Logischer Entwurf und Dokumentation von objektrelationalen Oracle Datenbanken

Prof. Dr.(PL) Michael Unterstein
Fachhochschule Frankfurt/Main

Schlüsselworte

Objektrelationale Datenbank, Typenhierarchie

Einleitung

Der logische Entwurf und die Dokumentation objektrelationaler Datenbanken stellt besondere Anforderungen. Die mögliche Vielzahl benutzerdefinierter Datentypen mit mehrstufigen Abhängigkeiten und deren Nutzung in Objekttabellen kann schnell dazu führen, dass Entwickler und Anwender der Datenbank den Überblick verlieren.

UML-Klassendiagramme haben sich als Darstellungsmethode für den technologie-unabhängigen Entwurf von Datenstrukturen und Verhalten für Datenbanken aller Art und objektorientierte Programme durchgesetzt. Sie sind nur gedacht für und können nur beim *konzeptuellen* Entwurf helfen. Der logische Entwurf ist dann daraus abzuleiten und unterstellt die Möglichkeiten der für die Implementierung vorgesehenen technischen Plattformen, insbesondere das Datenmodell (relational, objektorientiert, objektrelational) und sogar deren herstellerspezifische Besonderheiten. Für den Entwurf objektrelationaler Datenbanken heißt das: die Umsetzung von Klassen und Assoziationen in Datentypen und deren Verwendung in weiteren Datentypen sowie in typisierten Objekttabellen erfordert weitergehende Entscheidungen des Entwicklers.

Der Vortrag erläutert anhand eines Beispiels für eine objektrelationale Datenbank, wie man mithilfe der Systemtabellen eine Dokumentation existenter Datentypen und ihrer Abhängigkeiten erzeugen kann. Es wird ferner diskutiert, warum Klassendiagramme nur bedingt brauchbar sind und eine grafische Notation für ein objektrelationales Typendiagramm vorgeschlagen. Die konkrete Umsetzung des logischen Entwurfs und der Implementierung werden am objektrelationalen Konzept von Oracle diskutiert. Wem der theoretische Teil zu wenig praktisch ist, der findet am Ende einen Hinweis, wie man bei Oracle durch Abfragen verschiedener Dictionary-Tabellen eine geschlossene Darstellung der Typabhängigkeiten findet, sodass die Struktur der fertigen Datenbank einigermaßen deutlich dokumentiert werden kann.

Objektorientierung im SQL-Standard

Im SQL-Standard (ISO 9075) wird Objektorientierung als eine Erweiterung des relationalen Konzepts verstanden, das sozusagen „aufwärtskompatibel“ zum relationalen Modell herkömmlicher Prägung ist. Dieser Ansatz wird als "objektrelational" bezeichnet. So erlaubt SQL die Konstruktion von Datentypen beliebig komplexer Struktur einschließlich der Definition von Methoden, also „abstrakte Datentypen“. Diese Datentypen können Attributen zugewiesen werden oder es können Objekttabellen mit einer modifizierten Form der CREATE TABLE-Anweisung auf Basis eines Datentyps definiert werden. Der Behälter für Objekte eines Objekttyps ist immer eine Tabelle, wenngleich diese sich durch Attribute mit komplexer Struktur erheblich von einer „flachen“ Tabelle, die die erste Normalform relationaler Datenbanken erfüllt, unterscheiden kann. Ausführlichere Abhandlungen über die Thematik finden sich beispielsweise in [TüSa06] und [MaUn08, Kapitel 9].

Wer eine Datenbank entwirft, braucht ...

eine Methode, um die Entwurfsobjekte, ihre Eigenschaften und ihre Beziehungen untereinander darzustellen. Dazu haben sich die UML Klassendiagramme als Standard durchgesetzt. Hier als kleines

Beispiel die Datenstrukturen eines Versandhandels.

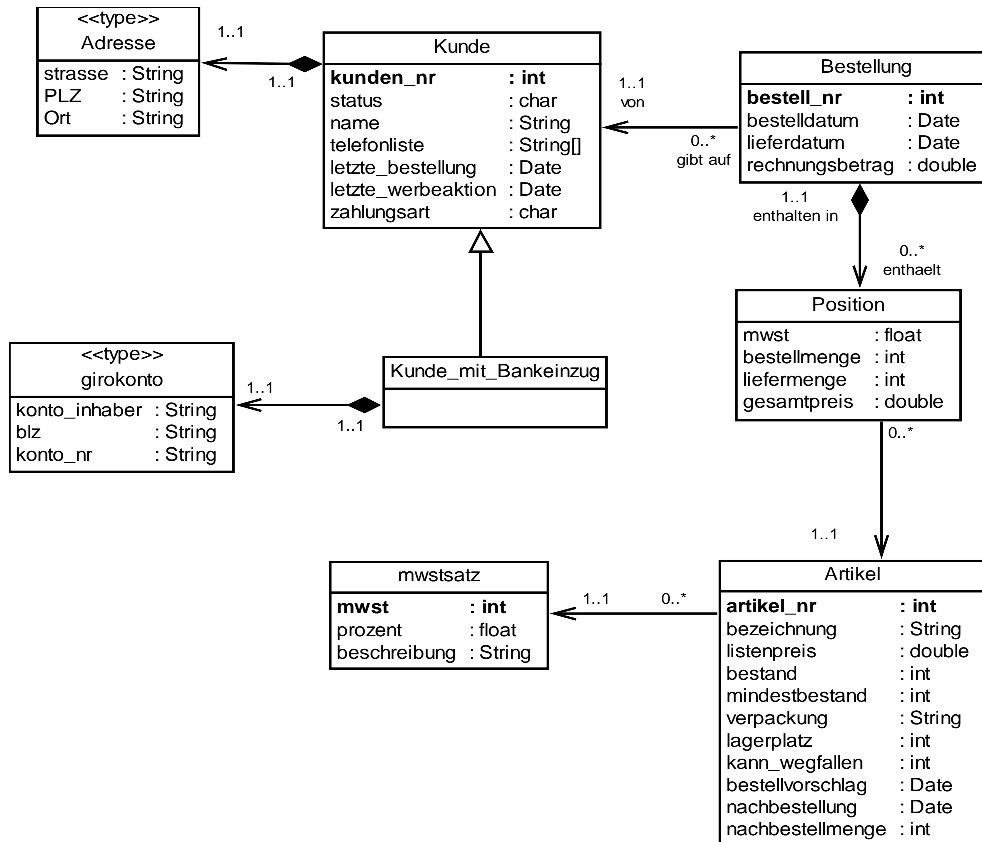

Abb. 1: UML Klassendiagramm

Zur Erläuterung des Klassendiagramms: Die Pfeile an den Beziehungslinien geben an, das man von einem Objekt zum anderen „navigieren“ kann, d.h. ohne explizite zusätzliche Lesezugriffe anzufordern. Beispielsweise können wir von Artikel zu mwst navigierend zugreifen, und zwar mit der „Punktnotation“, etwa in der Art: `artikel.mwst.prozent`. Den Typ `Adresse` benutzen wir lediglich als benutzerdefinierten Datentyp für das Attribut Adresse unserer Klasse Kunde. Adressen werden selbst nicht gespeichert. Wir haben dies durch den Stereotyp `<<type>>` kenntlich gemacht, die einzige Art, die UML zur Verfügung stellt, um solche Unterschiede deutlich zu machen. Sehr erhellend ist das aber nicht, da alle anderen Klassen natürlich auch Typen darstellen. Außerdem haben wir die Beziehung zwischen Bestellung und Position als Komposition dargestellt: Position ist ein Teil von Bestellung und eine Position kann ohne Bestellung nicht existieren. Auch die Beziehung zwischen Kunde und Adresse kann als Komposition (Teil-Ganzes-Beziehung mit Existenzabhängigkeit) aufgefasst werden, da eine Adresse Teil eines Kunden ist und nicht ohne diesen existieren kann. Das Attribut `telefonliste` der Klasse `Kunde` ist für die Aufnahme mehrerer Werte vorgesehen. Kenntlich ist dies an den eckigen Klammern, die implementierungstechnisch für ein `ARRAY` stehen. Dazu gleich mehr. `Kunde` wird spezialisiert zu `Kunde_mit_Bankeinzug`. Nur Objekte der Unterklasse haben eine Bankverbindung. Bei der Beziehung zwischen `Kunde_mit_Bankeinzug` und den Kontodaten handelt sich um eine Teil-Ganzes-Beziehung mit Existenzabhängigkeit, also eine Komposition. Bankverbindungen werden hier als Teil der Kunden verwaltet, da der Versandhandel keinen Wert darauf legt, einfach irgendwelche Daten über Konten zu sammeln.

Intent und Extent

Im objektorientierten Paradigma wird zwischen Intent und Extent einer Klasse unterschieden. Der

Intent bzw. die *intensionale Sicht* beschreibt die Struktur der Klasse, also deren Attribute und Methoden sowie Beziehungen zu anderen Klassen auf Typebene. Der *Extent* ist die Sammlung der Objekte einer Klasse. Die *extensionale Sicht* bezieht sich auf die Menge der Objekte und die Beziehungen zwischen Objekten.

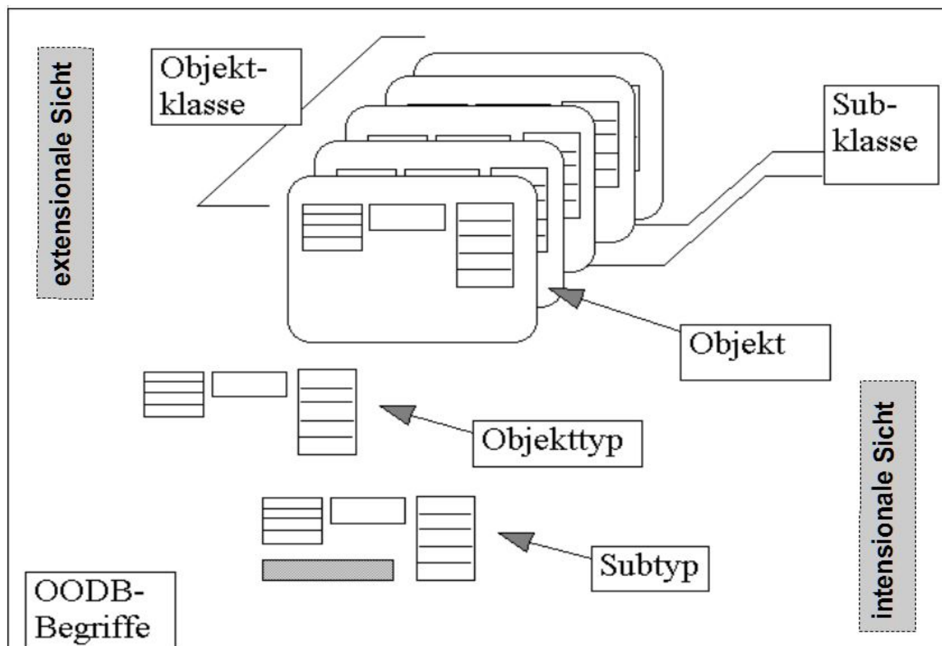

Abb 2.: Begriffe der Objektorientierung

In objektrelationalen Datenbanken ist beides deutlich voneinander getrennt. Die intensionale Sicht wird durch das Erzeugen von Typen mit der CREATE TYPE-Anweisung realisiert. Der Extent ist hingegen stets eine Tabelle, die auf einem benutzerdefinierten Typen basieren kann und wird mit der CREATE TABLE-Anweisung erzeugt.

So zum Beispiel wird ein Typ erzeugt (der bereits andere benutzerdefinierte Typen verwendet):

```
CREATE OR REPLACE TYPE otyp_kunde AS
OBJECT (
    kunden_nr                 number (4,0),
    status                    varchar2(1),
    name                      varchar2(30),
    telefonliste        varr_telefonliste,
    adresse             otyp_adresse,
    letzte_bestellung         date,
    letzte_werbeaktion        date,
    zahlungsart               char(1)
)
    NOT FINAL; -- Subtypen erlaubt
/
```

Und hier wird der Extent, eine typisierte Tabelle erzeugt:

```
CREATE TABLE otab_kunde OF otyp_kunde
       SUBSTITUTABLE AT ALL LEVELS
       (CONSTRAINT pk_kunde
        PRIMARY KEY (kunden_nr)
        -- weitere Constraints
           auf Extentebene);
```

Das Klassendiagramm zeigt jedoch intensionale und extensionale Sicht in einem. Die intensionale Sicht wird durch die Klassen als Typen, deren Attribute und Methoden und durch die Assoziationen angegeben. Die extensionale Sicht existiert darin, dass implizit die Klassensymbole auch für die Behälter der Objekte stehen. Außerdem gehören die Kardinalitäten an den Assoziationen zur extensionalen Sicht, da sie beschreiben, wieviele *Objekte* einer Klasse B mit jeweils einem Objekt der Klasse A in Beziehung stehen und umgekehrt. Die Vererbungsbeziehung vereint auf besondere Weise intensionale und extensionale Sicht: die Subklasse wird hinsichtlich der Strukturelemente „größer“, da sie alle Attribute und Methoden der Oberklasse „erbt“, sie ggf. überschreibt und weitere Attribute und Methoden hinzufügen kann. Extensional betrachtet, ist hingegen die Oberklasse „größer“, da alle Objekte der Unterklasse auch der Oberklasse angehören.

Verschiedene Möglichkeiten für die Umsetzung von Beziehungen

Für die Transformation des konzeptuellen Modells in Abb. 1 in objektrelationale Strukturen gibt es nun verschiedene Möglichkeiten, die Beziehungen zu realisieren. Wir können ein Objekt direkt in ein anderes einbetten, indem wir es als Datentyp für ein Attribut verwenden. Wir können es alternativ für sich bestehen lassen und von anderen Objekten mit einer Referenz darauf verweisen (Typkonstruktor REF). Wir können schließlich auch die relationalen Fremdschlüssel benutzen, was aber dem objektorientierten Paradigma widerspräche. Beziehungen mit Kardinalität „viele“ auf einer Seite und 1 auf der anderen Seite können analog dem Fremdschlüssel über eine Referenz auf das Objekt, das einmal an der Beziehung beteiligt ist, umgesetzt werden. Sie können auch von der anderen Seite her durch ein VARRAY oder eine eingebettete Tabelle realisiert werden. Schließlich lassen sich all diese Möglichkeiten kombinieren, z.B. sodass ein VARRAY Referenzen enthält usw. Die Kriterien für solche Entscheidungen sollen an dieser Stelle nicht vertiefend diskutiert werden. Sicher ist aber, dass eine Anwendung dieser Möglichkeiten dazu führt, dass die Struktur der Anwendungsdatenbank durch eine Vielzahl von benutzerdefinierten Datentypen geprägt wird, zwischen denen vielfältige Abhängigkeiten bestehen.

Die ganze im Klassendiagramm dargestellte Datenbank könnte etwa so in Typen umgesetzt werden, wobei wir die nutzerdefinierten Typen, die bei der Definition von anderen Typen angewandt werden hervorgehoben haben:

```
-- Anlage eines Typs fuer Adressen

CREATE OR REPLACE TYPE otyp_adresse AS
OBJECT (
          strasse         varchar2(30),
          plz             varchar2(5),
          ort             varchar2(30),
);

-- Anlage eines Typs fuer bis zu 5
-- Telefonnummern

CREATE OR REPLACE TYPE varr_telefonliste
AS VARRAY (5) OF VARCHAR2(20);
/
-- Anlage eines Typs fuer Kunden

CREATE OR REPLACE TYPE otyp_kunde AS
OBJECT
(
-- Zuvor definierte Typen werden benutzt.
```

```
-- Keine Constraints moeglich
    kunden_nr                 number (4,0),
    status                    varchar2(1),
    name                      varchar2(30),
    telefonliste        varr_telefonliste,
    adresse             otyp_adresse,
    letzte_bestellung         date,
    letzte_werbeaktion        date,
    zahlungsart               char(1)
)
    NOT FINAL; -- Subtypen erlaubt
/

-- Anlage eines Typs Bankverbindung zur
-- Erweiterung des Kundentyps bei
-- Bankeinzug

CREATE OR REPLACE TYPE otyp_girokonto AS
OBJECT
   (konto_inhaber varchar2(30),
    blz           varchar2(8),
    kontonr       varchar2(10)
);
/

-- Anlage des Subtyps für Kunden
-- mit Bankeinzug

CREATE OR REPLACE TYPE
otyp_kunde_mit_bankeinzug UNDER otyp_kunde
(
  bankverbindung otyp_girokonto
);
/

CREATE OR REPLACE TYPE otyp_mwstsatz AS
OBJECT (
        mwst  integer,
        prozent  number (3,3),
        beschreibung  varchar2(10)
        );

/

CREATE OR REPLACE TYPE otyp_artikel AS
OBJECT
     (artikel_nr       varchar2(4),
      mwst             REF otyp_mwstsatz,
      bezeichnung      varchar2(15),
      listenpreis      number(8,2),
      bestand          number(5,0),
      mindestbestand   number (5,0),
      verpackung       varchar2(15),
      lagerplatz       number(2,0),
      kann_wegfallen   number(1,0),
      bestellvorschlag date,
```

```
      nachbestellung   date,
      nachbestellmenge number(5,0),
      );
/

CREATE OR REPLACE TYPE otyp_position AS
OBJECT (pos_artikel  REF otyp_artikel,
        mwst         number (4,3),
      -- tatsaechlich angewandter
      -- MWST-Satz als Dezimalzahl
      bestellmenge   number (5,0),
      liefermenge    number(5,0),
      gesamtpreis    number(10,2),
      );
/
-- Anlegen eines Typs als eingebettete
-- Tabelle von Bestellpositionen

CREATE OR REPLACE TYPE ntyp_position
         AS TABLE OF otyp_position;
/

CREATE OR REPLACE TYPE otyp_bestellung AS
OBJECT (bestell_nr   number(6,0),
        bestellkunde REF otyp_kunde,
        bestelldatum date,
        lieferdatum  date,
        positionen   ntyp_position,
        rechnungsbetrag number(10,2)
        );
/
```

Für die Extents müssen dann jeweils Objekttabellen erzeugt werden (Auszug):

```
CREATE TABLE otab_kunde OF otyp_kunde
       SUBSTITUTABLE AT ALL LEVELS
       (CONSTRAINT pk_kunde
        PRIMARY KEY (kunden_nr)        );

CREATE TABLE otab_mwstsatz OF
otyp_mwstsatz
(
  mwst NOT NULL,
  prozent NOT NULL,
  CONSTRAINT pk_mwstsatz
      PRIMARY KEY (mwst)
)
/
CREATE TABLE otab_artikel OF otyp_artikel
(CONSTRAINT pk_artikel PRIMARY KEY
            (artikel_nr)
-- weitere constraints
);
```

```
CREATE TABLE otab_bestellung OF
otyp_bestellung
      NESTED TABLE positionen STORE AS
      ntab_position;
```

Die Anwendung benutzerdefinierter Typen bei der Definition weiterer Typen kann mehrstufig erfolgen, sodass Typ C Typ B benutzt, der auf Typ A basiert u.s.w. Da kann die Übersicht leicht verloren gehen. Eine schwache Hilfe ist die konsequente Einhaltung von Benennungskonventionen, so wie im Beispiel.

Plädoyer für ein eigenes Datentypenmodell

Die in den Listings gezeigte Umsetzung des in Form des Klassendiagramms (Abb 2) existierenden konzeptuellen Modells stellt eine von verschiedenen Möglichkeiten dar, keinesfalls die einzige. Wir sehen darin einen Grund, für ein eigenes Typmodell zu plädieren, das als Bauplan für die CREATE TYPE Anweisungen verstanden werden soll - und natürlich genauso der nachträglichen Dokumentation dienen kann. Einige Argumente mögen dies untermauern:

- Auf Basis eines abstrakten Datentyps können mehrere verschiedene Tabellen als Extent erzeugt werden.
- zwischen Typen sind andere (weniger) Beziehungen möglich als zwischen Klassen (im Wesentlichen „verwendet“, referenziert, „spezialisiert“)
- Die Beziehungen zwischen Extents sind mengenmäßig zu verstehen, Beziehungen zwischen Typen nicht. Die Semantik von Typmodell und Extentmodell sind also unterschiedlich.
- Bei der Erstellung der Strukturen in einer Objektrelationalen Datenbank unter SQL sind CREATE TYPE und CREATE TABLE ... OF zwei verschiedene Operationen, die in dieser Reihenfolge auszuführen sind.
- Jedes Modell für sich wird übersichtlicher, als wenn die Frage, ob zu einem Typ ein Extent erzeugt wird, durch ein Attribut wie „persistent“ gesteuert wird (vgl. Entwurfswerkzeuge wie Power Designer, die so verfahren).
- Wichtige Eigenschaften des Datenmodells wie Primary Keys und weitere deklarative Einschränkungen können nur auf Basis des Extents festgelegt werden. (Das ist allerdings ein Mangel an SQL).
- Schließlich erscheint eine grafische Visualisierung der Abhängigkeiten von Typen notwendig, um den Überblick über die Datenstrukturen zu behalten bzw. zu ermöglichen. Das Klassendiagramm leistet dies nicht.

Wir schlagen eine eigene Sorte von Diagrammen vor, das Datentypendiagramm. Soweit wie möglich, verwenden wir die UML-Notation. In Anlehnung an UML steht das Klassensymbol für einen Typen, allerdings mit speziellen Abwandlungen, um Arrays und eingebettete Tabellen von Objekttypen zu unterscheiden. Die Darstellung enthält noch viel Verbesserungspotenzial - der Autor ist offen für Vorschläge. Abb. 3 zeigt einen ersten Ausschnitt aus dem logischen Datenmodell.

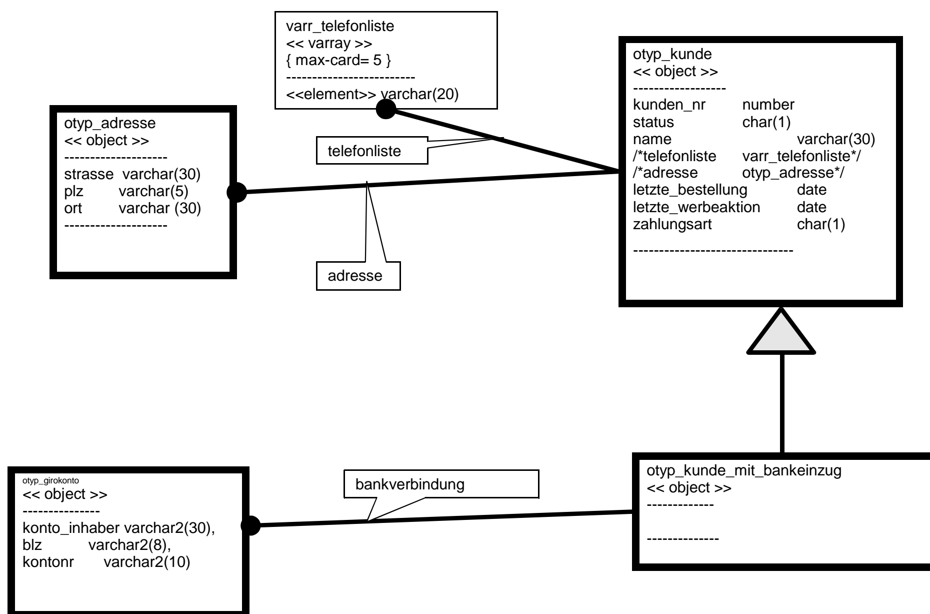

Abb. 3: Beziehungen zwischen Typen, erster Teil

Wie im Klassendiagramm ist das Symbol für einen abstrakten Datentyp unterteilt in drei Teile, oben Name und ggf. erläuternde Angaben wie Stereotyp (in doppelten spitzen Klammern) und Zusicherungen (in einfachen geschweiften Klammern). Der mittlere Teil enthält die Attribute respektive Felder. Der untere Teil ist für die Signaturen der Methoden vorgesehen, auf die wir hier jedoch nicht eingehen. Weiter benutzen wir Kommentare:

/* schließt einen Kommentar ein */

Standarddatentypen werden im Typsymbol direkt hinter dem Attributbezeichner angegeben

Typen die als Datentyp von Attributen verwendet werden, werden mit dem Typ, dem das Attribut gehört, über eine gerichtete Beziehung verbunden. Der Attributbezeichner wird wie eine Rolle als Eigenschaft der Beziehung notiert. Zusätzlich können Attribut und Datentyp im Typsymbol auskommentiert dargestellt werden.

Für Varrays gilt speziell: Die Notation muss einen Datentypbezeichner einschließen, aber keine Attributbezeichner, da die Elemente nicht benannt sind. Die maximale Kardinalität ist eine Eigenschaft des Datentyps. Sie wird daher innerhalb des Symbols dargestellt.

Abb. 4 beschreibt die verwendete Syntax der Pfeile.

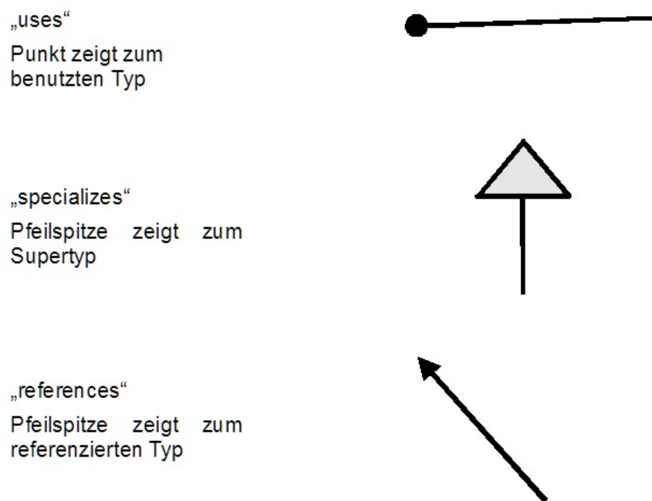

Abb. 4 Verwendete Beziehungssymbole

Die Beziehungslinien unterscheiden sich in drei Typen. Die einfache Verwendung eines Typs als Datentyp für ein Attribut eines anderen Typs bezeichnen wir als *„uses“*, die Referenzierung eines Typs durch einen anderen bezeichnen wir als *„references“* und die Spezialisierung (Subtypenbildung) wird mit dem aus Klassendiagrammen bekannten Symbol dargestellt und ggf. zusätzlich als *„specializes“* bezeichnet. Alle drei Beziehungsarten sind gerichtet, was normale Assoziationen in UML-Diagrammen nicht sind. Kardinalitäten werden grundsätzlich nicht dargestellt, da sie - wie oben begründet - in die extensionale Sicht gehören. Allerdings sind Arrays und nested Tables von sich aus Typen, die für die Aufnahme mehrerer gleichartiger Werte zuständig sind.

In Abb. 5 wird der „Rest“ des logischen Datenmodells angezeigt. Hier sind die Referenzen als spezielle Beziehungstypen verwendet worden, und das Symbol für die eingebettete Tabelle weicht ab von dem Symbol eines strukturierten Datentyps. Die Sprechblasen dienen der Erläuterung und könnten entfallen. Alternativ könnte man die referenzierenden Attribute in den Klassensymbolen weglassen und die Benennung der Pfeile beibehalten.

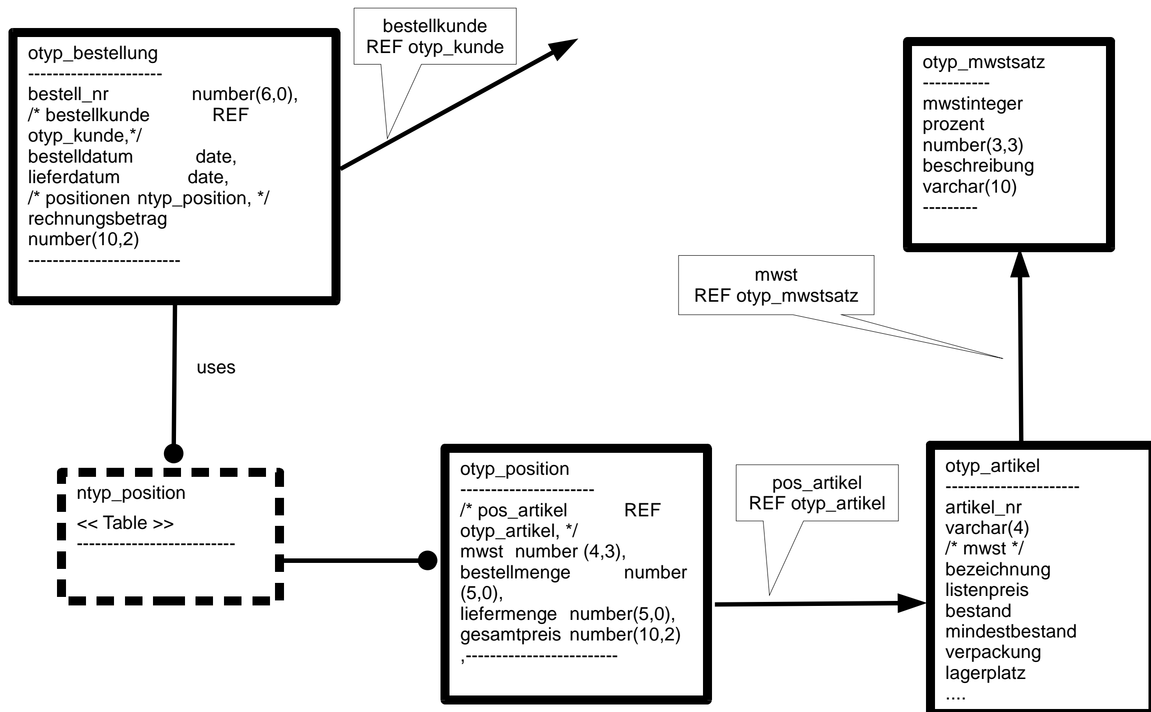

Abb. 5: Datentypendiagramm Teil 2

Die verwendeten Symbole für Arten von Datentypen und die Beziehungsarten sind lediglich Vorschläge. Wichtig ist nur, dass sie voneinander unterscheidbar sind und dass man bei den Beziehungen die Richtung erkennt.

Wer schon eine objektrelationale Datenbank hat, braucht ...

... eine Dokumentation derselben, sodass Informationen darüber, welche Typen es gibt, wie diese strukturiert sind und wie sie benutzt werden, verfügbar sind. Unser Ziel ist eine geschlossene Darstellung aller Abhängigkeiten der selbstdefinierten Typen, nach Möglichkeit so, dass durch entsprechende Einrückungen deutlich wird, welche Typen von welchen anderen Typen in der Weise abhängen, dass sie sie benutzen, darauf mit einer Referenz verweisen oder dass sie sie spezialisieren. Abb. 6 zeigt diese Auswertung. Die Frage ist nur, wie man sie bekommt.

Selbstverständlich werden in einem objektrelationalen Datenbanksystem die Metadaten über benutzerdefinierte Datentypen und ihre Verwendung im Dictionary gehalten. Verschiedene Views stehen dafür zur Verfügung, die von dem Besitzer des Schemas, zu dem die Typen und Objekttabellen gehören, abgefragt werden können.

Der View `user_types` zeigt alle benutzerdefinierten Datentypen an, wobei neben anderen Informationen auch der Supertyp erkennbar ist, sofern der abgefragte Typ eine Spezialisierung darstellt. Die Struktur eines Typen, d.h. seine Attribute, werden in `user_type_attrs` wiedergegeben. Dabei erscheinen alle Attribute, diejenigen mit Standarddatentypen ebenso wie die, deren Datentyp ein Objekttyp ist. Der View `user_dependencies` schließlich zeigt alle Abhängigkeiten innerhalb des Typsystems an, wozu auch die Verwendung von Objekttypen in Objekttabellen gehört, und dass ein Type Body (die Implementation der Methoden) von der Typdeklaration abhängig ist etc. Was dieser View nicht zeigt, sind Typhierarchien im Sinne der

Subtypenbildung. Um die folgende Ausgabe zu erhalten, sind ein paar Vorarbeiten notwendig.

```
TYPENAME                                 REFERENCED_NAME      DEPE  LEVEL
---------------------------------------- -------------------- ---- ------
OTYP_ADRESSE                                                            1
      OTYP_KUNDE                         OTYP_ADRESSE         HARD      2
            OTYP_BESTELLUNG              OTYP_KUNDE           REF       3
            OTYP_KUNDE_MIT_BANKEINZUG    OTYP_KUNDE           SPEC      3
      OTYP_KUNDE_MIT_BANKEINZUG          OTYP_ADRESSE         HARD      2
OTYP_GIROKONTO                                                          1
      OTYP_KUNDE_MIT_BANKEINZUG          OTYP_GIROKONTO       HARD      2
OTYP_MWSTSATZ                                                           1
      OTYP_ARTIKEL                       OTYP_MWSTSATZ        REF       2
            OTYP_POSITION                OTYP_ARTIKEL         REF       3
                  NTYP_POSITION          OTYP_POSITION        HARD      4
                        OTYP_BESTELLUNG  NTYP_POSITION        HARD      5
VARR_TELEFONLISTE                                                       1
      OTYP_KUNDE                         VARR_TELEFONLISTE    HARD      2
            OTYP_BESTELLUNG              OTYP_KUNDE           REF       3
            OTYP_KUNDE_MIT_BANKEINZUG    OTYP_KUNDE           SPEC      3
      OTYP_KUNDE_MIT_BANKEINZUG          VARR_TELEFONLISTE    HARD      2
```

Das folgende Listing zeigt die Definition eines VIEW, der bereits die nötigen Daten enthält. Die Definition beruht auf XE Version 11; für andere Versionen können leichte Abänderungen nötig sein. Für die strukturierte Ausgabe brauchen wir dann noch eine Abfrageanweisung, die anschließend wiedergegeben wird.

```
CREATE OR REPLACE VIEW
user_dependency_ext2 AS
SELECT name, referenced_name,
      dependency_type
FROM USER_DEPENDENCIES ud1
WHERE TYPE = 'TYPE'
     AND referenced_owner = 'OOCHEF'
     AND type NOT LIKE '%BODY%'
     AND NOT EXISTS
        (SELECT * FROM user_types u2
         WHERE  ud1.name = u2.type_name
         AND    ud1.referenced_name =
                u2.supertype_name)
UNION
SELECT type_name, supertype_name, 'SPEC'
FROM   user_types
WHERE  supertype_name IS NOT NULL
UNION
SELECT type_name, NULL, NULL
FROM   user_types ut
```

```
WHERE  NOT EXISTS
          (SELECT *
           FROM USER_DEPENDENCIES ud
           WHERE ut.type_name = ud.name
           AND   type NOT LIKE '%BODY%'
           AND   referenced_owner='OOCHEF'
);
-- OOCHEF ist hier der Name des Schemas
-- unter Oracle XE kann die Systemvariable
-- USER nicht verwendet werden.
-- Ggf. den Benutzernamen explizit angeben
```

Die den View erzeugende Abfrage besteht aus drei Teilen, die per `UNION` verbunden werden. Der erste Teil fragt die Tabelle `user_dependencies` ab und zeigt alle Typen, die dem Benutzer „gehören", wobei Abhängigkeiten des Typ Body vom Typ ausgeblendet werden. Weggelassen werden auch die Typen, die durch Spezialisierung aus anderen Typen hervorgegangen sind. Diese werden im zweiten Teil hinzugefügt. Für das Attribut `dependency_type` geben wir dabei den Wert `'SPEC'` aus. Der letzte Teil zeigt die „Spitze" der Abhängigkeitshierarchie, nämlich die Typen, die in `user_dependencies` nicht vorkommen, weil sie nur Standarddatentypen benutzen. Dazu gehört beispielsweise `otyp_adresse`, der nur Attribute von Typ `VARCHAR2` hat.

Die Ausgabe erfolgt schließlich mit dem folgenden Befehl, wobei mit der CONNECT BY-Klausel die innerhalb des VIEW enthaltene Hierarchie zwischen name und referenced_name ausgewertet wird.

```
SELECT LPAD(' ', 6*(level-1)) || name AS
typename, referenced_name,
dependency_type, level
FROM user_dependency_ext2
START WITH referenced_name IS NULL
CONNECT BY PRIOR name = referenced_name;
```

Literatur:

[MaUn08] Matthiessen, Unterstein: Relationale Datenbanken und Standard SQL. Addison Wesley, 4. Auflage 2008

[TüSa06] Türker, Saake: Objektrelationale Datenbanken. Dpunkt, 1. Auflage 2006

Kontaktadresse:

Prof. Dr.(PL) Michael Unterstein
Fachhochschule Frankfurt - University of Applied Sciences
Nibelungenplatz 1
D-60318 Frankfurt

E-Mail ustein@fb3.fh-frankfurt.de

Die SQL-Skriptdateien können auf Anfrage per Email bezogen werden