Objekte der Klasse `artik` angezeigt (siehe Abbildung 7.6).

Abbildung 7.6: Menge persistenter Objekte der Klasse `artik`

In O_2 sind Objekte durch `composition links` miteinander verbunden. D.h., beginnend bei einem Objekt kann der Teil der Datenbank „erforscht“ werden, der von diesem Objekt aus erreichbar ist, indem den Links nachgegangen wird.

Um ein Objekt aus der Menge der persistenten Artikel auszuwählen und anzuzeigen, klickt man mit der rechten Maustaste auf das Objekt und wählt in dem erscheinenden Menü den Punkt display. Auf diese Weise kann jedem `composition link` nachgegangen werden.

In der Klasse `artik` wurde die Methode `title` folgendermaßen redefiniert:

```
method body title: string in class artik
{
char chr[10];
o2 string help;

sprintf(chr,"%d",self->Artikelnummer);
strcpy(help,chr);
help += " " + self->Artikelbez;
return help;
};
```

Die Methode sorgt dafür, daß der Titel eines Objekts durch den String ersetzt wird, der von dieser Methode zurückgeliefert wird. Wird die Methode nicht redefiniert, so wird der Klassenname als Titel jedes Objekts angezeigt. Abbildung 7.6 zeigt, daß in diesem Fall die Artikelnummer und die Artikelbezeichnung des jeweiligen Objekts erscheint.

7.3.1 Migration ohne Benutzerinteraktion

Die Migration kann auch ohne Interaktion des Benutzers durchgeführt werden. Der Benutzer wird nur am Anfang aufgefordert, den Namen des Verzeichnisses, in dem sich die `unload`-Dateien befinden, sowie den Namen der Datei einzugeben, die auch *Informix* zum anlegen der Daten benutzt. Dazu wird eine Datei abgearbeitet, die `o2migration` heißt. In dieser Datei werden nacheinander alle Programme abgearbeitet, die zur Migration der Datenbank erforderlich sind. Die Datei muß von der O_2 Shell abgearbeitet werden. Dies geschieht, indem auf der O_2 Shell folgendes Kommando ausgeführt wird:

```
#"/<pathname>/o2migration"
^D
```

Die halbautomatische Migration der Datenstruktur muß bei der Ausführung dieses Skripts bereits abgeschlossen sein. Als Pfad muß auch hier der vollständige Pfad zu der Datei angegeben werden, relative Pfadangaben sind nicht möglich.

Kapitel 8

Diskussion

In diesem Kapitel werden die Vor- und Nachteile des angewendeten Migrationskonzeptes diskutiert und einige weiterführende Überlegungen angestellt.

Das hier vorgestellte Migrationskonzept ermöglicht ein vollautomatisches Übertragen der Daten einer relationalen Datenbank in eine objektorientierte Datenbank. Die Migration der Datenstruktur läßt sich jedoch nicht vollautomatisch durchführen. In Kapitel 5 wurde der kanonische Entwurf der Klassenhierarchie auf der Basis des ER-Diagramms beschrieben. Die Klassenhierarchie muß hier von Hand implementiert werden. Bei der Generierung der Tabellenklassen liegt das relationale Datenmodell des PPS-Kerns zugrunde. D.h., beim automatischen Erzeugen dieser Klassen wird auf die Datei zugegriffen, die hier mit `tables.sql` bezeichnet wurde, und nicht auf das in Abbildung 3.1 gezeigte ER-Diagramm. Aus den Tabellenklassen die Klassenhierarchie der objektorientierten Datenbank zu generieren, ist deshalb nicht möglich. Generell kann man sagen, daß zu Tabellen, die nur Zuordnungen zwischen anderen Tabellen vornehmen, keine Klassen erzeugt werden müssen, zu allen anderen aber schon. Die einzige Ausnahme bildet in diesem Beispiel die Klasse `stck_pos`. Aufgrund der Definition der Tabelle ist jedoch nicht erkennbar, ob es sich um eine Zuordnungstabelle handelt oder nicht. Das liegt daran, daß sich mit dem relationalen Modell fast keine semantische Information der Anwendungswelt darstellen läßt [HL82].

Wie die in Kapitel 5 aufgestellten Regeln zeigen, ließe sich das Erzeugen der Klassen auf der Basis des ER-Diagramms automatisieren. Dazu wäre allerdings das Aufstellen weiterer Regeln beispielsweise für tertiäre Beziehungen etc. erforderlich. Ein Möglichkeit wäre, bei der Erstellung des ER-Diagramms ein Tool einzusetzen, das über einen Databasedesigner verfügt, so daß die relationale Datenbank direkt aus dem ER-Diagramm generiert werden kann. Ein solcher Databasedesigner könnte z.B. bei $m : n$-Beziehungen Zuordnungstabellen erzeugen, deren Namen sich aus den beiden zueinander in Relation stehenden Tabellen zusammensetzt.

In jedem Fall wird das relationale Datenmodell nahezu 1:1 auf das objektorientierte Datenmodell abgebildet, damit das vollautomatische Übertragen der Daten in die ooDB ermöglicht werden kann. Ergibt sich bei einem Redesign in der Praxis eine völlig andere Datenstruktur für die objektorientierte Datenbank, würde dies das Übertragen der Daten erschweren. Das Konzept der Tabellenobjekte kann aber trotzdem genutzt werden. Beim Erzeugen der Objekte muß in einem solchen Fall evtl. anders vorgegangen werden.

Werden die in Kapitel 5 beschriebenen Regeln zur Erzeugung der Klassenhierarchie berücksichtigt, so kann die Migration der Daten automatisch erfolgen.

Die Migration der relationalen Datenbank in die objektorientierte Datenbank dauert, für die gesamte Datenbank mit allen 19556 Einträgen, auf einer SUN Sparc 20 etwa 8 Stunden.

Das Übertragen der Daten in die temporären Tabellenobjekte und auch das Erzeugen der einzelnen Objekte der endgültigen Datenbank ist dabei am wenigsten zeitaufwendig. Das Einlesen der Daten aus den Dateien in die Tabellenobjekte dauert knapp 6 Minuten. Es vergehen weitere 5 Minuten beim Erzeugen der Objekte aus den Tabellenobjekten. 95% der Zeit vergeht beim Eintragen der Relationen in den Objekten, da hier die Objekte in der Datenbank gesucht werden müssen. Die Suche ist deshalb besonders langsam, weil nicht die interne Objekt_ID zum Suchen verwendet werden kann, sondern die Objekte aufgrund ihrer relationalen Schlüssel identifiziert werden müssen. Das Generieren der Methoden sowohl für die temporären als auch für die anderen Klassen nimmt 10 Minuten in Anspruch. Die restliche Zeit vergeht beim Eintragen der Referenzen.

Das Verfahren läßt sich beschleunigen, indem auf die Attribute, die die „alten“ Primärschlüssel bzw. Fremdschlüssel darstellen, ein Index gelegt wird. Wenn in *OQL* ein oder mehrere Elemente mit einer bestimmten Eigenschaft aus einer Menge extrahiert werden, muß jedesmal die gesamte Menge durchsucht werden, um die gewünschten Elemente zu finden. Wird ein Index eingeführt, so ist es dem System möglich, auf die Elemente, auf die das Suchmuster paßt, direkt zuzugreifen. Dadurch kann die Suche erheblich verkürzt werden. Ein Index kann auf eine oder mehrere Attribute einer Menge gelegt werden. Vorraussetzung ist, daß die Menge einen *konstanten Namen* besitzt.

Durch Einführen eines Index auf die erhalten gebliebenen Primär- und Fremdschlüssel, kann das Verfahren erheblich beschleunigt werden. Die Migration der gesamten Datenbank verkürzt sich dadurch auf etwas mehr als 2 Stunden.

Kapitel 9

Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein Verfahren zu entwickeln, das es ermöglicht, Daten, die sich in einer relationalen Datenbank befinden, in eine objektorientierte Datenbank zu migrieren. Dazu wurde eine relationale Datenbank ausgewählt, die einem PPS-System zugrunde liegt.

Die PPS-System-Datenbank, die in der Industrie zum Einsatz kommt, ist sehr umfangreich. Daher wurde im Rahmen dieser Arbeit die Essenz des PPS-Systems extrahiert und ein „PPS-Kern" modelliert. Hierzu ist ein ER-Diagramm entstanden, das unter anderem zum kanonischen Entwurf der objektorientierten Klassen genutzt wurde. Zu diesem PPS-Kern wurde eine *Informix*-Datenbank erzeugt und mit Daten aus der Datenbank des PPS-Systems gefüllt. Die entstandene PPS-Kern-Datenbank wurde in die objektorientierte Datenbank O_2 migriert.

Die Migration selbst ist unterteilt in die Migration der Datenstruktur einerseits und die Migration der Daten andererseits. Um auf den Daten leichter operieren zu können, werden die tabellarisch gespeicherten Daten zuerst nach O_2 portiert und in temporären Objekten gespeichert. Diese temporären Objekte sind Instanzen von Klassen, die dynamisch angelegt werden und ebenfalls nur temporär in der Datenbank vorhanden sind. Die temporären Klassen bilden die Tabellenstruktur der relationalen Datenbank nach. Für jede relationale Tabelle wird eine solche Klasse zur Laufzeit erzeugt. Die Struktur der Klassen, die erzeugt werden, wird durch die Datei festgelegt, die auch *Informix* zum Erzeugen der relationalen Datenbank benutzt. Tabellenklassen können daher auch für andere als die PPS-Kern-Datenbank in O_2 angelegt werden. Von jeder dieser Klassen wird *ein* Objekt erzeugt und mit den Daten der entsprechenden relationalen Tabelle gefüllt.

Nachdem sich die Daten bereits in der objektorientierten Datenbank befinden, ist die Migration der Datenstruktur erforderlich. Da auf der Basis der relationalen Datenbank eine Generierung der Klassenhierarchie nicht möglich ist, geschieht der Entwurf der Klassen kanonisch anhand des ER-Diagramms. Für die im Rahmen des PPS-Kerns auftretetenden Beziehungen werden Regeln aufgestellt, um diese im objektorientierten Datenmodell abzubilden. Dabei entsteht für jedes Entity im ER-Diagramm eine Klasse. Die Beziehungen, die zwischen den Entities bestehen, werden durch zusätzliche Attribute in den Klassen abgebildet. Bei der Erzeugung der Klassen bleiben alle Attribute erhalten, d.h. relationale Schlüsselattribute sind auch in den Klassen als Attribute vorhanden.

Nach Abschluß der Migration der Datenstruktur werden die Daten aus den temporären Tabellenobjekten in Objekte der objektorientierten Datenbank überführt. Für jedes Tupel einer Tabelle, die nicht nur Zuordnungen zwischen anderen Tabellen abbildet, wird eine Instanz einer Klasse erzeugt. Nachdem alle Objekte erzeugt worden sind, werden die Beziehungen, die zwischen diesen Objekten bestehen, hergestellt. Dazu werden die Tabellenobjekte, die Fremdschlüssel enthalten, noch einmal bearbeitet. Aufgrund der Schlüsselattribute, die bei der Erzeugung der Klassen erhalten geblieben sind, können die Objekte nun in der Datenbank identifiziert und zueinander in Beziehung gesetzt werden.

Die Migration kann entweder interaktiv durch Aufruf der Applikationen `Tabellenmigration` und `Zuordnung_zu_Klassen` oder ohne Benutzerinteraktion mit Hilfe des Skripts `o2migration` durchgeführt werden. Wird die Migration nicht interaktiv durchgeführt, ist die Klassenhierarchie vorher anzulegen. Das Erzeugen der Tabellenobjekte funktioniert auch, wenn die Klassenhierarchie noch nicht existiert.

Mit dem hier vorgestellten Verfahren kann eine *Informix*-Datenbank in das objektorientierte Datenbankmanagementsystem O_2 migriert werden. Eine mögliche Erweiterung des Verfahrens besteht darin, nicht nur *Informix*-Datenbanken, sondern auch andere relationale Datenbanken zu migrieren. Beim Erzeugen der temporären Tabellenklassen und beim Laden der Daten aus den Dateien ist dabei das Format der `unload`-Dateien entscheidend. Die Tabellenobjekte werden in der Applikation `Tabellenmigration` angelegt, die im Rahmen der Arbeit entstanden ist. Diese Applikation kann so erweitert werden, daß auch `unload`-Dateien anderer relationaler Datenbankmanagementsysteme lesbar sind und somit auch andere relationale Datenbanken migriert werden können. Beim Erzeugen der Objekte aus den Tabellenobjekten spielt es keine Rolle, um welche relationale Datenbank es sich dabei handelt. Da schon die Tabellenobjekte zueinander in Beziehung gesetzt werden, muß die relationale Datenbank allerdings bereits über Konzepte für Primär- und Fremdschlüssel verfügen.

Die entwickelten Applikationen, Programme und Methoden sind in O_2C geschrieben und deshalb an das objektorientierte Datenbankmanagementsystem O_2 gebunden. Die entwickelten Konzepte können aber auch allgemein zur Migration von relationalen Datenbanken in objektorientierte Datenbankmanagementsysteme genutzt werden.

Anhang A

Installation des Datenbankschemas

Nachdem O_2 Shell gestartet worden ist, erscheint folgende Ausgabe auf dem Bildschirm:

`type your command and end with ^D`

Da alle Klassendefinitionen in einem Schema abgelegt werden, muß zunächst ein solches Schema erzeugt werden:

`create schema` $<$*Schema_name*$>$
`^D`

Da die Objekte in einer „Base“ abgelegt werden, ist auch das Erzeugen einer solchen Base erforderlich. Dazu ist folgendes Kommando auf der Shell abzusetzen:

`create base` $<$*base_name*$>$
`^D`

Damit in einer solchen Base Objekte angelegt bzw. verändert werden dürfen, muß der Status der Base auf `TEST` gesetzt werden. Ist der Status einer Base `CONTROLLED`, können in dieser Base keine Objekte angelegt werden.

`modify status TEST in base` $<$*base_name*$>$
`^D`

Mit dem Kommando `#"/<`*pathname*`>/<`*filename*`>"` können O_2-Befehle, die in einer Datei stehen, auf der Shell ausgeführt werden. Beim Speichern eines Schemas in einem Verzeichnis legt O_2 eine Datei mit dem Namen $<$*Schema_name*$>$`.load` an. Durch den Aufruf: `#"/<`*pathname*`>/<`*Schema_name*`>.load"` kann ein Datenbankschema wiederhergestellt werden. Dabei können keine relativen Pfadnamen verwendet werden, sondern es muß stets der vollständige Pfad zu der Datei eingegeben werden.

Alle genannten Befehle sind in der Datei `install_pps` zusammengefaßt. Wird nach dem Start von O_2 auf der Shell das Kommando: `#"/`<*pathname*>`/install_pps"` abgesetzt, so wird das Datenbankschema zur Migration des PPS-Kerns wiederhergestellt.

Anhang B

Die Methoden von `TabellenObj`

B.1 Die Methode `attach`

Falls es sich nicht um eine Zuordnungstabelle handelt, erzeugt die Methode für jedes Tupel des Tabellenobjekts ein Objekt einer Klasse und weist dem Objekt die Werte des Tuples zu (siehe Abschnitt 6.1.2, Seite 44). Die entstandenen Objekte werden anschließend persistent in der Datenbank abgelegt.

```
method public attach in class TabellenObj;
method body attach in class TabellenObj
{
  o2 string comp, name, promptstring, cmd, eintragliste;
  o2 list(string) attname;
  o2 integer i;

  /* Meta-Klassen-Variablen */
  o2 Meta_type typ;
  o2 Meta_tuple tup;
  o2 Meta_class m, cl;
  o2 Meta_definition def,tupdef;

  /* Attribute der Klasse */
  o2 list ( tuple(name: string, visibility: string,
      type: Meta_definition)) classattr;
  o2 tuple (name:string, type:Meta_definition) att;
  o2 tuple(name: string, visibility: string, type: Meta_definition) attrib;

  /* m ist das Meta-Klassenobjekt des jeweiligen Tabellenobjektes. */
  m = Schema->class(self);
```

```
  attname = list();

  eintragliste = "";  /* In eintragliste wird die Definition des Tupels
                   abgelegt, das einem Eintrag einer Tabelle entspricht */
  for ( att in self->get_attributes)
  {
    attname += list(att.name);
    if( att.type->is_class)
    {
      cl = (o2 Meta_class) att.type;
      if(cl->name == "Date")
      {
        eintragliste += att.name + ":" + cl->name + ",";
      }
    }
    else
    {
      typ = (o2 Meta_type)att.type;
      eintragliste += att.name + ":" + typ->kind + ",";
    }
  }
  /* Das letzte Komma mu"s aus eintragliste entfernt werden */
  eintragliste[(count(eintragliste)-1):(count(eintragliste)-1)]="";

  def = Schema->definition(self->classname);

/* Es handelt sich um keine Zuordnungstabelle */
if( def != nil)
{
  cmd = "run body{\n"  /* cmd-String fuer Programm erzeugen */
      + "o2 " + self->classname + " obj;\n"  /* Objekt das erzeugt werden soll */
      + "o2 tuple(" + eintragliste + ") eintrag;\n"  /* ein Tupel der Tabelle */
      + "for(eintrag in tmp_" + m->name + "->tabelle)\n{" /* Tabelle durchlaufen */
        + " obj = new " + self->classname + ";\n";    /* neues Objekt erzeugen */


  /* Attribute der Klasse abfragen */
  classattr = ((o2 Meta_class)def)->attributes;

  /* Attribute, deren Datentyp eine Klasse ist,
     bilden Relationen zwischen Objekten ab.
     Einzige Ausnahme ist die Klasse "Date" */
  for(attrib in classattr where (( !(attrib.type->is_class))
   || (((o2 Meta_class)attrib.type)->name == "Date")))
  {
```

```
    /* Attribute, die eine m:n-Beziehung abbilden, sind
       keine Klassen, sollen aber auch nicht bearbeitet
       werden. */
    if (!attrib.type->is_class)
    {
      if(((o2 Meta_type) attrib.type)->kind in
           list("set","tuple", "list", "unique set") < 0)
      {
         for( name in attname where name == attrib.name)
          cmd += "\tobj->" + attrib.name + "=eintrag." + name + ";\n";
      }
    }
    else
    {
      for( name in attname where name == attrib.name)
        cmd += "\tobj->" + attrib.name + "=eintrag." + name + ";\n";
    }
  }
  /* neues Objekt der Menge der persistenten Objekte hinzufuegen */
  cmd += "PPS_" + self->classname + "+= set(obj);}\n}";

  /* Kontrollausgabe */
  printf("%s",cmd);

  Schema->command( cmd );
  }
};
```

B.2 Die Methode `define_insert_references`

Die Methode generiert für alle Tabellenobjekte, deren Liste der Fremdschlüssel nicht leer ist, eine Methode `insert_references` (siehe Abschnitt 6.2.1 auf Seite 48). Handelt es sich um eine Zuordnungstabelle, wird `insert_references` von der Methode `define_insert_references` generiert. Andernfalls wird die Methode `def_insert_for_class` aufgerufen (siehe Anhang B.3).

```
method body define_insert_references in class TabellenObj
{
   char chr[2];
   o2 string tablename, body, classname, signature, methodcalls, variables,
         remember;
   o2 string cond, cond2, help, key_type,queries, element1, element2,
        loop, name,key;
   o2 integer i;
   o2 list(string) attrnames,param;
```

```
o2 tuple(name: string,type: Meta_definition) attr;
o2 tuple(name: string,table: string,
                       reference: string,
                       obj_ref: TabellenObj)foreign, alien;
o2 TabellenObj table;
o2 boolean ready;

/* Der Name dieser Klasse wird zum Erzeugen einer Methode benoetigt.*/
tablename = Schema->class(self)->name;

/* Hat das Objekt keine Fremdschluessel, wird keine Methode generiert */
if(self->foreign_keys == list())
   return 1;

ready= false;
if(self->is_class)
{
/* Es handelt sich nicht um eine Zuordnungstabelle */
   self->def_insert_for_class;
}
else
{
   signature = "create method public insert_references in class "
          + tablename +";\n";
   body = "method body insert_references in class "
      + tablename +"\n{";

   i=1;
   sprintf(chr,"%d",i);
   strcpy(help,chr);
   for(name in self->return_param)
   {
      remember += ",eintrag." + name;
   }

   /* Zur Variablendeklaration gehoert ein Tupel eintrag, das
      dieselbe Struktur hat wie jedes Tupel von self->tabelle */
   variables += "o2 tuple(";
   for (attr in self->get_attributes)
   {
      /* Name und Datentyp jedes Attributs des Tupels muessen
         mit dem von self->Tabelle uebereinstimmen */
      variables += attr.name + ":" + attr.type->name + ",";
   }
   variables[count(variables)-1:] = "";
```

```
variables += ")eintrag;\n";

loop = "for (eintrag in self->tabelle)\n{\n";
cond = "if (";
cond2 = "";
element2 ="";

for(foreign in self->foreign_keys)
{
   sprintf(chr,"%d",i);
   strcpy(help,chr);
   classname = is_attribute_in(foreign.obj_ref->classname, self->classname);
   if ((classname != foreign.obj_ref->classname) && ( classname != ""))
   {
      variables += "o2 " + foreign.obj_ref->classname + " help;\n"
               + "o2 " + classname + " obj" + help + "= new void "
               + classname + ";\n";
      element2 += "help = element(result" + help +");\n";
      cond2 += "if(help->class_of == obj" + help + "->class_of)\n{";
      cond2 += "obj" + help + " = (o2 " + classname +") help;\n";
      ready = true;
   }
   else
   {
      element1 += "obj" + help + " = element(result" + help +");\n";
   }
   if(! foreign.obj_ref->is_class)
   {

      for(alien in foreign.obj_ref->foreign_keys)
      {
         classname = is_attribute_in(alien.obj_ref->classname,self->classname);
         if (classname != "")
         {
            table = alien.obj_ref;
            key = alien.name;
            remember = "";
            break;
         }
      }
      variables += "o2 tuple(";
      for(attr in foreign.obj_ref->get_attributes)
      {
         variables += attr.name + ":" + attr.type->name + ",";
         if ( strcmp(foreign.name,attr.name)==0)
```

```
      {
         if( strcmp(attr.type->name,"integer")==0)
            key_type = "d";
         if(strcmp(attr.type->name,"real")==0)
            key_type = "lf";
         if(strcmp(attr.type->name,"string")==0)
            key_type = "s";
      }
   }
   variables[count(variables)-1:] = "";
   variables += ")tup;\n";
   variables += "char help[30];\no2 string param;\n";
   loop += "sprintf(help,\"%" + key_type + "\",eintrag." + foreign.name;
   loop+= ");\nstrcpy(param,help);\ntup = tmp_T_" + foreign.table;
   loop += "->get_tuple(\"" + foreign.name + "\",param);\n";
   variables += "o2 " + classname + " obj" + help +";\n";
   variables += "o2 set(" + classname + ") result" + help +";\n";
   queries += "o2query(result" + help + ",\"select x from x in PPS_"
      + classname + " where x." + key + "=$1\",tup." + key + ");\n";

}
else
{
   if( ready == false)
      variables += "o2 " + foreign.obj_ref->classname + " obj" + help +";\n";
   variables += "o2 set(" + foreign.obj_ref->classname
         + ") result" + help +";\n";
   queries += "o2query(result" + help + ",\"select x from x in PPS_"
      + foreign.obj_ref->classname + " where x." + foreign.reference
      + "=$1\",eintrag." + foreign.name + ");\n";
}
ready = false;
cond += "(count(result" + help + ")==1)";
methodcalls += "obj" + help + "->set_" +self->classname;

i++;
sprintf(chr,"%d",i);
strcpy(help,chr);
if(i <= count(self->foreign_keys))
{
   cond += "&&";
   methodcalls += "(obj" + help + remember + ");\n";
}
else
{
```

```
            methodcalls += "(obj1" + remember + ");\n";
         }
      }

      cond +=")\n{\n";

      if(cond2 != "")
         methodcalls += "}\n";

      /* Rumpf zusammensetzen */
      body += variables + loop + queries + cond + element2 + cond2 + element1
            + "\ntransaction;\n" + methodcalls + "validate;\n}\n}\n}\n";
      /* Kontrollausgabe */
      printf("%s\n",body);

      /* entstandene Methode uebersetzen */
      Schema->command( signature );
      Schema->command( body );

   }
   return 0;
};
```

B.3 Die Methode `def_insert_for_class`

Tabellenobjekte, die nicht nur Zuordnungen vornehmen, müssen zum Eintragen der Relationen evtl. mehr als einmal durchlaufen werden. Die Methode `def_insert_for_class` generiert für jeden Fremdschlüssel eine Methode, die zum Erzeugen der Relation aufgerufen werden muß. Außerdem wird eine Methode `insert_references` generiert, die alle vorher definierten Methoden aktiviert, so daß zum Eintragen der Relationen nur die Methode `insert_references` aufgerufen werden muß (siehe Abschnitt 6.2.1).

```
method body def_insert_for_class in class TabellenObj
{   char chr[2];
   o2 string cond, param, tablename, attname,helpstr, body, key, actual_name;
   o2 string signature, str, methodcalls,met_name, variables, help,
        queries, elements, loop;
   o2 integer i,j;
   o2 list(string) methodnames;
   o2 tuple(name: string,type: Meta_definition) attr;
   o2 tuple(name: string,table: string,
                          reference: string,
                          obj_ref: TabellenObj)foreign;
```

```
/* Der Name der aktuellen Klasse wird zum Generieren einer Methode in
   dieser Klasse benoetigt. */
tablename = Schema->class(self)->name;

/* Es handelt sich nicht um eine Zuordnungstabelle */

i=0;      /* Initialisierungen */
param="\"";

/* Besteht der Primaerschluessel aus mehr als einem Attribut,
   sind bei einer query alle anzugeben.*/
for(key in self->primary_key)
{
   i++;
   sprintf(chr,"%d",i);
   strcpy(help,chr);

   /* fuer alle Primaerschluessel muessen bei einer query
      Werte als Parameter uebergeben werden */
   param += ",eintrag." + key;

   /* in der where-Klausel muessen alle Primaerschluessel
      angegeben werden */
   cond += "x." + key + "=$" + help;
   if(i < count(self->primary_key))
      cond += " && ";
}
/* Es werden evtl. mehrere Methoden generiert. Diese
   werden von insert_references spaeter aufgerufen.
   In methodnames werden die Methodennamen abgelegt. */
methodnames = list();

/* Fuer jeden Fremdschluessel wird eine Methode generiert */
for(foreign in self->foreign_keys)
{
   met_name = "";   /* alter Methodenname wird geloescht */

   /* Klassenname, der Klasse zu der eine Beziehung herzustellen ist.*/
   actual_name = foreign.obj_ref->classname;

   if(foreign.obj_ref->classname in methodnames < 0)
   {
      /* falls noch keine Methode dieses Namens existiert
         wird der Name der Klasse verwendet */
      methodnames += list(foreign.obj_ref->classname);
```

```
}
else
{
   /* existiert eine Methode mit dem Namen insert_<<classname>>,
      dann gibt es ein zweites Attribut, das auf ein Objekt dieser
      Klasse verweist.*/
   attname = exists_second_attribute(self->classname,
                                     foreign.obj_ref->classname);
   if( attname != "")
   {
      /* Als Methodenname wird der Name dieses Attributs verwendet.*/
      methodnames += list(attname);
      actual_name = attname;
   }
}
/* Signatur der Methode erzeugen */
signature = "create method public insert_" + actual_name
     + " in class " + tablename +";\n";

/* Rumpf zusammensetzen */
body = "method body insert_" + actual_name + " in class "
     + tablename +"\n{\n";

/* Variablendeklaration erzeugen */
body += "o2 " + self->classname + " obj1;\n"
+ "o2 set(" + self->classname + ") first_result;\n"
+ "o2 " + foreign.obj_ref->classname + " obj2;\n"
+ "o2 set(" + foreign.obj_ref->classname + ") second_result;\n"
+ "o2 tuple(";

/* Zum Durchlaufen der Tabelle wird eine Variable eintrag vom Typ tuple()
   generiert. Die Attribute muessen mit denen von self->tabelle
   uebereinstimmen.*/
/* self->get_attributes liefert eine Liste der Attribute von self->tabelle */
for (attr in self->get_attributes)
{
   /* Namen und Datentyp fuer jedes Attribut
      des Tupels eintrag zusammensetzen */
   body += attr.name + ":" + attr.type->name + ",";
}
/* Das letzte Komma entfernen */
body[count(body)-1:] = "";
/* Variablendeklaration ist abgeschlossen */
body += ")eintrag;\n\n";
```

```
/* Durchlaufen der Tabelle */
body += "for(eintrag in self->tabelle)\n{\n";

/* Objekte der Klassen self->classname und foreign.obj_ref->classname
   anhand der Schluessel selektieren */
body += "o2query(first_result,\"select x from x in PPS_" + self->classname
   + " where " + cond + param + ");\n"
   + "o2query(second_result,\"select x from x in PPS_"
   + foreign.obj_ref->classname + " where x." + foreign.reference
   + "=$1\",eintrag." +  foreign.name + ");\n"
   + "if((count(first_result)==1)&&(count(second_result)==1))\n{\n"
   + "obj1 = element(first_result);\n"
   + "obj2 = element(second_result);\n\n"
   + "transaction;\n";

   /* Methodenaufruf zum Eintragen der Relation erzeugen */
   if( is_attribute_in(self->classname, foreign.obj_ref->classname) != "")
   {
      /* Die Methode die bei dem jeweiligen Objekt aufgerufen
         wird heisst entweder set_<<Klassenname>> oder
         ins_<<Klassenname>>. Falls ins_<<Klassenname>> existiert,
         ist diese Methode aufzurufen.*/
      if( actual_name != "")
      {
         str = "ins_" + actual_name;
         if( exists_diff_meth(self->classname, str))
            body +=  "obj1->ins_" + actual_name  + "(obj2);\n";
         else
            body +=  "obj1->set_" + actual_name  + "(obj2);\n";
      }
      else
      {
         str = "insert_" + foreign.obj_ref->classname;
         if( exists_diff_meth(self->classname, str))
            body +=  "obj1->ins_" + foreign.obj_ref->classname  + "(obj2);\n";
         else
            body +=  "obj1->set_" + foreign.obj_ref->classname  + "(obj2);\n";
      }
   }
   else
   {

      met_name = attrname_of_superclass(self->classname,
                                        foreign.obj_ref->classname);
      /* handelt es sich um ein Attribut der Superklasse muss ein anderer Name
```

```
            gewaehlt werden */
         if( met_name != "")
         {
            str = "ins_" + met_name;
            if( exists_diff_meth(self->classname, str))
               body +=  "obj1->ins_" + met_name  + "(obj2);\n";
            else
               body +=  "obj1->set_" + met_name  + "(obj2);\n";
         }
      }

      if( is_attribute_in(foreign.obj_ref->classname, self->classname) != "")
         body += "obj2->set_" + self->classname + "(obj1);\n";
      else
      {
         met_name = attrname_of_superclass(foreign.obj_ref->classname,
                                           self->classname);
         if( met_name != "")
         {
            str = "ins_" + met_name;
            if( exists_diff_meth(self->classname, str))
               body +=  "obj2->ins_" + met_name  + "(obj1);\n";
            else
               body +=  "obj2->set_" + met_name  + "(obj1);\n";
         }
      }

      body += "validate;\n}\n}\n}";

   /* Kontrollausgabe */
   printf("signature: %s\n",signature);

   /* Kontrollausgabe */
   printf("body: %s\n",body);

   /* Methode uebersetzen */
   Schema->command( signature );
   Schema->command( body );
}

/* Signatur von insert_references erzeugen */
signature = "create method public insert_references in class "
     + tablename +";\n";

/* Kontrollausgabe */
```

```
   printf("signature: %s\n",signature);

   /* Alle Methoden, die in methodnames enthalten sind muessen von
      insert_references aufgerufen werden. */
   for ( helpstr in methodnames)
      methodcalls += "self->insert_" + helpstr + ";\n";

   /* Rumpf von insert_references zusammensetzen */
   body = "method body insert_references in class " + tablename
        +"\n{" + methodcalls +"}";

   /* Kontrollausgabe */
   printf("body: %s\n",body);

   /* insert_references uebersetzen */
   Schema->command( signature );
   Schema->command( body );
};
```

B.4 Die Methode redefine_set_attr

Bei der Abbildung von $m:n$-Beziehungen mit zusätzlichem Attribut in das objektorientierte Datenmodell wird das zusätzliche Attribut nur in eine der beiden Klassen aufgenommen. Für die Attribute, die diese Beziehungen in den Klassen abbilden, muß die Methode `set_`<*Attributname*> redefiniert werden. Dies wird von der Methode `redefine_set_attr` übernommen (siehe Abschnitt 6.2.2).

```
method public redefine_set_attr in class TabellenObj;
method body redefine_set_attr in class TabellenObj
{
   o2 Meta_definition def;
   o2 Meta_class m,sub;
   o2 tuple(name: string, visibility: string, type: Meta_definition) att;
   o2 string param;
   o2 list(string) parameter;
   o2 tuple(name: string, type: Meta_definition) attr;
   o2 tuple(name: string, type: Meta_definition) rem;
   o2 list(tuple(name: string, type: Meta_definition)) remember;
   o2 tuple(name: string,
            table: string,
            reference: string,
            obj_ref: TabellenObj) foreign;

   remember = list();
```

```
def = Schema->definition(self->classname);
if(def == nil)
{
   /* Es handelt sich um eine Zuordnungstabelle */
   if( self->foreign_keys == list())
   {
      /* Zuordnungstabellen ohne Fremdschluessel kann es nicht geben*/
      printf("Fehlerhaftes Tabellenobjekt!\n");
      abort;
   }
   else
   {

      /* Attribute der Tabelle, die nicht zum Fremdschluessel gehoeren,
         werden beim Methodenaufruf als Parameter mit uebergeben */
      parameter = self->return_param;
      /* Falls die Liste der Parameter leer ist, ist das ueberdefinieren
         der Methode nicht erforderlich */
      if(parameter != list())
      {
         for (param in parameter)
         {
            /* falls parameter gefunden wurden, wird der Datentyp
               der Parameter festgestellt */
            for(attr in self->get_attributes where attr.name == param)
            {
               rem.name =  param ;      /* Name des Parameters */
               rem.type = attr.type;   /* Datentyp von param */

               /* remember wird der Funktion zum redefinieren der Methode
                  mit uebergeben */
               remember += list(rem);
            }
         }
         /* Alle Fremdschluessel des Tabellenobjekts durchlaufen */
         for (foreign in self->foreign_keys)
         {
            def = Schema->definition(foreign.obj_ref->classname);
            if(( def != nil)&&(def->is_class))
            {
               m = (o2 Meta_class) def;
               /* Attribute der Klasse durchlaufen */
               for(att in m->attributes where att.name == self->classname)
               {
```

```
            /* Stimmt der Name eines Attributs mit dem Namen der Tabelle
               ueberein, muss die Methode neu definiert werden */
            redefine_method(att, m->name,remember);
         }
         /* Alle Subklassen werden durchlaufen, falls sie existieren */
         for(sub in m->subclasses)
         {
            for (att in sub->attributes where att.name == self->classname)
            {
            /* Stimmt der Name eines Attributs mit dem Namen der Tabelle
               ueberein, muss die Methode neu definiert werden */
               redefine_method(att, sub->name,remember);
            }
         }
      }
    }
  }
 }
}
};
```

Anhang C

Funktionen

C.1 Die Funktion `create_method`

Diese Funktion wird zum Generieren einer Methode `set_`<*Attributname*> aufgerufen. Der Funktion wird der Name der Klasse, für die die Methode generiert werden soll, sowie Datentyp und Name des Attributs übergeben. Die Funktion wird von der Methode `define_set_attr` aktiviert (siehe Abschnitt 6.2.2, Seite 55).

```
function body create_method(att: tuple(name: string,
                                       visibility: string,
                                       type: Meta_definition),
                            actual_class: string)
{
   o2 Meta_definition def;
   o2 Meta_type typ;
   o2 Meta_tuple tup;
   o2 Meta_collection col;
   o2 string  signature,sig, impl, body, cond, kind, methodname,help;
   o2 tuple(name: string, type: Meta_definition) attr;
   o2 boolean ignore;


   ignore = false;
   methodname = att.name;
   impl =  "self->" + att.name;
   sig =  "obj:";
   cond = "";

   /* Der Datentyp des Attributs ist eine Klasse */
   if((att.type)->is_class)
```

```
{
   /* Der Signatur wird der Name der Klasse hinzugefuegt */
   sig += (att.type)->name;

   /* in der Implementierung muss self->attname auf das
      Objekt gesetzt werden, das uebergeben wurde */
   impl += " = obj;\n";
}
else
{
   /* Die strukturierten Datentypen muessen einzeln
      bearbeitet werden. */
   def  =  att.type;
   typ = (o2 Meta_type) att.type;

   /* die Methode what_kind den Namen des Datentyps als string
      zurueck */
   kind = typ->what_kind;
   if( kind == "set")
   {
      /* Es handelt sich um eine Menge */
      col = (o2 Meta_collection) typ;

      /* Die Menge besteht aus Objekten einer Klasse */
      if ((col->element_type)->is_class)
      {
         /* Jedes Objekt soll nur einmal in der Menge
            enthalten sein, daher wird eine if-Abfrage
            eingebaut */
         cond = "if (!(obj in self->" + att.name + "))\n{";

         /* Der Methode wird nur ein Objekt der Klasse
            uebergeben. Der Klassenname dieses Objekts
            muss in der Signatur festgelegt werden. */
         sig +=  (col->element_type)->name;

         /* In der Implementierung wird das uebergebene
            Objekt der Menge hinzugefuegt. */
         impl += " += set(obj);\n}\n";
      }
      else
      {
         /* handelt es sich um eine Menge von Tupeln,
            wird diese Methode zunaechst ignoriert */
         ignore = true;
```

```
         def = (o2 Meta_definition) col->element_type;
         tup = (o2 Meta_tuple)(def);
         sig = make_sig_tuple(tup);
         impl += " += set(obj);\n";
      }
   }
   else
      if(kind == "list")
      {
         /* Es handelt sich um eine Liste */
         col = (o2 Meta_collection)typ;

         /* Der Datentyp der Elemente der Liste muss
            in der Signatur enthalten sein */
         sig +=   (col->element_type)->name;
         help = (col->element_type)->name;

         /* Die Position an der das Element eingefuegt
            werden soll, muss mituebergeben werden */
         sig += ", position: integer";

         /* Das Element wird an der position eingefuegt */
         impl += "[position] = obj;\n";
         /* der Name der Methode ergibt sich in diesem Fall
            aus set_ und dem Datentyp der Listenelemente */
         methodname =  (col->element_type)->name;
      }
      else
      {
         if(kind == "tuple")
         {
            /* Es handelt sich um ein Tupel
               (keine Menge) */
            tup = (o2 Meta_tuple)typ;
            /* der Methode wird ein Tupel uebergeben
               make_sig_tuple liefert den string, der
               fuer die Signatur erforderlich ist */
            sig = make_sig_tuple(tup);
            /* das uebergebene Tupel wird eingetragen */
            impl +=  " = obj;\n";
         }
         else
         {
            /* Es handelt sich um den Datentyp
               integer, real,... */
```

```
            /* In der Signatur wird der Name des
               Datentyps eingetragen */
            sig += (att.type)->name;
            /* Der Attributwert wird auf den Wert gesetzt */
             impl += " = obj;\n";
          }
        }
      }
   if (! ignore)
   {

      /* Die Signatur der Methode wird zusammengesetzt */
      signature = "method public set_" + methodname + "(";
      signature += sig +  ") in class " + actual_class + ";\n";

      /* Der Rumpf der Methode wird zusammengesetzt */
      body = "method body set_" + methodname + " in class " + actual_class;
      body += "{\n" + cond + impl + "};\n";

      /* Kontrollausgabe */
      printf("%s\n",signature);
      printf("%s\n",body);

      /* die Methode wird erzeugt */
      Schema->command(signature);
      Schema->command(body);
   }
};
```

C.2 Die Funktion `redefine_method`

Diese Funktion redefiniert eine Methode `set_`<*Attributname*> für Attribute, die $m : n$-Beziehungen mit Attribut abbilden (siehe Abschnitt 6.2.2, Seite 56). Der Funktion wird der Name der Klasse, sowie Name und Datentyp des Attributs, für das die Methode generiert werden soll übergeben. Das Attribut bzw. die Attribute der $m : n$-Beziehung werden der Funktion ebenfalls als Parameter übergeben. Die Funktion wird von der Methode `redefine_set_attr` aktiviert (siehe Anhang B.4).

```
function body redefine_method(att: tuple(name: string,
                                          visibility: string,
                                          type: Meta_definition),
                              actual_class: string,
                              remember: list(tuple(name: string,
```

```
                                                    type: Meta_definition)))
{   o2 Meta_definition def;
   o2 Meta_type typ;
   o2 Meta_tuple tup;
   o2 Meta_collection col;
   o2 boolean gefunden;
   o2 string  signature,sig, impl, body, kind, variable, attach, methodname,help;
   o2 tuple(name: string, type: Meta_definition) attr, diff;

   signature = "create method public set_" + att.name + "(";
   body = "method body set_" + att.name + " in class " + actual_class + "{\n";
   impl = "self->" + att.name;

   /* Der Datentyp des Attributs ist eine Klasse */
   if((att.type)->is_class)
   {
      /* Der Signatur wird der Name der Klasse hinzugefuegt */
      signature += "obj:" + (att.type)->name;

      /* in der Implementierung muss self->attname auf das
         Objekt gesetzt werden, das uebergeben wurde */
      impl += "= obj;\n";
   }
   else
   {
      /* Es handelt sich um einen strukturierten Datentyp.
         Daher ist att.type ist ein Objekt der Klasse Meta_type */
      typ = (o2 Meta_type)att.type;

      /* die Methode what_kind den Namen des Datentyps als string
         zurueck */
      kind = typ->what_kind;
      if( kind == "set")
      {
         col = (o2 Meta_collection) typ;
         /* Die Menge besteht aus Objekten einer Klasse */
         if((col->element_type)->is_class)
         {
            /* Der Methode wird nur ein Objekt der Klasse
               uebergeben. Der Klassenname dieses Objekts
               muss in der Signatur festgelegt werden. */
            signature += "obj:" + (col->element_type)->name;

            /* In der Implementierung wird das uebergebene
               Objekt der Menge hinzugefuegt. */
```

```
      impl += "+= set(obj);\n";

      /* Der Methode werden die verbleibenden Attribute der
         Zuordnungstabelle mit uebergeben, obwohl diese hier
         nicht verarbeitet werden. */
      for (attr in remember)
      {
         signature += "," + attr.name + ":" + attr.type->name;
      }

   }
   else
   {
      /* Die Menge besteht nicht aus Objekten einer Klasse,
         sondern aus Tupeln. */
      def = (o2 Meta_definition) col->element_type;
      tup = (o2 Meta_tuple) def;
      for (attr in tup->structure)
      {
         /* In der Implementierung wird eine Variable vom Typ tuple()
            benoetigt, die diesem Tupel entspricht. */
         variable += attr.name + ":" + attr.type->name + ",";
         /* Dem Tuple eintrag werden in der Implementierung die
            Werte zugewiesen, die uebergeben wurden */
         attach += "eintrag." + attr.name + " = " + attr.name + ";\n";
      }

      /* Der string variable enthaelt am Ende ein Komma zuviel */
      variable[count(variable)-1:]="";

      /* Dem string impl wird vorne die Variablendeklaration hinzugefuegt */
      impl[0:0] = "o2 tuple(" + variable + ")eintrag;\n" + attach + impl[0:0];
      /* Da es sich um eine Menge von Tuplen handelt, wird das Tupel eintrag
         der entsprechenden Menge zugewiesen werden. */
      impl += "+= set(eintrag);\n";

      for (attr in remember)
      {
         /* Falls remember noch weitere Attribute enthaelt, die noch nicht
           beruecksichtigt wurden, sind diese der Signatur hinzuzufuegen. */
         gefunden = false;
         for( diff in tup->structure)
         {
            if(diff.name == attr.name)
               gefunden = true;
```

```
          }
          if( gefunden == false)
             variable += "," + attr.name + ":" + attr.type->name;

        }
        signature += variable ;

      }
    }
  }
  /* Signatur zusammensetzen */
  signature += ") in class " + actual_class + ";\n";
  /* Rumpf zusammensetzen */
  body += impl + "\n}";

  /* Kontrollausgabe */
  printf("signature: %s\n",signature);
  printf("body: %s\n",body);

  /* Methode uebersetzen */
  Schema->command( signature );
  Schema->command( body );
};
```

Literaturverzeichnis

[BDK92] François Bancilhon, Claude Delobel, Paris Kanellakis: *Building an object-oriented Database System - The Story of O_2*. Morgan Kaufmann Publishers, 1992

[Beh91] W. Behme: *Objektorientierte Datenbanksysteme - Übersicht und Entwicklungen.* Hildesheim, 1991

[Dit90] K.R. Dittrich: *Objektorientiert, aktiv, erweiterbar: Stand und Tendenzen der „nachrelationalen“ Datenbanktechnologie.* Informationstechnik, 32.5, 1990

[DLR95] C. Delobel, C. Lécluse, P. Richard: *Databases: From Relational to Object-Oriented Systems.* International Thomson Publishing, 1995

[Heu92] Andreas Heuer: *Objektorientierte Datenbanken Konzepte, Modelle, Systeme.* Addison-Wesley, 1992

[HL82] R.L. Haskin und R.A. Lorie: *On extending the functions of relational database system.* In *PROC. ACM SIGMOD Conference on Management of Data*, Seiten 207-212, 1982

[Hug92] John G.Hughes: *Objektorientierte Datenbanken.* Carl Hanser Verlag, 1992

[Kin89] R. King: *My cat is object-oriented.* In [KL89], Kapitel 2, Seiten 23-30. Addison-Wesley, 1989

[KL89] W. Kim und F.H. Lochovsky, Herausgeber: *Object-Oriented Concepts, Databases and Applications.* ACM Press Frontier Series. Addison-Wesley, Reading, MA, 1989

[O2C] Herausgeber: O_2 Technology *O_2C Reference Manual.* Release 4.6 - November 1995

[O2K] Herausgeber: O_2 Technology *O_2Kit User Manual* Release 4.6 - September 1995

[O2K] Herausgeber: O_2 Technology *O_2Look User Manual* Release 4.6 - September 1995

[OQL] Herausgeber: O_2 Technology *OQL User Manual.* Release 4.6 - January 1996

[RW91] Bernd Rieper, Thomas Witte: *Grundwissen Produktion - Produktions- und Kostentheorie*. 3. Auflage, Osnabrück und Siegen, 1991