

# Objektrelationale Programmierung

Dilek Stadtler

Lehrgebiet Programmiersysteme  
Fakultät für Mathematik und Informatik  
Fernuniversität in Hagen  
Dilek.Stadtler@fernuni-hagen.de

Friedrich Steimann

Lehrgebiet Programmiersysteme  
Fakultät für Mathematik und Informatik  
Fernuniversität in Hagen  
steimann@acm.org

**Abstract:** Bislang gelten vor allem objektrelationale Datenbanken als Antwort auf den sog. Impedance mismatch zwischen den Welten der relationalen Datenhaltung und der objektorientierten Programmierung. Angesichts jüngster Bestrebungen, im Gegenzug relationale Elemente in die objektorientierte Programmierung einzubringen (wie etwa mit Microsofts LINQ-Projekt), zeigen wir auf, wie das inhärent Zeiger dereferenzierende Modell der objektorientierten Programmierung erweitert werden kann, so daß sich auch relationale Teile eines Datenmodells direkt, d. h. ohne Ergänzung umfangreichen stereotypen Codes, in objektorientierte Programme umsetzen lassen.

## 1 Einleitung

Annähernd parallel zum Aufkommen der objektorientierten Programmierung fanden relationale Datenbanksysteme breiten Einzug in die kommerzielle Praxis. Da beide Entwicklungen für sich genommen ausgesprochen erfolgreich waren, findet man heute häufig Konstellationen vor, in denen objektorientierte Programme auf relationalen Datenbeständen operieren müssen. Unglücklicherweise unterscheiden sich die beiden Paradigmen gleich in mehreren Eigenschaften so sehr, daß an der Schnittstelle zwischen Datenhaltung und Programmierung ein beträchtlicher Übergangswiderstand<sup>1</sup> auftritt. Diesen gilt es zu beseitigen.

Nachdem dazu zunächst mit vergleichsweise wenig Erfolg versucht wurde, relationale Datenbanken durch objektorientierte abzulösen, findet man heute vielerorts sog. objektrelationale Datenbanksysteme vor, an die sich objektorientierte Programme (meistens mittels eines sog. Persistenzlayers) anbinden können. Die Annäherung der Paradigmen erfolgt dabei recht einseitig von Seiten der Datenbanken, die die Abbildung objektorientierter (d. h. im wesentlichen zeigerbasierter, dynamischer) Datenstrukturen in relationale Datenbanktabellen übernehmen. Grund hierfür scheint zu sein, daß die schöne neue Welt der Objektorientierung von den Niederungen der relationalen Datenhaltung möglichst ferngehalten werden soll, und tatsächlich werden Programmierer so von der Last befreit, die Modelle einer vielfältigen Anwendungsrealität auf die vergleichsweise starren Tabellen einer relationalen Datenbank herunterbrechen zu müssen. Auf der Strecke geblieben ist dabei jedoch die Relation als fundamentale konzeptuelle Abstraktion.

---

<sup>1</sup> In der englischsprachigen Literatur spricht man von einem „impedance mismatch“ [CM84].

In jüngerer Zeit ist eine Reihe von Arbeiten entstanden, die den umgekehrten Weg gehen und versuchen, die Vorteile einer relationalen Sicht auf Daten in die objektorientierte Programmierung einzubringen, indem sie objektorientierte Programmiersprachen um relationale Konstrukte erweitern [BW05, NPN08, Øs07, Ru87]. Die meisten dieser Arbeiten führen dazu Relationen als neben Klassen gleichberechtigte Sprachkonstrukte ein, deren Instanzen die herkömmliche Verzeigerung der Objekte über Instanzvariablen und Collections ersetzen sollen. Eine der sichtbarsten Arbeiten auf diesem Gebiet ist jedoch ausgerechnet Microsofts LINQ-Projekt, das gerade nicht die Möglichkeiten der Datenstrukturierung um Relationen ergänzt, sondern vielmehr eine (an SQL angelehnte) relationale Abfragesprache für die herkömmlichen, collection-basierten Datenstrukturen einführt [BMT07]. Die Datenstrukturen, die einem objektorientierten Programm zugrunde liegen, werden dadurch jedoch nicht relationaler.

Mit dieser Arbeit wollen wir beide Stoßrichtungen vereinen, indem wir eine sanfte Erweiterung des objektorientierten Datenmodells vorstellen, die seinen navigierenden (d. h. im wesentlichen Zeiger dereferenzierenden) Charakter erhält, also insbesondere ohne die Einführung von Relationen als separat zu verwaltenden Tupelmengen auskommt. Als Ansatzpunkte hierfür haben wir in einer parallelen Arbeit [SS09] die implementationsbedingte Unterscheidung zwischen Zu-1-Beziehungen (direkt über Zeiger realisiert) und Zu- $n$ -Beziehungen (per Umweg über Collections realisiert) sowie die mangelnde Bidirektionalität von Beziehungen, die bislang durch paarige Beziehungen kompensiert werden muß (in der Natur der Zeiger begründet), ausgemacht. Indem wir uni- und bidirektionale Zu-1- und Zu- $n$ -Beziehungen so vereinheitlichen, daß sie sich syntaktisch nur noch bei ihrer Deklaration unterscheiden, wollen wir den Weg für eine Programmierung bereiten, die wir (in Anlehnung an die objektrelationalen Datenbanken, aber dazu im Ansatz eher komplementär) *objektrelational* nennen.

Der Rest der Arbeit gliedert sich wie folgt: In Abschnitt 2 stellen wir das Problem aus unserer Sicht vor und diskutieren kurz die Arbeiten, die sich damit bereits befaßt haben. In den Abschnitten 3 und 4 stellen wir dann unsere Lösungen für die beiden obengenannten Einzelprobleme vor und zeigen, wie sie sich in die konventionelle objektorientierte Programmierung sowie in Abfragen mit LINQ eingliedern. In Abschnitt 5 skizzieren wir noch kurz die Implementierung (mittels Bibliotheken) in C#, bevor wir in Abschnitt 6 zusammenfassen und schließen.

## 2 Probleme

### 2.1 Das Problem der Unidirektionalität

Die Unidirektionalität, also die Tatsache, daß alle Beziehungen gerichtet und nur in diese Richtung navigierbar sind, ist immer dann ein Problem, wenn ein Sachverhalt Navigation in beide Richtungen erfordert. Hier zwei Beziehungen koordiniert pflegen zu müssen stellt gegenüber dem Relationenmodell, in dem man es von Haus aus immer nur mit *einer* Beziehung zu tun hat, die in beliebige Richtungen navigierbar ist, einen erheblichen Nachteil dar. Ein Beispiel soll dies verdeutlichen.

```

class Firma {
    ICollection<Angestellter> angestellte = new List<Angestellter>();
    ICollection<Arbeitsplatz> arbeitsplaetze = new List<Arbeitsplatz>();
    ICollection<Arbeitsplatz> freieArbeitsplaetzeMitTelefon() {
        ICollection<Arbeitsplatz> ergebnis = new List<Arbeitsplatz>();
        foreach (Arbeitsplatz aplz in arbeitsplaetze)
            if (aplz.mitTelefon) ergebnis.Add(aplz);
        foreach (Angestellter agst in angestellte)
            if (agst.arbeitsplatz != null) ergebnis.Remove(agst.arbeitsplatz);
        return ergebnis;
    }
}
class Angestellter { Arbeitsplatz arbeitsplatz; }
class Arbeitsplatz { bool mitTelefon; }

```

**Abbildung 1:** Beispielhafte unidirektionale Zu-1- und Zu-*n*-Beziehungen zwischen den Klassen Firma, Angestellter und Arbeitsplatz sowie eine darauf basierende Auswertung freieArbeitsplaetzeMitTelefon(), die die Umkehrung einer Beziehung verlangt (in C#).

Eine Firma verfüge über eine Reihe von Angestellten und eine Menge von Arbeitsplätzen, von denen manche mit einem Telefon ausgestattet sind. Einem Angestellten sei ein Arbeitsplatz zugeordnet — die umgekehrte Zuordnung sei jedoch nicht modelliert. Sie läßt sich aber aus den bestehenden Beziehungen ableiten, wie das Beispiel in Abbildung 1 zeigt. Die indirekte Form der Ableitung wird daran deutlich, daß die zweite Schleife in der Methode freieArbeitsplaetzeMitTelefon() über die Angestellten und nicht über die Arbeitsplätze iteriert. Während dies im gegebenen Beispiel vielleicht kein großes Problem ist, so findet man in der Realität doch leicht Varianten, die nicht nur umständlich zu programmieren, sondern auch noch ausgesprochen ineffizient in der Ausführung sind.

Eine mögliche Lösung ist, die umgekehrte Richtung der Beziehung ebenfalls explizit vorzusehen. Dazu wäre es dann notwendig, eine Instanzvariable angestellter vom Typ Angestellter in der Klasse Arbeitsplatz einzuführen. Die Schleife könnte dann über die Arbeitsplätze iterieren und in einem Schritt die unbesetzten mit Telefon herausfiltern. Das Problem an diesem Ansatz ist allerdings, daß die Anpassung der Inhalte der Instanzvariablen arbeitsplatz (in Angestellter) bzw. angestellter (in Arbeitsplatz) bei einer Zuweisung an eine der beiden für die andere ebenfalls explizit (per Zuweisung) erfolgen muß. Dies ist nicht nur lästig, sondern auch noch fehleranfällig.

## 2.2 Das Problem der Unterscheidung von Zu-1- und Zu-*n*-Beziehungen

Durch die Verweise Semantik von Variablen in der objektorientierten Programmierung stellt jede belegte Instanzvariable eine gerichtete Beziehung zu genau einem anderen Objekt her. Diese Beziehung zu navigieren entspricht der Dereferenzierung des Verweises (Zeigers) und ist damit extrem effizient umgesetzt. Wird der Variable ein neues Objekt zugewiesen, wird damit die Beziehung zum ursprünglichen Objekt durch die zum neuen ersetzt.

Da eine Variable nicht auf mehrere Objekte gleichzeitig verweisen kann, ist für die Umsetzung von Zu-*n*-Beziehungen der Umweg über Zwischenobjekte notwendig. Zwischenobjekte sind in der Regel (indirekte) Instanzen einer gemeinsamen Superklasse Collection o. ä., deren Zweck es ist, mehrere Objekte in einem, dem Collection-Objekt,

zu sammeln und über dieses Collection-Objekt einzeln oder der Reihe nach zugänglich zu machen. Eine Instanzvariable, die ein solches Zwischenobjekt zum Wert hat, verweist damit aber nicht auf die im Zwischenobjekt enthaltenen Objekte, sondern auf das Zwischenobjekt. Eine Zuweisung an diese Variable ersetzt somit auch nicht (wie im Zu-1-Fall) das bezogene Objekt (selbst dann nicht, wenn es tatsächlich nur eines ist), sondern das Zwischenobjekt. Soll sich die Menge der bezogenen Objekte ändern, ist dazu das Zwischenobjekt zu manipulieren.

Dieser fundamentale Unterschied in der Umsetzung von Zu-1 und Zu- $n$ -Beziehungen ist zwar implementierungstechnisch nachvollziehbar, aber logisch nicht gerechtfertigt. Schnell kann sich aus der Anwendungsdomäne heraus ergeben, daß eine Zu-1-Beziehung durch eine Zu-2- oder Zu- $x$ -Beziehung ersetzt werden muß, und es ist nicht nachvollziehbar, warum diese Änderung in der Kardinalität zu weitreichenden Codeänderungen führen sollte. So wird man vielleicht im obigen Beispiel von Arbeitsplätzen und Angestellten zunächst davon ausgehen, daß jedem Arbeitsplatz genau ein Angestellter zugeordnet ist. Dies geht solange gut, bis der erste Angestellte nur noch halbtags arbeiten will und sein Arbeitsplatz für die andere Hälfte des Tages nicht ungenutzt bleiben soll. Dann wäre es schön, wenn sich aus Sicht des Programmierers Zu-1- und Zu- $n$ -Beziehungen nur in der Angabe einer Kardinalität bei der Deklaration der Variable unterscheiden würden. Im Beispiel von Arbeitsplätzen und Angestellten würde das bedeuten, daß die Beziehung von Arbeitsplätzen zu Angestellten (eine Zu-2-Beziehung) nicht grundsätzlich anders umgesetzt ist als die umgekehrte Beziehung von Angestellten zu Arbeitsplätzen (eine Zu-1-Beziehung). Wie wir noch sehen werden, würde damit auch das überaus lästige Problem der Null-Zeiger-Dereferenzierung, das immer noch für viele Programmabbrüche verantwortlich ist, auf elegante Art und Weise umgangen.

### 2.3 Verwandte Arbeiten

Eine ganze Reihe von Arbeiten (so z. B. [BGE07, BW05, NNP, Øs07, Ru87]), von denen wir hier aus Platzgründen nur einige exemplarisch besprechen können, befaßt sich mit der Erweiterung der objektorientierten Programmierung um Relationen. Als eine der ersten muß die von Rumbaugh aus dem Jahr 1987 genannt werden, in der, von der objektorientierten Modellierung herkommend, die Unverzichtbarkeit von Relationen für die semantische Nachbildung der Realität in Programmen besonders hervorgehoben wird [Ru87]. Zwar führt Rumbaugh Relationen als eine spezielle Art von Klassen ein, die instanziiert werden können (und deren Instanzen dann die Extension der Relation enthalten), jedoch erkennt auch er schon die Wichtigkeit der Navigation von Objekt zu Objekt an und schlägt deswegen vor, automatisch Methoden für die an einer Relation beteiligten Klassen zu generieren, die die Beziehung von einer Instanz der beteiligten Klassen zu denen der anderen Seite der Relation herstellen. Allerdings stützt sich die Implementierung dieser Methoden auf die Tupelmengen der Relationen, die separat (von den Objekten unabhängig) verwaltet werden.

An die Arbeit von Rumbaugh anknüpfend stellen Bierman und Wren ihre formal spezialisierte Sprache RelJ vor, die Relationen als Typen modelliert [BW05]. Anders als bei Rumbaugh sind Instanzen dieser Typen Tupel (und keine Tupelmengen), die von einer Relation unabhängig existieren können. RelJ sieht auch die Vererbung unter Relationen

vor, jedoch mit einer eher fragwürdigen Semantik, auf die hier nicht weiter eingegangen werden kann. Daß die Relationen von RelJ gerichtet sind (und entsprechend nur in eine Richtung navigiert werden können), stellt zudem eine erhebliche Beschränkung dar.

Der Richtungscharakter der Relationen von RelJ bleibt in den Assoziationen von Østerbye nominal erhalten, wird jedoch durch unbeschränkte Zugriffsmöglichkeiten auf die (mit To und From gekennzeichneten) Assoziationsenden (und die daraus resultierende freie Navigierbarkeit) faktisch aufgehoben [Øs07]. Anders als [Ru87, BW05] schlägt Østerbye hierfür keine Spracherweiterung vor, sondern setzt auf eine Implementierung mittels Bibliotheken. Voll erhalten bleibt bei Østerbye die Unterscheidung von Zu-1- und Zu- $n$ -Beziehungen, obwohl auch für erstere Assoziationen vorgesehen sind.

Andere Arbeiten haben sich nicht zum Ziel gemacht, die objektorientierte Programmierung um Relationen zu erweitern, sondern befassen sich damit, wie (in der Regel aus Modellen stammende) Relationen in herkömmlichen objektorientierten Code abgebildet werden können. Amelunxen et al. [ABS04] beispielsweise haben dazu untersucht, wie sich die neuen assoziationsabhängigen Konzepte Union, Redefinition und Subset aus dem MOF-2.0-Standard [Omg06] in Code umsetzen lassen. Die Ergebnisse können als Basis eines Codegenerators herangezogen werden, wobei sich (wie üblich) die Frage nach der Umkehrbarkeit der Abbildungen (in der Realität der modellgetriebenen Softwareentwicklung nach wie vor wichtig) stellt. Entsprechend stellt Gessenharter Implementierungsmuster zur Umsetzung von UML-Assoziationen vor, wobei hier der Fokus u. a. auf der freien Kombination von Navigierbarkeit und Sichtbarkeit von Assoziationsenden liegt (die laut Gessenharter mit einem Bibliotheksansatz nicht umgesetzt werden kann) [Ge09].

Weder das eine noch das andere verfolgt Microsoft mit seinem LINQ-Projekt: Hier wird vielmehr eine gemeinsame Schnittstelle für verschiedene Arten von Datenquellen definiert, die es erlaubt, SQL-ähnliche, relationale Anfragen unabhängig von der Art einer Quelle zu formulieren [BMT07]. Der vermutlich interessanteste Beitrag ist hierbei die Vereinheitlichung des Zugriffs auf programminterne Collections und -externe Quellen wie relationale und XML-Datenbanken — die Abfragesprache selbst ist weniger revolutionär, insbesondere wenn man sich vor Augen hält, daß sie sich im wesentlichen auf Methoden, die  $\lambda$ -Ausdrücke als Parameter akzeptieren, zurückführen läßt (und damit nicht über das hinausgeht, was funktionale Sprachen und auch Smalltalk immer schon boten).

### **3 Beseitigung des Unterschieds von Zu-1- und Zu- $n$ -Beziehungen**

Für die Umsetzung von Zu- $n$ -Beziehungen haben Collections in der objektorientierten Programmierung eine zentrale Bedeutung. Gegenüber den immer gleich gearteten Relationen des Relationenmodells besitzen sie den Vorteil, daß man ihnen, da sie als ganz normale Klassen implementiert sind, beliebiges Verhalten beordnen kann. So lassen sich leicht geordnete oder gar sortierte Beziehungen definieren (also Beziehungen, in denen die Elemente auf der  $n$ -Seite eine feste Reihenfolge haben oder sortiert sind; in Smalltalk beispielsweise Ordered oder Sorted Collections), es lassen sich Elemente

durch einen Schlüssel gezielt auffinden (Arrays oder Dictionaries), es läßt sich festlegen, ob ein Objekt einfach oder mehrfach in einer Beziehung zum Ausgangsobjekt stehen kann (Sets oder Bags) und so weiter. Dieser Vorteil sollte nicht aufgegeben werden.

Es bleibt also nur, wenn man die grundlegende Unterscheidung von Zu-1- und Zu-*n*-Beziehungen beseitigen will, das programmiersprachliche Konstrukt für Zu-1-Beziehungen anzugleichen. Da Zu-1-Beziehungen ein Spezialfall von Zu-*n*-Beziehungen sind, ist dies kein theoretisches Problem — ein praktisches hingegen schon, zumindest wenn man für eine gewisse Akzeptanz unter Programmierern sorgen will, denen es ja heute schon freisteht, Zu-1-Beziehungen mittels (einelementiger) Collections umzusetzen, die dies aber schon aufgrund des zusätzlichen Programmieraufwands wohl kaum freiwillig tun würden. Es geht hier also zunächst um die geschickte Wahl einer einheitlichen Syntax.

### 3.1 Definition eines abstrakten Datentypen für die einheitliche Behandlung von Zu-1- und Zu-*n*-Beziehungen

Abbildung 2 stellt eine solche Syntax in Form der Definition eines abstrakten Datentypen (ADT) *Association* vor. Die Definition von *Association* stützt sich auf ein paar andere, als gegeben vorausgesetzte Datentypen: Der Datentyp *Boolean* ist Standard, *Card* entspricht den natürlichen Zahlen ergänzt um ein Element „∞“ für „beliebig“, das größer ist als jede Zahl aus *Card*, *Object* ist der Datentyp beliebiger Objekte (die natürlich selbst typisiert sind; die Typisierung lassen wir hier aber unberücksichtigt) und *Collection* der von beliebigen Collections (wir setzen hier lediglich die Existenz der Operationen *add* und *card* mit üblicher Syntax und Semantik voraus). „null“ ist ein Wert vom Typ *Object* mit der üblichen Bedeutung. „⊥“ benennt einen Funktionsausdruck, dessen Ergebnis undefiniert ist und der somit zu einem Fehler führt. Wie üblich vererben sich Fehler in dem Sinne, daß ein Funktionsausdruck, der einen undefinierten Funktionsausdruck enthält, ebenfalls undefiniert ist.

Instanzen von *Association* haben eine maximale Kardinalität *bound*, die bei ihrer Erzeugung (mittels *new*) angegeben werden muß und die sich unter keiner Operation ändert. Eine Überschreitung der maximalen Kardinalität macht einen Funktionsausdruck undefiniert; minimale Kardinalitäten (einschließlich einer Unterschranke 1, entsprechend not null) und eine Einhaltung derselben sind nicht vorgesehen, sollten aber problemlos hinzugefügt werden können. Während sich Zu-1-Beziehungen von Beziehungen beliebiger Kardinalität in der Definition des ADT nicht unterscheiden, ist es natürlich sinnvoll, für Zu-1-Beziehungen spezielle Implementierungen vorzusehen, die die Beschränkungen auf ein Element ausnutzen; deren Auswahl kann aber dem Compiler überlassen werden.

Die Funktionen *add*, *remove* und *replace* können in konkreter (Programmiersprachen-) Syntax durch die Infix-Operatoren *+=*, *-=* bzw. *:=* ersetzt werden. Sie sind in der zweiten Operandenstelle überladen; der Unterstrich an der Stelle steht für einen beliebigen Operanden eines der für die Stelle zulässigen Typen (*Object* oder *Collection*). *replace* ersetzt die Zuweisung (doch Achtung: Durch *a := null* wird nur der Inhalt der Assoziation *a* ersetzt und nicht *a* selbst!) und ist im Falle von Zu-1-Beziehungen vermutlich die am häufigsten verwendete Operation.

**abstract data type** *Association***imports** *Boolean, Card, Object, Collection***syntax**

|             |   |   |
|-------------|---|---|
| new:        | <i>Card</i>                                   | $\rightarrow$ <i>Association</i>              |
| bound:      | <i>Association</i>                            | $\rightarrow$ <i>Card</i>                     |
| card:       | <i>Association</i>                            | $\rightarrow$ <i>Card</i>                     |
| add:        | <i>Association</i> $\times$ <i>Object</i>     | $\rightarrow$ <i>Association</i> $\vee \perp$ |
| add:        | <i>Association</i> $\times$ <i>Collection</i> | $\rightarrow$ <i>Association</i> $\vee \perp$ |
| remove:     | <i>Association</i> $\times$ <i>Object</i>     | $\rightarrow$ <i>Association</i>              |
| remove:     | <i>Association</i> $\times$ <i>Collection</i> | $\rightarrow$ <i>Association</i>              |
| replace:    | <i>Association</i> $\times$ <i>Object</i>     | $\rightarrow$ <i>Association</i> $\vee \perp$ |
| replace:    | <i>Association</i> $\times$ <i>Collection</i> | $\rightarrow$ <i>Association</i> $\vee \perp$ |
| contains:   | <i>Association</i> $\times$ <i>Object</i>     | $\rightarrow$ <i>Boolean</i>                  |
| collection: | <i>Association</i>                            | $\rightarrow$ <i>Collection</i>               |
| object:     | <i>Association</i>                            | $\rightarrow$ <i>Object</i> $\vee \perp$      |

**semantics** $\forall n \in \text{Card}, a \in \text{Association}, o, o' \in \text{Object}, o \neq \text{null}, c \in \text{Collection}:$ 

$\text{bound}(\text{new}(n)) = n$   
 $\text{card}(\text{new}(n)) = 0$   
 $\text{bound}(\text{add}(a, \_)) = \text{bound}(a)$   
 $\text{add}(a, \text{null}) = a$   
 $\text{card}(\text{add}(\text{new}(n), o)) = 1$   
 $\text{card}(\text{add}(a, o)) \perp \text{ if } \neg \text{contains}(a, o) \wedge \text{card}(a) = \text{bound}(a)$   
 $\text{card}(\text{add}(a, o)) \geq \text{card}(a) \quad \vee \perp$  (von Subtyp festzulegen)  
 $\text{card}(\text{add}(a, c)) \geq \max(\text{card}(a), \text{card}(c)) \quad \vee \perp$   
 $\text{bound}(\text{remove}(a, \_)) = \text{bound}(a)$   
 $\text{remove}(\text{new}(n), o) = \text{new}(n)$   
 $\text{remove}(a, \text{null}) = a$   
 $\text{remove}(\text{add}(a, o'), o) = \text{add}(\text{remove}(a, o), o')$   
 $\text{card}(a) - 1 \leq \text{card}(\text{remove}(a, o)) \leq \text{card}(a)$   
 $\max(\text{card}(a) - \text{card}(c), 0) \leq \text{card}(\text{remove}(a, c)) \leq \text{card}(a)$   
 $\text{bound}(\text{replace}(a, \_)) = \text{bound}(a)$   
 $\text{replace}(a, \text{null}) = \text{new}(\text{bound}(a))$   
 $\text{replace}(a, o) = \text{add}(\text{new}(\text{bound}(a)), o)$   
 $\text{replace}(a, c) = \text{add}(\text{new}(\text{bound}(a)), c) \quad \vee \perp$   
 $\text{contains}(a, o) = \text{card}(a) > \text{card}(\text{remove}(a, o))$   
 $\text{collection}(\text{add}(\text{new}(n), c)) = c$   
 $\text{object}(\text{add}(\text{new}(n), o)) = o \quad \text{if } n = 1 \quad \text{else } \perp$

**Abbildung 2:** ADT *Association* zur einheitlichen Repräsentation von Zu-1- und Zu-*n*-Beziehungen. *Association* ist absichtlich unterspezifiziert (s. Text).

Die Operatoren *collection* und *object* dienen der Verwendung von *Association*-Objekten in Ausdrücken, die eine *Collection* bzw. ein Objekt erwarten, so z. B. in For-each-Schleifen oder bei einem Test auf Gleichheit mit einem Objekt. Beide Operationen können durch einen Compiler automatisch eingefügt werden (ähnlich dem sog. Auto-Unboxing [Sun04]). Man beachte jedoch, daß hier die Unterscheidung zwischen Zu-1- und Zu-*n*-Beziehungen wieder eingeführt wird: Die Anwendung von *object* auf einem *Association*-Objekt, dessen Kardinalität als  $> 1$  angegeben wurde, führt zu einem Fehler.

Dem aufmerksamen Leser wird aufgefallen sein, daß *Association* unterspezifiziert ist. So ist beispielsweise nicht klar, was passiert, wenn einer Instanz von *Association* ein Objekt hinzugefügt wird, das sie schon enthält. Dies ist Absicht und liegt darin begründet, daß mit *Association* nicht festgelegt sein soll, ob es sich beim Inhalt um Mengen, Multimengen, Listen oder was auch immer handeln soll. Da aber keiner dieser konkreten Typen ein Supertyp aller anderen ist, ist hier *Association* gewissermaßen als abstrakter ADT definiert, was soviel heißen soll wie daß es keine Implementierungen gibt, die genau seiner Spezifikation entsprechen, d. h., die keine zusätzlichen funktionalen Eigenschaften haben.

### 3.2 Implementierung und Verwendung des ADT *Association* in objektorientierten Programmiersprachen

Der ADT *Association* ist analog zur Wurzel *Collection* eines *Collection-Frameworks* wie dem *Smalltalks* oder *Javas* (in *Association* durch den ADT *Collection* repräsentiert) zu sehen, was soviel heißt wie daß konkrete Implementierungen Eigenschaften hinzufügen und insbesondere einen Typparameter (für den Typ der bezogenen Objekte, hier durch *Object* vertreten) haben können. *Association* wird also typischerweise als abstrakte Klasse implementiert, von der andere, konkrete ableiten. Da wir hier aber an Implementierungsdetails (und auch an den zahlreichen Erweiterungsmöglichkeiten, die denkbar sind) nicht interessiert sind, nehmen wir im folgenden an, daß *Association* eine konkrete Klasse ist, die über alle Operationen des ADT *Association* verfügt. Genau wie *Collection* in *Java*, *C#* und anderen Sprachen sei *Association* dem Compiler bekannt, so daß er bestimmte syntaktische und semantische Tests durchführen sowie speziellen Code generieren kann. So kann der Compiler beispielsweise bei einer (impliziten) Konversion des Inhalts einer Assoziation mit Kardinalität  $> 1$  in ein Objekt (per *Auto-Unboxing*; s. o.) einen semantischen Fehler anzeigen.

Da sie der Umsetzung von Beziehungen dienen, können nur Instanzvariablen (Felder) mit *Association*-Typen (konkreten Implementierungen unseres ADT *Association*) deklariert werden. Wir nennen diese Instanzvariablen dann *Assoziationen* und unterscheiden sie fortan von den anderen Instanzvariablen sorgfältig. Anders als normale Instanzvariablen sind *Assoziationen* nämlich nicht zuweisbar — sie können also weder auf der linken Seite einer Zuweisung noch als formale Parameter von Methoden auftreten. Die Speicherplätze, die eine *Assoziation* zur Verfügung stellt, können damit wie die indizierten Instanzvariablen *Smalltalks* [GR83] als fixer Bestandteil der Objekte, zu denen die *Assoziation* (hier allerdings als Inhalt einer benannten Instanzvariable) gehört, angesehen werden — es handelt sich bei *Assoziationen* also gewissermaßen um benannte indizierte Instanzvariablen, die gegenüber den einfachen indizierten Instanzvariablen den Vorteil haben, daß es von ihnen mehrere pro Objekt geben kann.

Da *Assoziationen* in direkter Konkurrenz zu gewöhnlichen Instanzvariablen stehen (sie sollen diese immer dann ersetzen, wenn durch eine Instanzvariable eine Beziehung dargestellt wird), sollen sie auch so ähnlich deklariert werden. Insbesondere soll nicht der *Containertyp* bei der Deklaration im Vordergrund stehen (wie er es bei der Deklaration von Instanzvariablen, die *Zu-n-Beziehungen* mit generischen *Collections* umsetzen, tut, also z. B. bei *Collection*<Angestellter>), sondern der *Elementtyp*. Statt

```

class Firma {
    Angestellter(*) angestellte = new Association<Angestellter>();
    Arbeitsplatz(*) arbeitsplaetze = new Association<Arbeitsplatz>();
    ICollection<Arbeitsplatz> freieArbeitsplaetzeMitTelefon() {
        ICollection<Arbeitsplatz> ergebnis = new List<Arbeitsplatz>();
        foreach (Arbeitsplatz aplz in arbeitsplaetze)
            if (aplz.mitTelefon && aplz.angestellte.Count() == 0)
                ergebnis.Add(aplz);
        return ergebnis;
    }
}

class Angestellter {
    Arbeitsplatz(1) arbeitsplatz = new Association<Arbeitsplatz>();
}

class Arbeitsplatz {
    bool mitTelefon;
    Angestellter(2) angestellte = new Association<Angestellter>();
}

```

**Abbildung 3:** Beispiel aus Abbildung 1 nebst Ergänzungen, mit Assoziationen umgesetzt

```
Angestellter angestellter;
```

für eine Zu-1-Beziehung schreiben wir für eine Zu-*n*-Beziehung daher nicht

```
Association<Angestellter> angestellte = new Association<Angestellter>();
```

wie man vielleicht annehmen könnte, sondern

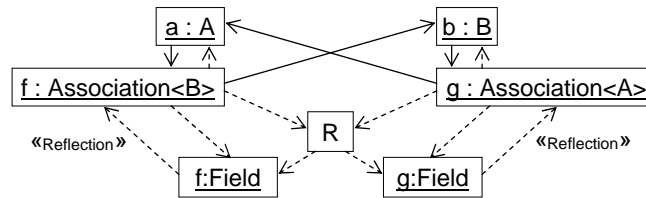
```
Angestellter(<x>) angestellte = new Association<Angestellter>();
```

wobei *<x>* für die Kardinalität (mit „\*“ für „∞“) steht. Dies ist auch insofern gerechtfertigt, als, wie wir gleich sehen werden, der genaue Containertyp, Association oder ein Subtyp davon, außer bei der Initialisierung von Assoziationen keine Rolle spielt — Assoziationen können daher immer vom Typ Association angenommen werden. Abbildung 3 zeigt das auf Assoziationen beruhende Programm aus Abbildung 1 nebst Änderungen.

Wie man dem Beispiel entnehmen kann, kann über die Elemente einer Assoziation (per impliziter Konversion in eine Collection) mittels `foreach` iteriert werden — es werden dann der Laufvariable der Reihe nach die Elemente, die mit dem Besitzer der Assoziation (hier repräsentiert durch das implizite `this`) über diese in Beziehung stehen, zugewiesen. Für den Fall der Zu-1-Beziehung `arbeitsplatz` stellt das zumindest syntaktisch eine Verschlechterung dar, da man hier eine gewöhnliche Instanzvariable zur Repräsentation der Beziehung verwenden und somit die Schleife nebst Laufvariable weglassen könnte (vgl. Abbildung 1), und tatsächlich war eines unserer wichtigsten Anliegen zunächst, die Verwendung von Zu-*n*-Beziehungen syntaktisch der von Zu-1-Beziehungen anzugleichen (und nicht umgekehrt, wie es jetzt der Fall ist). Die Erkenntnisse, die im nächsten Unterabschnitt dargelegt werden, haben uns jedoch eines besseren belehrt.

### 3.3 Objekte vom Typ Association in Iterationen und LINQ

Wenn man die abstrakte Klasse Association (als Implementierung des ADT *Association*) das Interface `IEnumerable` implementieren läßt, können ihre Instanzen wie die einer Collection auch ohne Umformung mittels `collection(.)` in For-each-Schleifen und LINQ-



**Abbildung 4:** Objektdiagramm für die Zuweisung  $a.f += b$  bzw.  $b.g += a$ . Gestrichelte Pfeile stellen die für die Herstellung der Rückrichtung notwendigen Verknüpfungen dar (s. Text).

Abfragen vorkommen. Dadurch werden LINQ-Abfragen aber auch auf Zu-1-Beziehungen ausgedehnt. So kann man dann statt dem auf Abbildung 1 basierenden

```

anzErreichbar = 0;
foreach (Angestellter agst in firma1.angestellte)
    if (agst.arbeitsplatz != null && agst.arbeitsplatz.mitTelefon)
        anzErreichbar += 1;
  
```

mittels der LINQ-Methoden `Select()`, `Where()` und `Count()` auch

```

anzErreichbar = (firma1.angestellte.Select(
    agst => agst.arbeitsplatz).Where(aplz => aplz.mitTelefon).Count());
  
```

schreiben, wobei hier ausgenutzt wird, daß `arbeitsplatz` auch eine Assoziation ist (was u. a. erlaubt, den gern vergessenen Test auf null wegzulassen). Entsprechend läßt sich in der Klasse `Angestellter` eine Methode `ICollection<Angestellter> teilenArbeitsplatz()` wie folgt implementieren (diesmal in LINQ-Abfragesyntax):

```

return (from aplz in arbeitsplatz
        from agst in aplz.angestellte select agst).ToList();
  
```

Man beachte, daß diese Implementierung invariant gegenüber Veränderungen der Kardinalitäten von `arbeitsplatz` und `angestellte` ist: Ergibt sich irgendwann, daß ein Angestellter auch an mehreren verschiedenen Arbeitsplätzen sitzen kann, ändert sich nichts.

## 4 Einführung von Bidirektionalität

Eine bidirektionale Beziehung als aus zwei unidirektionalen zusammengesetzt zu implementieren ist nichts grundsätzlich Schlechtes (ein relationales Datenbanksystem wird, wenn beide Richtungen häufiger genutzt werden, auch zwei Indextabellen aufbauen) — es kommt nur darauf an, den Programmierer von der Verantwortung zu befreien, die beiden zu koordinieren. Daß diese Verantwortung eine Belastung ist, zeigt Abbildung 4 am Beispiel der Herstellung einer 1:1-Beziehung zwischen zwei Objekten `a` und `b` mit ihren Assoziationen `f` bzw. `g`, die, ausgedrückt durch  $a.f += b$  (oder  $b.g += a$ , aber nicht beides) zur Folge haben soll, daß `a` über `f` mit `b` und `b` über `g` mit `a` verbunden ist.

Das Problem hierbei ist, daß zur Umsetzung die Angabe von `f`, der Assoziation, die `b` aufnehmen soll, und von `b`, dem Objekt, zu dem die Beziehung hergestellt werden soll, nicht ausreicht: Neben `a`, das für den Eintrag der Rückrichtung benötigt wird, muß auch noch `b`'s Assoziation `g`, die die Rückrichtung repräsentiert, angegeben werden. Insbesondere letzteres ist ein Problem, da `g` von `b` abhängt (jedes Objekt `b` hat seine eigene

**abstract data type** *Relation***imports** *Role***syntax**

|              |                        |                                   |
|--------------|------------------------|-----------------------------------|
| declare:     | $Role \times Role$     | $\rightarrow Relation \vee \perp$ |
| counterrole: | $Relation \times Role$ | $\rightarrow Role \vee \perp$     |

**semantics**

$\forall r, s, t \in Role, r \neq s \neq t \neq r$ :

new( $r, r$ )  $\perp$

counterrole(new( $r, s$ ),  $t$ )  $\perp$

counterrole(new( $r, s$ ),  $r$ ) =  $s$

counterrole(new( $r, s$ ),  $s$ ) =  $r$

**abstract data type** *Role***imports** *Object, Association***syntax**

|        |                      |                                      |
|--------|----------------------|--------------------------------------|
| new:   |                      | $\rightarrow Role$                   |
| apply: | $Role \times Object$ | $\rightarrow Association \vee \perp$ |

**abstract data type** *HalfRelation***imports** *Relation, Role, Association, Object***syntax**

|         |   |                            |
|---------|---|----------------------------|
| new:    | $Object \times Relation \times Role \times Association$ | $\rightarrow HalfRelation$ |
| add:    | $HalfRelation \times Object$                            | $\rightarrow HalfRelation$ |
| remove: | ...   |                            |

**semantics**

$\forall a, b \in Object, r \in Role, R \in Relation$ :

$\langle \text{add}(\text{new}(a, R, r, r(a)), b); \text{new}(b, R, \text{counterrole}(R, r), \text{counterrole}(R, r)(b)) \rangle =$   
 $\langle \text{new}(a, R, r, \text{add}(r(a), b)); \text{new}(b, R, \text{counterrole}(R, r), \text{add}(\text{counterrole}(R, r)(b), a)) \rangle$

remove ...

**Abbildung 5:** ADTs *Relation*, *Role* und *HalfRelation* zur Repräsentation bidirektionaler Beziehungen. Die Definition von *HalfRelation* folgt ab remove analog zu *Association* (Abbildung 2).

Assoziation  $g$ ) und  $b$  mehrere Assoziationen haben kann, so daß nicht automatisch klar ist, welcher  $a$  hinzugefügt werden soll. Es muß also eine Möglichkeit geben, von einem Objekt und seiner Stelle in einer Relation  $R$  auf die Assoziation zu schließen, die „seine“ (die ihm zugeordnete) Richtung der Relation repräsentiert.

#### 4.1 Spezifikation mit abstrakten Datentypen

Um dies zu erreichen, führen wir zunächst einen ADT *Relation* ein, der Relationsdeklarationen spezifiziert. Den Konstruktor von *Relation* nennen wir *declare*, um auszudrücken, daß er nicht irgendwann im Programmablauf, sondern, eben als Deklaration, bereits bei der Übersetzung ausgewertet wird. Neben den Argumenttypen (den Typen, die den Stellen der Relation zugeordnet sind und die deren Herkunft bestimmen), die wir hier weglassen und implizit als *Object* annehmen<sup>2</sup>, verlangt der Konstruktor (die Deklarati-

<sup>2</sup> Wir hätten Sie (wie zuvor auch schon bei *Association*) als Typparameter der Definition des ADT angeben können; da sie aber für unsere weiteren Betrachtungen keine Rolle spielen, lassen wir sie weg.

on) die Angabe zweier Funktionen, die jeweils ein Objekt auf eine Assoziation abbilden (und zwar genau die, die die Relation für das Objekt in Richtung auf seine Gegenüber realisiert). Diese Funktionen werden selbst durch einen ADT repräsentiert, den wir (zugegebenermaßen etwas unbeholfen) *Role* genannt haben. Die Syntax beider Datentypen sowie die Semantik von *Relation* sind in Abbildung 5, oben, zu sehen; die Semantik von *Role* wird im Kontext seiner Verwendung weiter unten definiert werden. Es sei jedoch hier schon bemerkt, daß wir dabei anstelle von  $apply(r, o)$  wie für Funktionsanwendungen üblich  $r(o)$  schreiben werden. Daß wir hier überhaupt auf eine Funktion höherer Ordnung zurückgreifen müssen, deckt sich damit, daß die Implementierung nicht ohne Reflektion (Metaprogrammierung) auskommt (s. Abschnitt 5).

Als letztes benötigen wir noch einen Datentyp, der die Pflege von bidirektionalen Beziehungen erlaubt. Im Falle unidirektionaler Beziehungen geschah dies ja mittels des ADT *Association*, der jedoch wie oben erläutert hier nicht mehr ausreicht, da ihm die für die Pflege der Rückrichtung benötigten Parameter fehlen. Um den Anwendungscode soweit wie möglich von der Gerichtetheit der Relationen unabhängig zu machen, definieren wir einen neuen ADT *HalfRelation* mit bis auf den Konstruktor zu *Association* identischer Syntax (Abbildung 5, unten). Auch die Semantik von *HalfRelation* basiert wesentlich auf dem ADT *Association*, für den er damit eine Art Wrapper bildet.

Im Gegensatz zu dem von *Association* verlangt der Konstruktor von *HalfRelation*, *new*, die Angabe des besitzenden Objekts, also des Objekts, von dem die Assoziation ausgeht. Man beachte, daß dieses Objekt, *a*, im Fall des Ausdrucks  $add(a.f, b)$  des Beispiels aus Abbildung 4 zwar im Kontext bekannt ist, nicht aber der Assoziation *f*, der *b* hinzugefügt werden soll. Dies wird durch die Einführung von *HalfRelation*, dessen Instanz *f'* neben einer Assoziation *f* auch ihren Besitzer *a* kennt, korrigiert:  $add(a.f, b)$  delegiert das Hinzufügen zur eigenen Richtung an die zur Halrelation *f'* gehörende Assoziation *f* und das Hinzufügen der Rückrichtung an die entsprechende Assoziation von *b*. Damit letztere bestimmt werden kann, verlangt *new* weiterhin die Angabe einer Relationsdeklaration *R* und einer Rolle *r*; die Funktionsanwendung *counterrole(R, r)* liefert dann eine Funktion, die, auf *b* angewendet, die Assoziation der Rückrichtung (*g* im obigen Beispiel) liefert, der dann *a* hinzugefügt wird. Mit anderen Worten: Für  $a.f' = new(a, R, r, f)$  mit  $f \in Association$  führt  $add(a.f', b)$  zu  $add(f, b)$  und zu  $add(counterrole(R, r)(b), a)$ . Man beachte, daß die axiomatische Definition der Semantik von *add* die Identität von  $r(a)$  und *f* im obigen Beispiel ausnutzt.

## 4.2 Übertragung auf die objektorientierte Programmierung

Um eine bidirektionale Beziehung in einem objektorientierten Programm herzustellen, deklarieren wir nun einfach Instanzvariablen mit Klassen, die den ADT *HalfRelation* implementieren. Gegenüber dem Beispiel aus Abbildung 3 ändert sich der Code nur an den Stellen der Deklarationen, bei denen nun *HalfRelation* anstelle von *Association* stehen muß, wobei der Konstruktoraufruf einen Parameter erhält, der die Relation benennt (wir setzen wieder die Existenz einer konkreten Klasse *HalfRelation* voraus):

```
Arbei tspl atz(1) arbei tspl atz = new Hal fRel ati on<Arbei tspl atz>(Si tzt);
```

in der Klasse *Angestellter* sowie

```
Angestellter(2) angestellter = new HalfRelation<Angestellter>(Sitzt);  
in Arbeitsplatz. Die Relation Sitzt wird im Programm durch  
relation Sitzt(Angestellter, Arbeitsplatz, Arbeitsplatz, angestellter)  
deklariert.
```

Man beachte, daß bis auf die Angabe der Relation sämtliche Information, die zur Instanziierung einer Halbrelation benötigt wird, vom Compiler aus dem Kontext gewonnen werden kann: Das besitzende Objekt steckt in *this*, die Rolle ergibt sich aus dem Namen der Instanzvariable, die die Halbrelation benennt (der sie im Rahmen der Deklaration zugewiesen wird), und die Assoziation wird im Konstruktor neu erzeugt. Die syntaktische Last für den Programmierer ist also ausgesprochen gering.

## 5 Prototypische Implementierung in C#

In einem ersten Ansatz haben wir die oben beschriebenen ADTs *Association* und *HalfRelation* in parametrisierte C#-Bibliotheksklassen umgesetzt, die der Deklaration entsprechender Instanzvariablen und der Erzeugung ihrer Inhalte dienen. Nicht umsetzen konnten wir damit die Bedingungen, daß *nur* Instanzvariablen mit ihnen deklariert, daß diese Instanzvariablen keinen anderen Variablen zugewiesen (auch nicht vom Typ *Object*) und daß ihnen selbst nach der Initialisierung keine weiteren Werte zugewiesen werden dürfen (letzteres kann aber zumindest durch Verwendung des C#-Modifiers *readonly* bei der Instanzvariablendeklaration erzwungen werden). Auch die Ableitung der für die Instanziierung einer Halbrelation notwendigen Information aus dem Kontext konnten wir auf diese Weise nicht umsetzen — sie muß beim Konstruktoraufruf explizit übergeben werden.

Anstelle einer Relationsdeklaration wie oben beschrieben übergeben wir beim Konstruktoraufruf von *HalfRelation* neben *this* den Namen der Relation als String. Bei Aktualisierungen von Instanzen von *HalfRelation* wird dann anhand des hinzugefügten oder weggenommenen Objekts die Klasse der Gegenseite ermittelt und dort, via Reflektion, nach der Instanzvariable gesucht, die eine Halbrelation mit gleichem Relationsnamen enthält (und die — bei homogenen Relationen — von der Ausgangshalbrelation verschieden ist). Die Aktualisierung wird dann auf der Gegenseite entsprechend durchgeführt.

## 6 Zusammenfassung und Schluß

Wir machen die objektorientierte Programmierung relationaler, indem wir die herkömmliche Umsetzung von Zu-1- und Zu-*n*-Beziehungen über direkte Zeiger bzw. *Collections* vereinheitlichen. Zu diesem Zweck führen wir Assoziationen ein, die den indizierten Instanzvariablen *Smalltalks* insoweit gleichen, als sie Beziehungen zu mehreren anderen Objekten gleichzeitig herzustellen erlauben, ohne dabei (wie *Collections*) von ihrem Objekt getrennt werden zu können, die aber im Gegensatz zu den indizierten Instanzvariablen *Smalltalks* benannt sind, so daß ein Objekt mehrere haben kann. Indem wir zusätzlich Relationen als Paare von korrespondierenden Assoziationen einführen, ermöglichen

wir auch noch die vollautomatische Pflege von bidirektionalen Beziehungen. Im Zusammenspiel mit relationalen Abfragesprachen wie LINQ, die auf Collections operieren können, ergibt sich somit eine relationale Sicht auf objektorientierte Daten, die trotzdem den navigierenden (d. h., zeigerbasierten) Charakter der objektorientierte Programmierung vollständig erhält, die also insbesondere ohne Relationen als von Objekten separat zu verwaltende Tupelmengen auskommt. Damit glauben wir, die Vorzüge beider Weltanschauungen in einer gemeinsamen vereint zu haben.

## Literaturverzeichnis

- [ABS04] Amelunxen, C.; Bichler, L.; Schürr, A.: Codegenerierung für Assoziationen in MOF 2.0. In Proceedings of Modellierung 2004. Lecture Notes in Computer Science, vol. P-45. Springer-Verlag, Berlin, Heidelberg, 2004, S. 149-168
- [BGE07] Balzer, S.; Gross, T.R.; Eugster, P.: A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships. In Proceedings of ECOOP 2007 - Object-Oriented Programming. Lecture Notes in Computer Science, vol. 4609. Springer-Verlag, Berlin, Heidelberg, 2007, S. 323-346
- [BMT07] Bierman, GM; Meijer, E.; Torgersen, M.: Lost in translation: formalizing proposed extensions to C#. OOPSLA 2007, 479–498.
- [BW05] Bierman, G.; Wren, A.: First-Class Relationships in an Object-Oriented Language. In Proceedings of ECOOP 2005 - Object-Oriented Programming. Lecture Notes in Computer Science, vol. 3586. Springer-Verlag, Berlin, Heidelberg, 2005, S. 25-29
- [CM84] Copeland, C.; Maier, D.: Making smalltalk a database system. In ACM SIGMOD Records, vol. 14, 2, 1984, S. 316-325
- [Ge09] Gessenharter, D: Implementing UML associations in Java: a slim code pattern for a complex modeling concept. In Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages. RAOOL '09. ACM Press., New York, 2009, S. 17-24
- [GR83] Golberg, A; Robson, D.: Smalltalk-80: The Language and its Implementation. Addison Wesley, Reading, Massachusetts, 1983.
- [NPN08] Nelson, S.; Noble, J.; Pearce, D.J.: Implementing first-class relationships in Java. In Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages. RAOOL'08. ACM Press, New York, 2008.
- [Omg06] Object Management Group (OMG): Meta Object Facility (MOF) Core Specification, OMG Available Specification, Version 2.0 (OMG Document formal/06-01-01), 2006.
- [Øs07] Østerbye, K.: Design of a class library for association relationships. In Proceedings of the 2007 Symposium on Library-Centric Software Design. LCSD '07. ACM Press, New York, 2007, S. 67-75
- [PN06] Pearce, D.J.; Noble, J.: Relationship aspects. In Proceedings of the 5th international Conference on Aspect-Oriented Software Development. AOSD '06. ACM Press, New York, 2006, S. 75-86
- [Ru87] Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. OOPSLA '87. ACM Press, New York, 1987, S. 466-481
- [SS09] Stadtler, D.; Steimann, F.: Wie die Objektorientierung relationaler werden sollte — Eine Analyse aus Sicht der Datenmodellierung. eingereicht bei: Modellierung 2010 (von den Autoren erhältlich).
- [Sun04] Sun Microsystems, Inc.: New Features and Enhancements J2SE 5, 2004, <http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html> (letzter Zugriff: 09.10.2009)