Wie die Objektorientierung relationaler werden sollte

Eine Analyse aus Sicht der Datenmodellierung

Dilek Stadtler[1] und Friedrich Steimann[2]

Abstract: Die Kluft zwischen der relationalen Sichtweise auf Daten, wie sie u. a. in UML-Klassendiagrammen ausgedrückt werden kann, und der objektorientierten Programmierung ist seit langem Gegenstand von Untersuchungen. Angesichts jüngster Bestrebungen, relationale Elemente in die objektorientierte Programmierung einzubringen (wie etwa mit Microsofts LINQ-Projekt), zeigen wir auf, wie das Modell der objektorientierten Programmierung erweitert werden müßte, so daß sich auch die relationalen Teile eines Klassendiagramms direkt, d. h. unter Verzicht auf Transformationen, die umfangreichen stereotypen Code einführen, in objektorientierte Programme umsetzen lassen, ohne daß das objektorientierte Programmiermodell dadurch seinen navigierenden Charakter verlieren würde. Dabei beschränken wir uns hier auf die Ableitung der Anforderungen an eine solche Erweiterung — eine konkrete Umsetzung ist in einer parallelen Arbeit beschrieben.

1 Einleitung

Vor dem Aufkommen der objektorientierten Modellierung und ihrer Notationen dominierte für geraume Zeit das Entity-Relationship-Modell (ERM) [Ch76] die Datenmodellierung. Seine Vorteile sind die direkte Übersetzbarkeit in das relationale Datenmodell (RDM, einschließlich seiner mathematische Fundierung, dem Relationenkalkül [Co70]) sowie die generelle Anwendungsunabhängigkeit seiner Modelle: Mit der Vorgabe der Entitäts- und Beziehungstypen[3] wird lediglich ein logischer Sachverhalt wiedergegeben, der von beliebigen Anwendungen beliebig genutzt werden kann. Insbesondere sind durch Relationen keine Navigationsrichtungen vorgesehen, so daß Daten mittels relationaler Abfragesprachen in jede Richtung ausgewertet werden können.

Mit dem Aufkommen der objektorientierten Programmierung begann sich die Sicht auf die Daten wieder zu ändern. Zwar basiert die Objektorientierung auf Klassen, deren Datenanteil im wesentlichen den Entitätstypen des ERM entspricht, aber die Beziehungen zwischen Klassen werden durch Zeiger dargestellt und sind damit grundsätzlich gerichtet. Das objektorientierte Datenmodell (OODM), welches als die Projektion des objektorientierten Programmiermodells auf seine Datenanteile verstanden werden kann

[1] Lehrgebiet Programmiersysteme, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen, dilek.stadtler@fernuni-hagen.de

[2] Lehrgebiet Programmiersysteme, Fakultät für Mathematik und Informatik, Fernuniversität in Hagen, steimann@acm.org

[3] Wir übersetzen hier „Relationship“ mit „Beziehung“. Zu Codds Unterscheidung der Begriffe „Relation“ und „Relationship“ s. Abschnitt 3.3.1.

(also im wesentlichen als Klassen mit Attributen, ohne Methoden, jedoch mit Vererbung), ist damit aber näher am (überkommen geglaubten) Netzwerkmodell als am ERM, was auch daran erkennbar ist, daß die Auswertung der Daten typischerweise nicht mehr deklarativ wie im relationalen Fall erfolgt, sondern imperativ mittels Methoden.

Seither gibt es immer wieder Bestrebungen, den Rückschritt, den die Objektorientierung in Sachen Datenmodellierung bedeutet, zu korrigieren und das OODM um Relationen zu

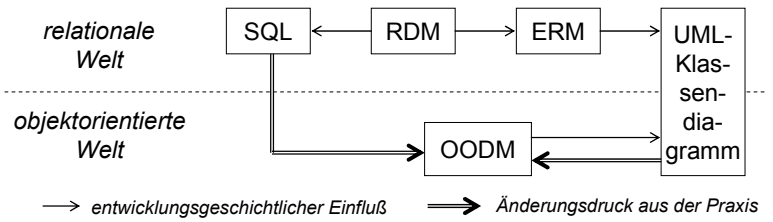

Abb. 1: Beziehung der in dieser Arbeit herangezogenen Modelle und deren Notationen. Wir leiten aus dem Änderungsdruck unter Berücksichtigung der Entwicklungsgeschichte eine sanfte Erweiterung des OODMs um relationale Aspekte ab.

erweitern (s. z.B. [BW05, NPN08, Øs07, Ru87]). Wir sehen hierfür vor allem zwei Beweggründe (zum ursächlichen, entwicklungsgeschichtlichen Zusammenhang s. Abbildung 1):

1. Wichtige objektorientierte Modellierungssprachen wie die UML setzen in ihren statischen Strukturbeschreibungen (das sind im wesentlichen Klassendiagramme) auf Relationen als neben Klassen gleichberechtigte Modellierungskonstrukte [OMG09a, OMG09b]. Daß sich diese in objektorientierten Programmen nicht unmittelbar wiederfinden, führt zu Problemen beim Übergang von Modellen zu Programmen und umgekehrt (s. z.B. [AHM07, Ge09, GRL03, No97]).

2. Die Datenhaltung erfolgt aus technischen Gründen derzeit noch vor allem mittels (objekt)relationaler Datenbanken (s. z. B. [MM95]). Selbst wenn die Probleme der transparenten Übertragung von Objekten aus objektorientierten Programmen in solche Datenbanken inzwischen weitgehend gelöst sind, so stören sich aus der relationalen Datenbankwelt kommende Programmierer doch an imperativ ausformulierten Datenauswertungen auf der Programmseite und vermissen die Prägnanz relationaler Abfragesprachen wie der SQL [ISO08].

In dieser Arbeit versuchen wir herauszuarbeiten, welche relationalen Ergänzungen des OODMs durch diese Argumente tatsächlich begründet werden können. Wie sich herausstellen wird, lassen sich die in der Praxis wichtigsten Anforderungen in eine vergleichsweise behutsame Erweiterung des OODMs transformieren, die insbesondere dessen zeigerbasierten Charakter voll erhält. Die Umsetzung dieser Erweiterung zeigen wir aufgrund ihrer Ferne zur Modellierung hier jedoch nicht; sie ist in einer parallelen Arbeit beschrieben [SS10].

Der Rest dieser Arbeit ist wie folgt gegliedert: In Abschnitt 2 diskutieren wir kurz die verwandten Arbeiten, bevor wir in Abschnitt 3 untersuchen, welche Elemente relationalen Ursprungs das UML-Klassendiagramm enthält, die nicht zugleich im OODM vorkommen, und welche Relevanz diese Elemente in der Praxis haben. In Abschnitt 4 untersuchen wir, wie viel „Relationalität" relationale Abfragesprachen vom OODM verlangen und stellen fest, daß dies lediglich eine Anforderung, die schon eine relationale Datenmodellierung an das OODM stellt, bestärkt. Wir schließen mit der Übertragung der Anforderungen in die Skizze einer sanften Erweiterung des OODMs, die einen guten Teil der Erwartungen an ein „OODM mit Relationen" bedienen können sollte.

2 Verwandte Arbeiten

Daß dem OODM Relationen fehlen, ist in der Literatur weitgehend unbestritten. Auch bei den Arbeiten dazu, wie sie eingebracht werden können, herrscht große Einigkeit: Sie lassen sich grob in solche einteilen, die Relationen mit stereotypen Codefragmenten emulieren (dazu gehören auch die Transformationsansätze aus dem Model-driven-Bereich), und in solche, die das OODM erweitern wollen (wozu auch die unsere zählt).

Noble beschreibt gleich eine ganze Reihe von Mustern zur Umsetzung von allgemeinen Relationen mit Hilfe der durch objektorientierte Programmiersprachen vorgegebenen Mittel [No97]. Gängiger Programmierpraxis folgend werden bei ihm unidirektionale und bidirektionale 1:1- und 1:*n*-Beziehungen mit Hilfe von Attributen und Collections umgesetzt. Für komplexere Beziehungen schlägt [No97] die Realisierung anhand von „relationship objects", also der Reifizierung der Beziehungen als Klasse/n, vor.

Einen Schritt weiter gehen Genova et al., indem sie einen Generator für die Umsetzung von binären UML-Assoziationen (inkl. Multiplizität, Sichtbarkeit und Navigierbarkeit) in Java-Codemuster vorstellen [GRL03]. Die Muster basieren auf einer Reifizierung von Beziehungen als Klassen („reified tuples"). Andere Konzepte wie z.B. höherstellige Beziehungen, Assoziationsklassen etc. werden jedoch nicht behandelt.

Auch Akehurst et al. betrachten die Problematik aus Sicht der MDA, fokussieren in ihrer Arbeit aber besonders auf in UML 2.0 neu definierte Konzepte und deren Realisierung in den neuen Sprachkonstrukten von Java 5 [AHM07]. Über andere Arbeiten hinausgehend schlagen sie auch eine Umsetzung von qualifizierten Assoziationen und eine Möglichkeit zur Einhaltung von unteren Schranken (Multiplizitäten) vor. Gessenharter geht noch einen Schritt weiter und löst das Problem der Kombination von Navigierbarkeit und Sichtbarkeit [Ge09]. Er widerlegt damit die Behauptung von [GRL03], daß dies mit reifizierten Beziehungen nicht möglich sei. Auf einer höheren Ebene befaßt sich die Arbeit von Amelunxen et al. mit der Umsetzung von Assoziationen in Java, indem sie die Sicht des Metamodells MOF 2.0 ein- und sich da insbesondere die assoziationsabhängigen Konzepte Union, Redefinition und Subset vornimmt [ABS04].

Während all diese Arbeiten im wesentlichen Relationen aus (mentalen oder reellen) Modellen in objektorientierte Codefragmente transformieren, die vom Programmierer später weiterzupflegen sind, erlauben es Bibliotheken wie die von Østerbye [Øs07], Relationen — analog zu Collections — als Klassen in ein Programm zu importieren. Allerdings unterliegt die Verwendung solcher Bibliotheksrelationen gewissen Konventionen, die wiederum zu stereotypem Code führen. Insbesondere lassen sich damit programmiersprachliche Primitive wie beispielsweise die Zuweisung nicht modifizieren.

Diese Beschränkung vermeiden Spracherweiterungen wie beispielsweise die von Schmidt et al., Rumbaugh oder die von Bierman und Wren. Schmidt et al. haben in verschiedenen Arbeiten den Einsatz von relationalen Datenbankprogrammiersprachen (Sprachen, in denen programmiersprachliche und datenbanksprachliche Konzepte miteinander verschmolzen werden), die das Relationenmodell in ihr Typsystem integrieren, untersucht. Bedeutende Arbeiten in diesem Bereich sind beispielsweise Pascal/R [Sc77] und DBPL [SM92]. Beide Sprachen haben gemeinsam, das Konzept der Relation im Sinne einer Datenbankrelation (mitsamt der Definition von Schlüsseln) als eigenen Datentyp in eine höhere Programmiersprache zu integrieren. Die jeweilige Programmiersprache wird dabei in vergleichsweise hohem Maße erweitert. Einen ebenfalls von Schmidt et al. entwickelten, bibliotheksbasierten Ansatz, stellt das Tycoon-System [MS93] dar. Das Tycoon-System ist eine Programmierumgebung zur Konstruktion von persistenten Objektsystemen, welche, je nach Version, auf der strikt typisierten, polymorphen Programmiersprachen Tycoon Language (TL) [MS92] oder der objektorientierte Variante, Tycoon Object-Oriented Language (TooL) [GM96], basiert. Mit diesem Ansatz wurde ein System zur Verfügung gestellt, dessen Fokus auf der bestmöglichen Erweiterbarkeit eines minimalen vordefinierten Sprachkerns liegt, und somit die Möglichkeit bietet, Datenbank-Funktionalitäten auf unterschiedlichste Weise zu integrieren.

Rumbaugh [Ru87] und Bierman und Wren [BW05] führen dagegen jeweils Relationen als native programmiersprachliche Konstrukte auf gleicher Ebene mit Klassen ein. Die beiden Arbeiten repräsentieren dabei zwei grundsätzlich verschiedene Auffassungen von Relationen als Typen: Für Rumbaugh sind Relationen Tupelmengen, die in Instanzen eines Typs `Relation` gehalten werden, während für Bierman und Wren Relationen die Typdeklarationen von Tupeln sind (und die Instanzen von Relationen entsprechend einzelne Tupel). Beiden Arbeiten gemein ist wiederum, daß Tupel an zentraler Stelle, also insbesondere außerhalb der Objekte, die ihre Komponenten ausmachen, gehalten werden. Im Gegensatz dazu favorisieren wir einen Ansatz, bei dem Tupel an die Komponenten gebunden sind. Die globale Extension einer Relation ergibt sich damit als die Vereinigung der Tupel aller beteiligten Objekte.

3 Notwendige relationale Ergänzungen des objektorientierten Datenmodells aus Modellierungssicht

In diesem Abschnitt versuchen wir, mögliche Defizite des OODMs aus Modellierungssicht systematisch herzuleiten. Wir gehen zu diesem Zweck davon aus, daß das UML-Klassendiagramm (begründet durch den entwicklungsgeschichtlichen Einfluß; s. Abbildung 1) ein weitgehend konsensuelles Datenmodell repräsentiert, daß es also insbesondere was die Modellierung mit Relationen angeht, nichts Wesentliches vermissen läßt, und setzen die Modellierungssicht auf objektorientierte Daten mit dem UML-Klassendiagramm gleich. Diese Annahme erlaubt uns, aus dem Aufbau des Diagramms (richtiger: des Diagrammtyps) sowie der Herkunft seiner Elemente entweder aus dem ERM (als Stellvertreter einer relationalen Sichtweise) oder aus dem OODM[4] systematisch abzuleiten, was dem OODM in seiner Datensicht an Relationalem möglicherweise fehlen könnte. Ob es ihm tatsächlich fehlt, versuchen wir anschließend durch die Einbeziehung der Modellierungspraxis zu klären und durch Betrachtungen, inwieweit das OODM ohne Erweiterungen in der Lage ist, für brauchbaren Ersatz zu sorgen. Doch bevor wir uns diesen Fragestellungen zuwenden, rekapitulieren wir kurz das ERM und das OODM.

3.1 Das Entity-Relationship- und das objektorientierte Datenmodell

Das ERM besteht in seiner ursprünglichen Form aus Entitätstypen, Beziehungstypen, Attributdeklarationen für beide sowie der Angabe von Kardinalitäten für Attribute die Stellen eines Beziehungstyps. Entitätstypen werden unterschieden in starke und schwache: Erstere sind durch eine Menge von Attributen als Primärschlüssel eindeutig identifizierbar, letztere nicht. Ergänzen läßt sich das ERM durch eine Reihe von generischen Operationen zum Einfügen, Ändern und Löschen von Daten.

Auch wenn das ERM vielleicht nicht als speziell auf das RDM Codds [Co70] zugeschnitten erscheinen sollte, so weist es doch starke Ähnlichkeiten mit ihm auf. Gleichwohl ist das ERM u. a. aufgrund seiner Unterscheidung von Entitäts- und Beziehungstypen differenzierter als das RDM — tatsächlich kann das RDM aus Modellierungssicht sogar als unzureichend angesehen werden, weil es nicht zwischen Entitäts- und Beziehungstypen unterscheiden kann und damit eine fundamentale semantische Differenzierung nicht auszudrücken erlaubt. Wenn wir also im folgenden davon sprechen, daß die Objektorientierung relationaler werden sollte, meinen wir damit immer relationaler im Sinne des ERMs und nicht im Sinne des puristischen RDMs.

[4] Genaugenommen gibt es *das* OODM gar nicht — zumindest ist eine den anderen Datenmodellen vergleichbar allgemein akzeptierte Definition in der Literatur nicht zu finden. Wie bereits weiter oben beschrieben, kann es jedoch im wesentlichen als Projektion des objektorientierten Programmiermodells auf seine Datenanteile verstanden werden.

Dem ERM gegenüber steht das OODM. Seine Klassen entsprechen zwar für sich genommen im wesentlichen den Entitätstypen des ERMs (zumal sie wie letztere über Attribute zur Beschreibung ihrer Instanzen, Objekte genannt, verfügen), jedoch kennt das OODM anders als das ERM keine Relationen: Beziehungen zwischen Objekten werden ausschließlich über die Attribute hergestellt, wobei die Attribute zu diesem Zweck Zeiger auf Objekte enthalten. In gewisser Weise leidet das OODM damit am selben Defizit wie das RDM (nur daß hier Relationen gegen Klassen und Primär- bzw. Fremdschlüssel gegen Objektidentitäten bzw. Zeiger getauscht sind): Ihm fehlt eine fundamentale semantische Abstraktion. Generische Operationen des OODMs sind die Instanziierung von Klassen in Objekte, die Wertzuweisungen an Attribute und das Löschen von Objekten.

3.2 Notationselemente des UML-Klassendiagramms

Ein UML-Klassendiagramm kann die folgenden primären (unabhängigen) Notationselemente enthalten:

1. *Klasse* [OMG09a, §7.3.7],
2. *Assoziation* (wobei zweistellige besonders häufig vorkommen und aus diesem Grund über eine vereinfachte Notation verfügen) [§7.3.3],
3. *Assoziationsklasse* (als Spezialisierung von *Klasse* und *Assoziation*) [§7.3.4],
4. *Generalisierung* [§7.3.20],
5. *Abstraktion* [§7.3.1] (mit Spezialfällen *Realisierung*, *Substitution* [§7.3.45, §7.3.50]),
6. *Abhängigkeit* [§7.3.12] (mit dem Spezialfall *Verwendung* [§7.3.53]) sowie
7. *Schnittstelle* [§7.3.24] (tritt typischerweise in Zusammenhang mit *Interface Realization* [§7.3.25], einem Spezialfall von *Realisierung* und *Verwendung*, auf).

Klasse ist nominal ein Element des OODMs, faktisch aber genausogut eines des ERMs, wo es allerdings *Entitätstyp* heißt. *Assoziation* findet man als Modellierungselement im ERM (wo es *Beziehungstyp* oder engl. *relationship type* heißt), jedoch nicht im OODM (s. o.): hier müssen Beziehungen durch Attribute ausgedrückt werden, was aber zunächst, da ein Attribut immer nur auf ein Objekt verweisen kann, auf zweistellige, gerichtete Zu-1-Beziehungen beschränkt bleibt (mehr dazu unten). *Assoziationsklasse* ist ein Konzept des ERM, dort aber nicht von Beziehungstypen zu unterscheiden (Beziehungstypen können im ERM genau wie Entitätstypen Attribute deklarieren) und kommt im OODM nur dann vor, wenn man als Klassen reifizierte Relationen als eigenständiges Modellierungskonstrukt betrachtet (mehr zur Reifizierung von Relationen in Abschnitt 3.3.4). Instanzen von *Generalisierung*, *Abstraktion* und *Abhängigkeit* definieren im Gegensatz zu *Assoziation* (deren Instanzen Beziehungen zwischen den Elementen der an einer Assoziation beteiligten Klassen deklarieren) Instanzen von mit der Definition der UML vorgegebenen Relationen auf der Menge der Klassen; sie stehen damit aus model-

lierungstechnischer Sicht eine Ebene über den Assoziationen und haben somit keine Beziehung zu relationalen Modellen.[5] Das gleiche gilt im Ergebnis für *Schnittstelle*. Es bleibt also, daß von den Diagrammelementen auf oberster Ebene lediglich *Assoziation* und *Assoziationsklasse* Modellierungskonstrukte repräsentieren, denen es im OODM (im Gegensatz zum ERM als Stellvertreter einer relationalen Sichtweise) an Entsprechung mangelt.

Von den beiden für unsere Belange relevanten primären Diagrammelementen *Assoziation* und *Klasse* (letztere, weil sie Teile der Definition von *Assoziationsklasse* enthält) abhängig sind eine Reihe weiterer definiert, die die primären näher bestimmen oder einschränken. Es sind dies[6]

1. für *Klasse*:
   a) ein *Name* zur eindeutigen Identifizierung,
   b) optional *Attribute*, deren Werte die Zustände der Instanzen der Klassen codieren,
   c) optional *Operationen*, die das Verhalten der Instanzen einer Klasse definieren,
   d) optional die Auszeichnung als *abstrakte Klasse*,
   e) optional die Auszeichnung als *aktive Klasse*;
2. für *Assoziation*:
   a) optional eine *Name*,
   b) optional eine *Leserichtung*,
   c) mindestens zwei *Assoziationsenden*,
   d) optional die Auszeichnung als *abgeleitete Assoziation* sowie
   e) optional eine Menge von vordefinierten *Eigenschaften* (Property Strings) die Eigenschaften der Assoziation ausdrücken (wie beispielsweise *ordered*, *subsets*,…).

Von diesen sind alle Elemente, die sich mit Verhalten befassen (*Operation*, *aktive Klasse* etc.), definitionsgemäß weder Bestandteil des ERMs noch des OODMs und können daher für weitere Betrachtungen entfallen. Von den verbleibenden im Fall *Klasse* entspricht *Name* dem Vorkommen im OODM und im ERM gleichermaßen. *Attribute* kommen zwar ebenfalls in beiden vor, unterscheiden sich aber, was deren Semantik angeht: Während im ERM Attribute stets Wertsemantik haben, können sie im OODM Wert- oder Referenzsemantik haben, also insbesondere auch Verweise auf andere Objekte enthalten. Das UML-Klassendiagramm folgt hier dem OODM. *Abstrakte Klasse* ist Zusammenhang mit *Generalisierung* zu sehen (s.o.) und deswegen hier ohne Bedeutung.

[5] Man beachte, daß in sog. semantischen Datenmodellen die Generalisierung häufig als besondere Relation auf gleicher Ebene mit der Aggregation eingeführt wird; dies ist ein Fehler.

[6] Elemente, die überall vorkommen können und für das Klassendiagramm keine spezifische Bedeutung haben, sind hier weggelassen. Dazu zählen Stereotype, Kommentare und Constraints.

Im Fall der *Assoziation* wird die Sache interessanter: Da es im OODM keine Assoziationen gibt, handelt es sich bei *Name* um die Namen der Relationen aus dem ERM. Das gleiche gilt im Prinzip für *Leserichtung* und *Assoziationsenden*, auch wenn es diese nominal im ERM nicht gibt: *Leserichtung*, wo notwendig, ergibt sich implizit aus Rollennamen (s.u.), *Assoziationsenden* entsprechen den Stellen der Beziehungstypen. *Abgeleitete Assoziation* ist vor allem im Zusammenhang mit Vererbung zu sehen und bringt keinen neuen relationalen Aspekt ein.

Von den obengenannten, von *Klasse* und *Assoziation* abhängigen Notationselementen können wiederum weitere abhängig sein. Es sind dies für

1. *Klasse*

   b) *Attribute*:

      i. ein *Name* zur Identifikation,

      ii. optional eine *Sichtbarkeit*,

      iii. optional ein *Typ*,

      iv. optional eine *Multiplizität*,

      v. optional eine Menge von vordefinierten *Eigenschaften* (Property Strings), die Eigenschaften des Attributs ausdrücken (bspw. *ordered*, *read only*, ...),

      vi. optional die Auszeichnung als *abgeleitetes Attribut*,

      vii. optional die Auszeichnung als *Klassenattribut* (durch Unterstreichung) und

      viii. optional ein *Ausdruck*, der den Default-Wert des Attributs darstellt; und für

2. *Assoziation*

   c) *Assoziationsende*:

      i. optional ein *Rollenname*,

      ii. optional eine *Navigierbarkeit* und, davon abhängig, *Navigationsrichtung*,

      iii. eine optionale *Qualifizierung*,

      iv. optional eine *Multiplizität*,

      v. optional eine Menge von *vordefinierten Eigenschaften* (Property Strings), die Eigenschaften der Assoziation bezogen auf das Assoziationsende ausdrücken (wie beispielsweise *ordered*, *read only*, …),

      vi. bei binären Assoziationen optional eine *Aggregation* oder *Komposition*,

      vii. optional eine *Sichtbarkeit*.

Die auf Attribute von Klassen bezogenen Diagrammelemente 1 b) i–viii können allesamt dem OODM zugerechnet werden (wobei *Name*, *Typ* und *Multiplizität* auch im ERM

vorkommen). Man würde vielleicht erwarten, daß im Gegenzug die auf Assoziationsenden bezogenen Diagrammelemente 2 c) i–vii relationalen Ursprungs sind — tatsächlich sind sie das aber bis auf *Rollenname*, *Multiplizität*, *Qualifizierung* und (wenn man Erweiterungen des ERM hinzunimmt) *Aggregation*/*Komposition* nicht. Es fällt vielmehr auf, daß sie mit *Sichtbarkeit*, *Multiplizität* und den vordefinierten *Eigenschaften* Elemente aufweisen, die denen der Attribute entsprechen. Mit *Navigierbarkeit* und *Navigationsrichtung* werden zudem zwei Elemente eingebracht, die dem ERM und seinen Beziehungstypen fremd sind und die vielmehr einen Zeigercharakter von Beziehungen nahelegen.

3.3 Mögliche Kompensation der relationalen Defizite im OODM

Bei der Analyse des UML-Klassendiagramms haben wir einige Notationselemente identifiziert, deren Ursprung relational ist und die keine direkte Entsprechung im OODM haben. Dabei hatte sich jedoch schon angedeutet, daß die Tatsache, daß sie darin nicht vorkommen nicht heißt, daß sie darin nicht ausgedrückt werden können — genauso, wie das RDM Klassen mit Relationen emuliert, kann vielleicht auch das OODM Relationen mit Bordmitteln umsetzen. Inwieweit dies möglich ist, soll im folgenden geklärt werden.

3.3.1 Mangelnde bidirektionale Navigierbarkeit

Im Gegensatz zu Relationen, die grundsätzlich in alle Richtungen navigierbar sind, sind Zeiger grundsätzlich nur in eine Richtung navigierbar. Insofern ist auch die häufig angetroffene mathematisch Interpretation von Zeigern als Funktionen, die ein Objekt, auf ein anderes abbilden (s. z. B. [BO97]), nicht ganz korrekt: Während eine Funktion zumindest im Prinzip immer umkehrbar ist (auch wenn die Umkehrung nur noch eine Relation und keine Funktion mehr ist), so daß man vom Bild zu seinen Urbildern gelangen kann, gilt das für einen Zeiger nicht — während der Zeiger als Attribut der Quelle dieser fest zugeordnet ist, weiß das Ziel noch nicht einmal, daß auf es gezeigt wird, geschweige denn, welche Quellen dies tun. Allerdings gilt für Zeiger wie für Funktionen, daß sie rechtseindeutig sind, daß also zu jeder Quelle höchstens ein Ziel gehört.

3.3.2 Darstellung von Beziehungen zu Mehreren und ihre Grenzen im OODM

Vernachlässigen wir zunächst die mangelnde bidirektionale Navigierbarkeit von Zeigern bzw. Attributen, die Beziehungen darstellen sollen, dann bleibt immer noch das Problem, daß sich Beziehungen von einer Quelle zu mehreren Zielen grundsätzlich nur über mehrere Zeiger realisieren lassen. Da Zeiger aber (als Attribute) benannt sind, ergibt sich daraus ein praktisches Problem, nämlich daß man schlecht eine variable Anzahl von Namen für eine variable Anzahl von mit einem Objekt in Beziehung stehenden anderen Objekten vorsehen kann (vgl. dazu auch [No97]). Dieses Problem kann man (wie beispielsweise in Smalltalk geschehen [GR83]) mittels sog. indizierter Instanzvariablen

lösen, also mit Attributen, auf deren Inhalt nicht über einen Namen (den Namen des Attributs), sondern über einen Index zugegriffen wird (Abbildung 2). Dieser Index muß keine natürliche Zahl, sondern kann jeder beliebige Wert und sogar ein Objekt sein, so daß sich damit sogar (der Zugriff über) ein qualifiziertes Assoziationsende umsetzen läßt. Da jedes Objekt jedoch nur ein namenloses, indiziertes Attribut haben kann, kann das Objekt auch nur in höchstens einer Zu-*n*-Beziehung (also derselben Beziehung zu mehreren anderen Objekten) stehen. Das jedoch ist unrealistisch, so daß Zu-*n*-Beziehungen in der Regel über eine Art Zwischenobjekte, also Objekte, deren einziger Zweck es ist, mehrere andere Objekte aufzunehmen (z. B. in indizierten Instanzvariablen), umgesetzt werden, wobei dann ein (benanntes) Attribut im Quellobjekt auf ein Zwischenobjekt und das Zwischenobjekt auf die *n* Objekte des Ziels verweist (Abbildung 2). Das aber bedeutet nicht nur eine Indirektion beim Zugriff, sondern führt auch dazu, daß man als Navigator immer wissen muß, ob es sich bei einer durch ein Attribut repräsentierten Beziehung um eine Zu-1- oder eine Zu-*n*-Beziehung handelt, da man im ersten Fall direkt zum in Beziehung stehende andere Objekt gelangt, im zweiten Fall hingegen direkt nur zum (nur zu Hilfszwecken eingeführten) Zwischenobjekt.[7]

Damit aber nicht genug, denn auch die Möglichkeiten zur Umsetzung von Zu-1-Beziehungen sind nicht vollständig befriedigend: Während Beliebig-zu-1-Beziehungen kein Problem sind, ja wegen der Zeigersemantik von Attributen sogar immer angenommen werden müssen (es können beliebig viele Zeiger auf dasselbe Objekt verweisen), lassen sich *n*:1-Beziehungen mit fester oberer Schranke *n* praktisch kaum durchsetzen. Allenfalls 1:1-Beziehungen ließen sich noch erzwingen, indem man Attribute mit Wertsemantik (die also das Objekt selbst und nicht einen Zeiger enthalten) verwendet und man die mangelnde Flexibilität (Objekte eines Subtyps sind in der Regel nicht mehr zuweisbar) sowie eine weitere Anomalie (Zielobjekte von 1:1-Beziehungen haben Wertsemantik) in Kauf nimmt; alles darüber hinaus fällt jedoch in das Gebiet der Alias-Kontrolle und liegt damit außerhalb der Ausdrucksmöglichkeiten heute gängiger objektorientierter Programmiermodelle (und damit auch des OODMs; vgl. Fußnote 2).

3.3.3 Darstellung von Bidirektionalität durch paarige Attribute

Die oben erwähnte mangelnde Umkehrbarkeit von Zeigern läßt sich natürlich durch paarige Zeiger, also Zeiger, die paarweise auftreten und so miteinander gekoppelt sind, daß der eine immer in die Gegenrichtung des anderen zeigt, kompensieren. Für 1:1-Beziehungen ist das direkt möglich, für alle anderen über indizierte Attribute oder per Umweg über Zwischenobjekte. Was dabei jedoch berücksichtigt werden muß, ist die Bedingung, daß zu jedem Zeiger von einem Objekt zu einem anderen auch ein Zeiger vom anderen genau zum Ausgangsobjekt existieren muß — es reicht dazu nicht, im Mo-

[7] In typisierten objektorientierten Programmiersprachen äußert sich diese Anomalie übrigens auch darin, daß die Instanzvariable im ersten Fall mit dem Typ des Ziels der Beziehung deklariert ist, im zweiten aber mit dem Typ des Zwischenobjekts (meistens ein Subtyp von `Collection` o. ä.), dem bestenfalls der Typ des Ziels als Typparameter beigeordnet ist.

dell einen Zeiger von Quelle zu Ziel und einen von Ziel zu Quelle verweisen zu lassen, denn die Ausprägungen könnten dann ja immer noch auf verschiedene Objekte derselben Klassen zeigen. Diese Bedingung muß insbesondere bei Zuweisungen an eine Instanzvariable beachtet werden, da sie hierbei verletzt werden kann (ja bei sequentieller Ausführung temporär immer verletzt wird), aber auch beim Löschen einzelner Objekte.

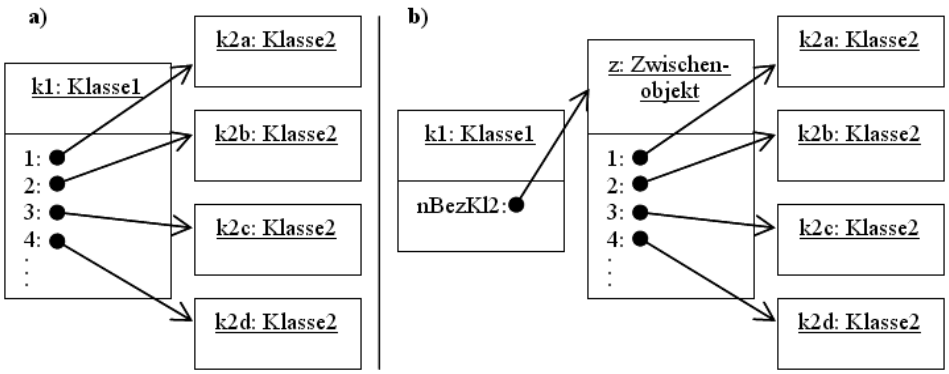

Abb. 2: Unterschiedliche Umsetzungen von Zu-n-Beziehungen im OODM: a) Indizierte Instanzvariablen mit natürlicher Zahl als Index. b) Umsetzung mit Hilfe eines Zwischenobjekts: Das Attribut mit dem Bezeichner nBezKl2 verweist auf das Zwischenobjekt z und ermöglicht dadurch eine 1-n-Beziehung zw. Objekten von Klasse1 und Klasse2.

Mit Hilfe paariger Attribute läßt sich auch auf einfache Weise die Einhaltung der Multiplizitäten beliebiger Beziehungen sicherstellen, da es nun keine Zeiger mehr gibt, von denen ein Objekt nichts weiß (vgl. oben) — es besitzt schließlich selbst Zeiger auf alle Objekte, die auf es verweisen. Diese Zeiger können leicht gezählt und ihre Anzahl damit kontrolliert werden.

3.3.4 Darstellung von höher als zweistelligen Relationen und von Assoziationsklassen

Da Zeiger nur eine Quelle und ein Ziel haben können, sind sie prinzipiell schlecht geeignet, höher als zweistellige Relationen umzusetzen. Man kann hier höchstens mehrere Stellen einer Relation zusammenfassen und gemeinsam (als Tupel) zu Quelle oder Ziel eines Zeigers machen. Die so entstehende geschachtelte Relation ist jedoch semantisch nicht dasselbe wie die Ausgangsrelation, so daß dieser Ansatz hier nicht weiter verfolgt werden soll. Es bleibt dann nur, die Ausgangsrelation zu reifizieren, also in eine Klasse umzuwandeln, die über zweistellige Beziehungen mit den Klassen der Ausgangsrelation verbunden ist (s. z. B. [No97]). Man verliert dadurch die direkte Navigierbarkeit zwischen diesen Klassen, eine Situation, die man allerdings auch schon von der Umsetzung von Zu-*n*-Beziehungen her kennt. Anders als Zwischenobjekte haben reifizierte Relationen aber eine semantische Entsprechung im Modell — sie stehen für ein bestimmtes

Konzept des modellierten Gegenstandsbereichs und es ist in der Tat die Frage, ob die Repräsentation dieses Konzeptes als Klasse weniger adäquat ist als die als Relation. Ein anderes als ein ästhetisches Defizit läßt sich aus dem Fehlen einer direkten Umsetzbarkeit mehrstelliger Relationen daher aus unserer Sicht nicht ableiten.

Ähnliches gilt im wesentlichen für die Umsetzung von Assoziationsklassen, also von Assoziationen, die selbst über Attribute verfügen, die nicht sinnvoll einer der beteiligten Klassen zugewiesen werden können — auch hier bleibt, da Zeiger keine Attribute haben können, nur die Reifizierung als Klassen. Diese ist insbesondere dann auch semantisch sinnvoll, wenn die Assoziationsklassen neben Attributen auch noch Verhalten aufweisen. Beziehungen mit eigenem Verhalten kann man aber konzeptuell auch als Kollaborationen auffassen, die jedoch im relationalen Kontext nicht auftauchen und deswegen hier nicht Gegenstand weiterer Betrachtungen sein sollen.

3.4 Bedeutung der Defizite für die Praxis

Nachdem die relationsbezogenen Defizite des OODM klar sind, stellt sich die Frage, welche Bedeutung diese für die Praxis haben. Rumbaugh berichtet, daß drei- und höherstellige Relationen in der Praxis nur selten vorkommen [Ru87]. Dies deckt sich mit den Erfahrungen der Autoren dieser Arbeit, die das OODM deswegen davon unberührt lassen, zumal die Reifizierung von solchen Relationen als Klassen durchaus eine gangbare Lösung ist. Auch aus dem Vorkommen von Assoziationsklassen in Modellen leiten sie keinen Zwang ab, Relationen als eigenständige Konstrukte in das OODM aufzunehmen, da auch hier Reifizierungen eine sinnvolle Lösung zu sein scheinen.

Was die zweistelligen Beziehungen angeht, ergibt sich in der Praxis ein differenziertes Bild. Die fehlende Kontrolle der Multiplizität *m* von unidirektionalen *m*:*n*-Beziehungen scheint durchaus verschmerzbar, zumal andere feste Obergrenzen als 1 häufig willkürlich erscheinen (selbst das oft bemühte Auto mit seinen vier Rädern ist als Beispiel angreifbar) — nötigenfalls muß hier eine bidirektionale Beziehung eingeführt werden, um die Multiplizität zu kontrollieren. Die Umsetzung von Zu-*n*-Beziehungen über Zwischenobjekte ist sicher lästig, aber im wesentlichen ein Implementierungsdetail, das hinter einer entsprechenden Abstraktion verborgen werden könnte (genauso, wie die Implementierung eines relationalen Datenbanksystems für effizienten Zugriff Indizes verwendet, die aber, abgesehen vielleicht von der Schemadefinition, nirgends explizit auftauchen). Vielmehr muß man anerkennen, daß die Dereferenzierung eines Zeigers, die die Navigation einer Zu-1-Beziehung im OODM bedeutet, sowohl aus programmiersprachlicher als auch aus implementierungstechnischer Sicht so effektiv ist, daß man nicht ohne Not darauf verzichten sollte. Was man sich vielmehr wünschen sollte, ist, daß Zu-*n*-Beziehungen genauso (sprach- und implementations-)effizient wie Zu-1-Beziehungen navigierbar wären. An dieser Stelle erscheint also eine Änderung am OODM durchaus wünschenswert — eine Notwendigkeit der Erweiterung um Relationen läßt sich daraus aber ebenfalls nicht ableiten.

Entsprechendes gilt für die Umsetzung bidirektionaler Beziehungen: Hier die Verantwortung für die Koordinierung paariger Attribute (für die Repräsentation beider Richtungen) in die Hand derjenigen zu geben, die für eine korrekte Umsetzung eines Modells zu sorgen haben, ist der Häufigkeit dieser Aufgabe und ihrer immer wieder gleichen Form nicht angemessen. Statt dessen Relationen einzuführen, denen die Bidirektionalität immanent ist, hätte aber den Preis, auf die einfache und effiziente Dereferenzierung von Zeigern (s. o.) zu verzichten. Alternativ wäre vielmehr zu bedenken, ob nicht die koordinierte Pflege paariger Attribute Bestandteil des OODM werden sollte.

3.5 Notwendige Erweiterung aus Modellierungssicht

Im wesentlichen können wir aus der Betrachtung der relationalen Elemente im UML-Klassendiagramm zwei wünschenswerte Erweiterungen des OODMs ableiten:

1. eine Integration von Zu-*n*-Beziehungen, die den Umweg über explizite Zwischenobjekte (Collections) überflüssig macht und gleichzeitig einen qualifizierten (mit beliebigen Werten indizierten) Zugriff auf die Elemente der Gegenseite erlaubt, und
2. die Einführung von paarigen Attributen.

Zur Umsetzung wären dann auch die Update-Operationen des OODMs zu erweitern bzw. anzupassen: Zur Zuweisung kommen Einfüge- und Entfernoperationen für die Attribute, die Zu-*n*-Beziehungen repräsentieren, hinzu und die Pflege (Zuweisung bzw. Einfügen/ Entfernen) von paarigen Attributen muß so angepaßt werden, daß zu jeder Richtung automatisch auch die Gegenrichtung gepflegt wird.

4 Notwendige Ergänzungen der Objektorientierung aus Datenabfragesicht

Ein weiterer Grund für eine relationale Datenmodellierung ist die Existenz von ausgereiften Datenabfragesprachen. So erlaubt es beispielsweise die weit verbreitete SQL [ISO08], Abfragen höchst unterschiedlicher Komplexitätsgrade gegen relationale Datenbestände deklarativ zu formulieren. Die Prägnanz solcher Abfragen steht im Gegensatz zur Ausführlichkeit imperativer Formulierungen, wie man sie als objektorientiert programmierte Abfragen gleichen Inhalts antrifft.[8] Die Ursache für diese (vermeintliche) Geschwätzigkeit liegt u. a. darin, daß objektorientierte Umsetzungen von Abfragen stets iterativ arbeiten, also im Gegensatz zur Mengenwertigkeit des Relationalen jedes Element (Objekt) einzeln behandeln müssen, und zwar selbst dann, wenn es nur darum geht, über ein Element zum nächsten zu navigieren (das Element selbst also gar nicht

[8] Daß dies längst nicht für alle möglichen Abfragen gilt und daß im übrigen die Formulierung korrekter SQL-Abfragen längst nicht immer einfach ist, soll hier nicht weiter diskutiert werden.

betrachtet wird). Dem kann man entgegenhalten, daß die Mengenbehandlung relationaler Abfragesprachen auch Nachteile hat, z. B. wenn das Ergebnis einer Anfrage genau ein Objekt sein soll oder wenn man mit jedem Objekt einer Ergebnismenge hinterher etwas anderes machen will. Nichtsdestotrotz gibt es Bestrebungen, relationale Anfragesprachen in die objektorientierte Programmierung zu integrieren, und nicht zuletzt wurde mit Microsofts LINQ-Projekt (genauer: mit LINQ to Objects) die Möglichkeit eröffnet, auch Collections (also programminterne, objektorientierte Datenstrukturen) mengenorientiert auszuwerten [To07]. Dies genauer zu betrachten hat sich als ausgesprochen instruktiv erwiesen.

4.1 Auswahlabfragen

Relationale Abfragesprachen wie SQL oder LINQ liefern grundsätzlich eine Ergebnismenge, das sog. Result set. Selbst wenn das Ergebnis der Abfrage genau ein Objekt ist, bleibt es eine Menge, die selbst wieder Gegenstand einer Abfrage sein kann. Ein Vorteil der Mengenbetrachtung ist, daß „kein Ergebnis“ durch eine leere Menge repräsentiert wird und somit in der Regel keiner Sonderbehandlung (wie einer Prüfung auf `not null`) bedarf. Ein weiterer Vorteil ist, daß das Ergebnis als Ganzes mit einem Schritt präsentiert werden kann, z. B. indem es eine neue Datenbanktabelle füllt oder in einer Tabelle auf dem Bildschirm angezeigt wird.

Letzteres ist aber längst nicht immer das Ziel und so entpuppt sich der vermeintliche Vorteil nicht selten als Nachteil: Auf eingebettete relationale Datenbankabfragen folgt häufig prompt eine Iterationen über die Ergebnismenge, bei der jedes Element des Ergebnisses einzeln betrachtet und behandelt wird. So haben praktisch alle Beispiele für LINQ-Abfragen, die keine Aggregatfunktionen oder Joins verwenden, die folgende stereotype Form:

```
var query = from <Laufvariable> in <Collection>
    where <Auswahlkriterium auf Laufvariable>
    select <Ausdruck auf Laufvariable>;
foreach (<Laufvariable> in query) {…}
```

Auf die Mengenwertigkeit des Ergebnisses kann also ohne Verlust verzichtet werden, wenn die Elemente des Ergebnisses mit gleichem Aufwand der Reihe nach innerhalb der For-each-Schleife erzeugt werden können. Genau dies ist aber häufig der Fall, wie die Gegenüberstellung obigen Beispiels mit der objektorientierten Standardformulierung

```
foreach (var <Laufvariable> in <Collection>)
  if (<Auswahlkriterium auf Laufvariable>) {…}
```

zeigt.

4.2 Aggregatfunktionen

Ein weiterer Vorteil mengenwertiger Abfragesprachen ergibt sich aus der Verfügbarkeit von Aggregatfunktionen wie der Bildung der Summe oder der Berechnung des Durchschnitts einer Menge von Werten. Entsprechende Abfragen in LINQ haben in etwa die folgende Form:

```
var query = (from <Laufvariable> in <Collection>
    where <Auswahlkriterium auf Laufvariable>
    select <Ausdruck auf Laufvariable>)
    .<Aggregatfunktion>([<Ausdruck auf Laufvariable>]);
```

Tatsächlich spart man sich hier gegenüber der iterativen Form

```
<Akkumulator initialisieren>
foreach (<Laufvariable> in <Collection>)
  if (<Auswahlkriterium auf Laufvariable>) {<akkumulieren>}
```

das Führen der Akkumulatoren, die zur Berechnung des Aggregatwerts notwendig sind. Allerdings ist die Zahl der möglichen Aggregatbildungen fest vorgegeben und man muß schon die Bibliothek erweitern, wenn man eine neue benötigt. Im Vergleich zu den sog. Folds, wie sie beispielsweise mit der Collection-Methode `inject:into:` von Smalltalk zur Verfügung stehen (und wie es sie auch in LINQ gibt), mit denen man beliebige Aggregatfunktionen realisieren kann, erscheinen die Aggregatfunktionen relationaler Abfragesprachen geradezu altbacken starr (und liefern sicher keinen Anhaltspunkt, das OODM über das, was sowieso schon sinnvoll erscheint, hinaus zu erweitern).

4.2.1 Joins

Vielleicht der größte Vorteil relationaler Abfragesprachen ist die Möglichkeit der Bildung von Verknüpfungen von Relationen durch sog. Joins. Anhand des OODMs

```
class A { Collection<B> b; … }
class B { Collection<C> c; … }
class C { D d; … }
```

läßt sich in LINQ beispielsweise die Abfrage

```
foreach (D dl in (
    from tmp1 in a1.b from tmp2 in tmp1.c select tmp2.d)
  ) { … dl … }
```

formulieren, die die Menge der Attributwerte `d` von Objekten `c` vom Typ `C` gewinnt, die zu einem Objekt `a1` vom Typ `A` indirekt über Objekte `b` vom Typ `B` (über die Verknüpfung der Beziehung von `A` zu `B` und von `B` zu `C`) in Beziehung stehen. Man beachte, daß sich diese Verknüpfung objektorientiert nur im Fall von Zu-1-Beziehungen zwischen `A` und `B` sowie `B` und `C` durch die Verkettung von Attribut-Dereferenzierungen wie in

```
a1.b.c.d
```

herstellen läßt. (wobei dies auch noch ignoriert, daß `a1.b` oder `b.c` auch `null` sein könnten). Andernfalls sind geschachtelte Iterationen wie in

```
foreach (B b in a1.b)
  foreach (C c in b.c)
    … c.d …
```

notwendig, die doch deutlich imperativen Charakter haben.

Wie man an diesem Beispiel allerdings deutlich sieht, sind für Joins auf objektorientierten Datenstrukturen gar keine Relationen als eigenständige Konstrukte notwendig — das Vorhandensein eines Attributs, das eine Collection enthält (die Standardrepräsentation von Zu-*n*-Beziehungen im OODM) reicht vollkommen aus. Jede als einfaches Attribut repräsentierte Zu-1-Beziehung in einer Kette von Joins wurde allerdings insofern zu einer Anomalie führen, als dann die dazugehörige From-Klausel durch eine Attributdereferenzierung ersetzt werden müßte. Dies ist insbesondere dann schlecht, wenn sich die Kardinalität im Laufe der Entwicklung ändert.

Die Schlußfolgerung aus diesem Umstand darf aber nicht sein, daß man Relationen (als eigenständiges Konstrukt) zur Vereinheitlichung von Zu-1- und Zu-n-Beziehungen einführen muß, sondern vielmehr, daß man Zu-1- und Zu-*n*-Beziehungen als Attribute grundsätzlich gleich behandeln sollte (was sich ja andeutungsweise oben schon als Forderung ergab, wenn auch durch die durch Zwischenobjekte verursachte Anomalie motiviert).

4.2.2 Navigationsrichtungen von Abfragen

Mit jeder Abfrage ist für jede Beziehung, die sie ausnutzt, eine Navigationsrichtung fest vorgegeben. Eine Forderung nach bidirektionalen Beziehungen ergibt sich von Abfrageseite nur dann, wenn verschiedene Abfragen verschiedene Richtungen derselben Beziehung benötigen.

4.3 Notwendige Erweiterung aus Datenabfragesicht

Während sich aus Modellierungssicht vielleicht noch umfangreichere Anforderungen an Erweiterungen des OODMs ableiten lassen, bleibt aus Abfragesicht im wesentlichen übrig, Zu-1- und Zu-*n*-Beziehungen gleichartig zu repräsentieren. Nimmt man die aus der Navigation motivierte Forderung ernst, bedeutet es in der Konsequenz, daß für beide Arten von Attributen, denen, die Zu-1-Beziehungen umsetzen und denen, die Zu-*n*-Beziehungen umsetzen, auch dieselben Update-Operationen (Zuweisung, Einfügen, Entfernen) zur Verfügung stehen sollten, so daß der Unterschied nur noch in ihrer Deklaration sichtbar wird. Bidirektionalität als Forderung ergibt sich nur dann, wenn sie auch aus Modellierungssicht notwendig ist (sofern man in Anfragen nur Zusammenhänge

ausnutzen können soll, die auch bei der Modellierung der auszuwertenden Daten vorgesehen waren).

5 Eine Sanfte „Relationalisierung“ der Objektorientierung

Unsere Betrachtungen münden im wesentlichen in der Forderung, die Attribute des OODMs in zweierlei Hinsicht zu erweitern (vgl. [SS10] für eine genauere Erläuterung und Beschreibung einer möglichen Umsetzung in Form einer Programmbibliothek):

1. Attribute, die Relationen repräsentieren, enthalten grundsätzlich *eine Menge* von Zeigern, die über einen Schlüssel indiziert (qualifiziert) sein kann (mit nur einem Zeiger als Spezialfall). Die Operation „Zuweisung“ wird so umdefiniert, daß stets eine Menge von Zeigern zugewiesen wird, und durch Operationen zum Hinzufügen und Entfernen von Zeigern ergänzt. Die Dereferenzierung von Attributen, die Relationen repräsentieren, erfolgt ausschließlich in Abfragen (analog zu denen LINQs).
2. Attribute, die bidirektionale Relationen repräsentieren, werden immer als Paare geführt. Die Paare werden dazu in einer separaten Deklaration, einer Relationsdeklaration, angegeben. Genau wie die Generalisierung und andere Beziehungen höherer Ebene (vgl. Abschnitt 3.2) wird die Relationsdeklaration nicht zur Laufzeit instanziiert; die Tupelmengen, die die Extension einer Relation darstellen, werden durch die Inhalte der entsprechenden Attribute der beteiligten Objekte repräsentiert.

Die auf relationale Aspekte zurückzuführenden Unterschiede zwischen dem UML-Klassendiagramm und dem OODM reduzieren sich damit im wesentlichen auf das Fehlen von höher als zweistelligen Relationen und von attributierten Relationen im OODM, die jedoch vergleichsweise selten sind und durch Klassen ersetzt werden können.

6 Zusammenfassung und Schluß

Das Fehlen von Relationen im OODM ist immer wieder bemängelt worden. Anstatt wie in der modellgetriebenen Softwareentwicklung die Relationen eines Modells in immer gleichen Code zu transformieren, der die Bedeutung der Relationen in den Primitiven objektorientierter Programme ausdrückt, schlagen wir vor, mit dem OODM der relationalen Modellierung entgegenzukommen, indem wir es an einigen Stellen behutsam erweitern. Die aus unserer Sicht sinnvollen Erweiterungen des OODMs haben wir aus der Betrachtung des UML-Klassendiagramms und seinem systematischen Vergleich mit dem OODM sowie einer Untersuchung der Ausdrücke einer relationalen Abfragesprache abgeleitet; die vorgeschlagene Erweiterung haben wir in einer parallelen Arbeit ([SS10]) konkretisiert.

Literaturverzeichnis

[ABS04] Amelunxen, C.; Bichler, L.; Schürr, A.: Codegenerierung für Assoziationen in MOF 2.0.Modellierung 2004. LNCS, vol. P-45. Springer-Verlag, 2004, S. 149-168.

[AHM07] Akehurst, D.; Howells,G.; McDonald-Maier, K.: Implementing associations: Uml 2.0 to java 5. Software and Systems Modeling, vol. 6(1), Springer, 2007, S. 3-35.

[BO97] Bock, C.; Odell, J.J.: A More Complete Model of Relations and Their Implementation – Part II: Mappings. In Journal Of Object-Oriented Programming, vol 10(6), 1997.

[BW05] Bierman, G.; Wren, A.: First-Class Relationships in an Object-Oriented Language. In Proceedings of ECOOP 2005. LNCS, vol. 3586. Springer, 2005, S. 25-29.

[Ch76] Chen, P.P.: The entity-relationship model – toward a unified view of data. In ACM Trans. Database Syst. 1, 1 (Mar. 1976), 1976, S. 9-36.

[Co70] Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM 13, 6, 1970, S. 377-387.

[Ge09] Gessenharter, D: Implementing UML associations in Java: a slim code pattern for a complex modeling concept. RAOOL '09. ACM Press., 2009, S. 17-24.

[GM96] Gawecki, A; Matthes, F.: Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. 10th ECOOP. LNCS, vol. 1098. Springer, 1996, 26-47.

[GR83] Golberg, A; Robson, D.: Smalltalk-80: The Language and its Implementation. Addison Wesley, Reading, Massachusetts, 1983.

[GRL03] Génova, G.; Llorens, J.; Ruiz del Castillo, C.: Mapping UML Associations into Java Code. In Journal of Object Technology, vol. 2(5), 2003, S. 135-162.

[ISO08] ISO/IEC 9075-2:2008. Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation), 2008.

[MM95] Stonebraker, M.; Moore, D.: Object Relational Dbmss: the Next Great Wave. Morgan Kaufmann Publishers Inc., 1995.

[MS92] Matthes, F.; Schmidt, J.W.: Definition of the Tycoon Language TL - a preliminary report. Informatik Fachbericht FBI-HH-B-160/92, Universität Hamburg, 1992.

[MS93] Matthes, F.; Schmidt, J.W.: System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. Euro-Arch'93 Congress. Springer, 1993, S. 301-317.

[No97] Noble, J.: Basic relationship patterns. In Proceedings of EuroPLOP, 1997.

[NPN08] Nelson, S.; Noble, J.; Pearce, D.J.: Implementing first-class relationships in java. In Proceedings of RAOOL 2008. ACM Press, 2008.

[OMG09a] Object Management Group (OMG): OMG Unified Modeling LanguageTM (OMG UML), Superstructure, Version 2.2 (OMG Document formal/2009-02-02), 2009.

[OMG09b] Object Management Group (OMG): OMG Unified Modeling LanguageTM (OMG UML), Infrastructure, Version 2.2 (OMG Document formal/2009-02-04), 2009.

[Øs07] Østerbye, K.: Design of a class library for association relationships. In Proceedings of LCSD '07. ACM Press, 2007, S. 67-75.

[Ru87] Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In Proceedings of OOPSLA '87. ACM Press, 1987, S. 466-481.

[Sc77] Schmidt, J.W.: Some high level language constructs for data of type relation. In ACM Transactions on Database Systems, vol.2, No.3, 1977, S. 247-261.

[SM92] Schmidt, J.W.; Matthes, F.: The database programming language DBPL – Rationale and Report. Technical Report FIDE/92/46, FB Informatik, Universität Hamburg, 1992.

[SS10] Stadtler, D.; Steimann, F.: Objektrelationale Programmierung. In SE 2010 (im Druck).

[To07] Torgersen, M.: Querying in C#: how language integrated query (LINQ) works. In OOPSLA '07. ACM Press, 2007, S. 852-853.