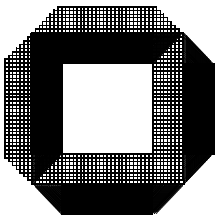

UNIVERSITÄT KARLSRUHE
FAKULTÄT FÜR INFORMATIK

Postfach 6980, D-7500 Karlsruhe 1

Die Datenbankprogrammiersprache GOMpl

Alfons Kemper[*] *Christoph Kilger*[†] *Guido Moerkotte*[†] *Hans-Dirk Walter*[†] *Andreas Zachmann*[†]

[*]RWTH Aachen
Lehrstuhl für Informatik III
5100 Aachen, Germany
[*kemper*]@informatik.rwth-aachen.de

[†]Universität Karlsruhe
Fakultät für Informatik
7500 Karlsruhe, Germany
[*kilger*|*moer*|*walter*|*zachmann*]@ira.uka.de

Interner Bericht Nr. 25/92 * November 1992

Zusammenfassung

Das objektorientierte Datenbanksystem GOM wurde am Institut für Programmstrukturen und Datenorganisation (IPD) im Rahmen des Sonderforschungsbereichs 346 "Rechnerintegrierte Konstruktion und Fertigung von Bauteilen" entwickelt. Dieses Dokument enthält eine Beschreibung des Datenmodells von GOM, der persistenten Programmiersprache GOMpl. Hierbei gehen wir in Form einer Einführung auf die grundlegenden Konzepte von GOM ein. Insbesondere werden die für GOMpl zentralen Konzepte *Typ, Objekt, Persistenz, Polymorphie* und *Generizität* beschrieben. Weiterhin gehen wir auf Nebenläufigkeit von GOM-Anwendungen sowie auf die Schnittstelle zwischen GOMpl und der Programmiersprache C ein. Im Ausblick skizzieren wir mögliche Erweiterungen von GOMpl, die als Konzepte bereits vorliegen, aber noch nicht in die Sprache aufgenommen wurden. Eine ausführlichere Beschreibung von GOMpl findet sich in dem GOM-Handbuch [KKM+90].

1 Einleitung

Elektronisch verarbeitbare Informationen sind heute existenzbestimmend für den modernen Industriebetrieb. Dementsprechend groß sind die Anstrengungen, die allerorten unternommen werden, um immer bessere Systeme zur Informationsverarbeitung zu entwickeln. Während in Bereichen, die originär mit der Verwaltung großer Datenbestände befaßt sind—üblicherweise mit administrativ und kommerziell umschrieben—schon seit geraumer Zeit Rechner mit Hilfe spezialisierter Programmsysteme für diese Aufgabe eingesetzt werden, wurde bis vor kurzem die Datenhaltung im ingenieurwissenschaftlichen Bereich eher als untergeordnet betrachtet. Hier schien sich der Nutzen des Rechners im wesentlichen bei der Umsetzung technischer Verfahren, in Form von Programmen beschrieben, zu manifestieren.

Mit der Erkenntnis des wirtschaftlichen Nutzens, der in der Zusammenfügung des Sammelsuriums von Insellösungen der Datenverarbeitung zu einer geschlossenen Welt liegt—in ihrer weitreichendsten Form mit dem Schlagwort CIM umschrieben—wurde die Datenhaltung zu einem Dreh- und Angelpunkt auch des technischen Sektors. Unterschiedlichste Programmsysteme sollen auf einem gemeinsamen, integrierten sowohl Produkt als auch Produktionsvorgang beschreibenden Datenbestand arbeiten.

Die im administrativ-kommerziellen Bereich so erfolgreich eingesetzten Datenbanksysteme haben sich dort als geeignete Medien der Datenintegration bewährt. Grob gesprochen dienen sie zur redundanzarmen, konsistenten Speicherung großer Datenmengen, auf die jedoch effizient zugegriffen werden kann. Sie erlauben auf einfache Art die Schematisierung der zu erwartenden Daten mit Hilfe von Datenmodellierungswerkzeugen. Sie stellen Funktionen zur Verfügung, mit denen neue Daten dem Bestand hinzugefügt, Daten entfernt werden können und auf beliebige Teile der Datenbestände effizient zugegriffen werden kann, wobei die Zugriffsgeschwindigkeit häufig wiederkehrender Anfragearten explizit durch sogenannte Zugriffspfade noch weiter gesteigert werden kann. Es können a-priori Bedingungen formuliert werden, deren Erfüllung einen Datenbestand als korrekt ausweisen, und die vom Datenbanksystem automatisch überwacht werden. Dies und vielfältige Sicherungsmaßnahmen gegen Fehler und Systemausfälle (Recovery) sind wesentliche Beiträge zur Wahrung der Datenintegrität, die ohne Datenbanksysteme von Anwendungen selber erbracht werden müßten. Gleiches gilt für die Koordination mehrerer Anwendungen, die gleichzeitig auf denselben Daten arbeiten. Auch hier müßten sich die verschiedenen Anwendungen beim Zugriff auf ihre gemeinsamen Daten weitgehend selbst synchronisieren. Und schließlich bleibt den Anwendungsprogrammen die Organisation der Daten auf den Speichermedien verborgen—im Gegensatz zur herkömmlichen Dateiverwaltung—wodurch sie von einer Umorganisation der Daten und einer Erweiterung des Datenschemas weitgehend unbetroffen bleiben.

Am weitesten verbreitet sind sogenannte relationale Datenbanksysteme, die erlauben, die Strukturen der potentiell anfallenden Daten mittels Tupel, das sind eine Reihe von Attributen, deren Werte aus einigen wenigen vordefinierten Wertebereichen stammen müssen, zu definieren und die zu Relationen, das sind Mengen solcher Tupel, zusammengefaßt werden. Diese tabellenartigen Konstrukte können dann unter Anwendung fest vorgegebener generischer Funktionen manipuliert werden. Der „Charme“ des Relationenmodells liegt in seiner Einfachheit, die eine weitgehende Formalisierung ermöglichte. So geeignet solche Systeme für ihre ursprünglichen Anwendungsgebiete auch sind, zur Beschreibung der im Entwurfs- und Produktionsprozeß eines Betriebes anfallenden Daten stellen sie nicht genügend ausdrucksstarke Mittel zur Verfügung. Zwar sind auch hier große Datenmengen zu bewältigen, die sich aber durch eine viel reichhaltigere, mittels Tupelmengen nur unzulänglich zu beschreibende Struktur auszeichnen, und die auf weitaus spezifischere und vielfältigere Art manipuliert werden müssen, als es die generischen Funktionen relationaler Systeme ermöglichen [SZ87]. Insgesamt ist die Welt des industriellen Betriebes zu komplex, um durch so einfache, wenig Abstraktionsmöglichkeiten bietende Modelle,

wie das Relationenmodell, ausreichend erfaßt zu werden.

In diese Lücke stoßen objektorientierte Konzepte, die in den unterschiedlichsten Bereichen der Informationsverarbeitung mit Enthusiasmus aufgenommen wurden. Bei der Erstellung komplexer Programmsysteme mit höchsten Ansprüchen an Sicherheit, Wartbarkeit und Erweiterbarkeit haben sie sich schon glänzend bewährt. Objektorientierte Modelle bieten hohe Expressivität, wobei sie immer noch auf einer relativ einfachen Begriffswelt basieren. Der Grundgedanke der Objektorientierung besteht darin, Entitäten aus der Realität, z.B. einen Gegenstand oder, allgemeiner, einen Sachverhalt, als Individuum mit eigener, unverwechselbarer Identität in die Informationswelt aufzunehmen, und dieses Individuum als *Objekt* durch einen sich über die Zeit ändernden Zustand und ein spezifisches Verhaltensmuster zu beschreiben. Es herrscht allgemeiner Konsens, daß objektorientierte Modellierungsmethoden für technische Anwendungen allen anderen heute bekannten Modellen, wie hierarchischen, netzwerkartigen oder relationalen Modellen, vorzuziehen sind.

Mit GOM wird nun eine objektorientierte Datenbankprogrammiersprache vorgelegt, die die Erstellung von Ablauf- und Verfahrensbeschreibungen sowie die Modellierung dauerhaft zu speichernder Daten nahtlos verknüpft. Die bisher künstlich vorgenommene Trennung zwischen Methoden und Datenbank ist damit überwunden, und „Reibungsverluste“, die beim Zugriff von Methoden auf Teile der Datenbasis auftreten, werden vermieden. GOM ist nicht nur eine „reine“ Datenbanksprache, wie z.B. SQL, sondern bietet alle Möglichkeiten einer allgemeinen Programmiersprache, wie z.B. C, zusammen mit der Möglichkeit, die in Programmen verwendeten Daten ohne umständliche Dateiverwaltung dauerhaft zu speichern. GOM ist jedoch auch nicht nur eine persistente Programmiersprache, da eine Komponente zur Verwaltung von Typen und Operationen, eine sogenannte Schemaverwaltung, im System integriert ist. Zudem bietet GOM Mechanismen zur gezielten Optimierung des Datenzugriffs sowie Synchronisation und Recovery durch ein Transaktionskonzept†. Da GOM nicht nur für die vollständige Neu-Erstellung von ingenieurwissenschaftlichen Anwendungssystemen gedacht ist, also nicht nur für den Einsatz in der „nackten“ Fabrik, sondern auch existierende Programme zusammen mit GOM verwendet werden sollten, verfügt GOM auch über eine Schnittstelle zu beliebigen C-Programmen†.

GOM bietet alle wesentlichen Konzepte, die gemeinhin im Zusammenhang mit dem Begriff „Objektorientierung“ genannt werden und die in [ABD+89] zusammengefaßt sind:

1. Komplexe Objekte
2. Objektidentität
3. Datenkapselung
4. Typhierarchien und Vererbung von Eigenschaften eines Obertyps an seine Untertypen
5. Dynamisches Binden verfeinerter Operationen
6. Persistenz

Diese Begriffe werden im vorliegenden Handbuch eingehend erläutert.

Weitergehende Konzepte von GOM sind:

Strenge Typisierung, eine Eigenschaft, die es ermöglicht, alle Typfehler schon zur Übersetzungszeit eines Programmes zu erkennen. Hierdurch wird eine der häufigsten Fehlerursachen zur Laufzeit von Programmen vermieden. Auch läßt sich bei streng typisierten Programmen effizienterer Code erzeugen.

Polymorphe Operationen, die den Verlust an Flexibilität, der durch strenge Typisierung ansonsten zu beklagen wäre, ausgleichen helfen. Auch unterstützen polymorphe Operationen die Erstellung wiederverwendbarer „Operationsbibliotheken“.

Generische Typen, die ebenfalls dazu beitragen, Teile des Programmcodes in Form von „Typbibliotheken“ wiederzuverwenden.

Mit † gekennzeichnete Konzepte oder Konstrukte der Datenbankprogrammiersprache GOM sind in der vorliegenden Version noch nicht implementiert. Zunächst wollen wir jedoch ein kleines Beispielprogramm präsentieren und zeigen, wie man ein GOM-Programm übersetzt und ausführt.

2 Programm-Übersetzung und -Ausführung

Ein ausführbares Programm wird in GOM durch die Definition eines *Prozesses* beschrieben. Bei der Übersetzung der Datei, die eine solche Prozeßdefinition enthält, wird daraus vom Compiler eine ausführbare Binärdatei erzeugt. Die Definition eines Prozesses entspricht syntaktisch der Definition des Codes von Operationen, wie es in Abschnitt 6 beschrieben wird. Anstelle des bei einer Operationsdefinition benötigten *Codenamens* wird bei der Definition eines Prozesses jedoch der Name des ausführbaren Programms angegeben:

process ⟨ProcessName⟩ **is**
⟨VariablenSektion⟩
⟨OperationsRumpf⟩

Das folgende Programm soll die Worte „Hello World“ auf dem Bildschirm ausgeben:

```
process HelloWorld is
    print ( "Hello World\n" );
```

Unter der Annahme, daß das Programm in der Datei *hello.gom* abgelegt ist, kann es durch Aufruf von

```
GOMC hello.gom
```

übersetzt und gebunden werden. Voraussetzung hierfür ist jedoch die korrekte Installation des GOM-Systems.[1] Nachdem der Übersetzer seine Arbeit beendet hat, kann das Programm durch Eingabe von *HelloWorld* ausgeführt werden.

3 Sorten und Werte

Sorten repräsentieren Mengen von elementaren Werten, die nicht *änderbar (mutierbar)* sind, d.h. diese Sorten können ihren internen Zustand nicht verändern. Aus diesem Grund bezeichnen wir Sorten auch oft als *atomare Typen*. Man kann nur die (eingebauten) Funktionen verwenden, um eine neue Instanz einer Sorte zu erhalten. Aus diesem Grund besitzen Sorten in GOM auch keine Objektidentität; der Wert selbst repräsentiert sozusagen die Identität einer Instanz einer Sorte. Wegen der fehlenden Objektidentität können—konsequenterweise—Sorten nicht autonom in der Objektbank existieren. Sie können nur als Bestandteil komplexer Objekte (mit Objektidentität) existieren. Beispielsweise kann die Zahl 37 als Wert des Attributs *Alter* der *Person*-Instanz namens „Heinrich Maier“ in der Objektbank vorkommen.

Einige der in GOM eingebauten Sorten sind nachfolgend aufgeführt:

- *bool,* die booleschen Werte *true* und *false*
- *int,* die im Rechner darstellbaren ganzen Zahlen, z.B. $37, -5$
- *float,* Fließkommazahlen unterschiedlicher Präzision, z.B. $3.14, 2.5E5$

[1] Die Installation des GOM-Systems ist in einem separaten Dokument beschrieben [Zac92].

```
[virtual] type ⟨Typ-Name⟩ [supertype ⟨Obertyp-Name⟩] is
    [public ⟨Operationen-Liste⟩]
    [body ⟨Typ-Struktur⟩]
    [operations
        ⟨Operationen-Signatur⟩;
        ...
        ⟨Operationen-Signatur⟩;
    implementation
        ⟨Operationen-Implementierung⟩;
        ...
        ⟨Operationen-Implementierung⟩;]
end type⟨Typ-Name⟩;
```

Abbildung 1: Die syntaktische Struktur des Typdefinitionsrahmens

- *decimal*$(m, n)^\dagger$, d.h. Dezimalzahlen mit m Ziffern Präzision, wovon n Nachkommastellen sind
- *char*, beliebige Zeichen, z.B. „a“, „7“, aber auch nicht darstellbare Zeichen

Daneben kann ein Benutzer selbst Sorten als Aufzählungen der oben genannten Sorten definieren, z.B.

sort AmpelFarbe **is enum** (rot, gelb, grün);

4 Objekttypen

Eines der wichtigsten Hilfsmittel zur Strukturierung einer zu modellierenden „Miniwelt“ ist die Zusammenfassung von „gleichartigen“ Objekten durch *Typen*. Gleichartig bedeutet in diesem Zusammenhang, daß die Objekte die gleiche *Struktur* und das gleiche *Verhaltensschema* aufweisen. Struktur und Verhalten sind auch die beiden wichtigsten Beschreibungsmerkmale von Objekten.

Während in relationalen Modellen durch die Definition von Tabellen lediglich die Struktur von Objekten festgelegt wird, die Verarbeitung dieser Objekte jedoch über typunabhängige generische Operationen (z.B. der select-Operation) erfolgt, können in GOM typspezifische Operationen definiert werden. Damit ist eine weitaus genauere Erfassung der Semantik des zu modellierenden Umweltausschnitts möglich.

4.1 Der Typdefinitionsrahmen

Zur Definition neuer Objekttypen gibt es in GOM den sogenannten *Typdefinitionsrahmen*, dessen syntaktische Struktur in Abbildung 1 skizziert ist.

Mit [ ] gekennzeichnete Klauseln des Typdefinitionsrahmens sind optional, d.h. sie werden für eine vollständige Typspezifikation nicht notwendigerweise benötigt.

Ohne jetzt auf die Semantik der verschiedenen Teile dieses Typdefinitionsrahmens im Detail eingehen zu wollen—dies wird in den nachfolgenden Abschnitten schrittweise erfolgen—wollen wir hier kurz deren Bedeutung anreißen.

Der Name ⟨Typ-Name⟩ des neu definierten Typs muß eindeutig sein, d.h. er darf vorher noch nicht zur Benennung eines Typs verwendet worden sein. Ein GOM-Typ hat in jedem Fall genau einen Obertyp. Entweder wird der Obertyp als ⟨Obertyp-Name⟩ in der **supertype** Klausel explizit spezifiziert, oder aber der Typ *ANY* wird als impliziter Obertyp angenommen, falls

diese optionale Klausel fehlt. Der Untertyp, also der neu definierte Typ namens ⟨Typ-Name⟩, erbt alle Eigenschaften, d.h. Attribute und Operationen des Obertyps. GOM unterstützt also das Konzept der *singulären (einfachen) Vererbung*.

In der **public** Klausel werden die Operationen aufgeführt, die den sogenannten Klienten des Typs zum Zugriff auf Objekte bzw. zum Modifizieren des Zustands der Objekte des jeweiligen Typs zur Verfügung gestellt werden. Insofern stellt die **public** Klausel die Spezifikation der Schnittstelle des Typs nach außen dar.

In der **body** Klausel wird die interne Struktur des Objekttyps genauer spezifiziert. Wir unterscheiden tupelstrukturierte Typen, deren **body** aus einer Anzahl von benannten Attributen besteht, und Kollektionstypen. Die Unterschiede werden in nachfolgenden Abschnitten weiter ausgeführt.

In der **operations** Klausel werden die abstrakten Signaturen der dem Typ zugeordneten Operationen aufgeführt. Eine ⟨Operationen-Signatur⟩ spezifiziert den Namen der Operation, der innerhalb des Typs eindeutig sein muß[2], die Typen der Argumente, den Ergebnistyp und—optional—den Namen, unter dem die Implementierung der Operation in dem **implementation** Teil erfolgt (wird der Code-Name weggelassen, müssen Operationsname und Implementierungsname übereinstimmen).

Im **implementation** Teil des Typdefinitionsrahmens werden die im **operations** Teil aufgeführten Operationen codiert. Die Implementierung erfolgt in der GOM-Sprache, deren grundlegende Kontrollstrukturen der Programmiersprache C angelehnt sind. Darüberhinaus bietet die GOM-Programmiersprache weiterhin alle aus objektorientierten Sprachen bekannten Konzepte.

4.2 Struktur und Verhalten eines Objekttyps

Ein Objekttyp dient der allgemeinen Beschreibung einer Menge von ähnlichen Objekten. In GOM werden die individuellen Objekte durch Instantiierung des Typs erzeugt. Somit kann man den Typ als sogenannte Schablone („Plätzchenform“) auffassen, mit der man beliebig viele, gleich strukturierte Instanzen („Plätzchen“) generieren kann. Allerdings legt ein GOM-Typ nicht nur die Struktur, sondern auch das Verhalten seiner Instanzen fest. Somit kann man die Typbeschreibung als Spezifikation zweier dualer, sich gegenseitig ergänzender Dimensionen betrachten:

1. *Strukturelle Repräsentation*
   In der Typdefinition wird festgelegt, welche (interne) Struktur die Instanzen des betreffenden Typs haben. Die interne Struktur wird benötigt, um den internen Zustand der Objekte zu speichern.

2. *Verhaltensbeschreibung*
   Die Verhaltensbeschreibung besteht in GOM aus einer Kollektion von Operationen, die auf Objekten dieses Typs anwendbar sind. Grundsätzlich unterscheidet man drei Klassen von Operationen:

   - *Konstruktoren*, die dazu dienen, neue Instanzen zu generieren,
   - *Leseoperationen* oder *Observierer* und mit denen der aktuelle Zustand einer Objektinstanz abgefragt werden kann.
   - *Mutatoren*, mit denen der interne Zustand einer Objektinstanz verändert werden kann.

Die strukturelle Repräsentation eines Typs[3] wird in der **body**-Klausel festgelegt. Das Verhalten eines Typs wird durch die drei Klauseln **public**, **operations** und **implementation** bestimmt,

[2] Diese Bedingung wird durch *Overloading* noch entschärft.

[3] Genauer müßte es heißen: die strukturelle Repräsentation *der Objekte eines Typs* ... Wir werden uns die etwas unpräzise Formulierung aber auch weiterhin leisten, solange keine Mißverständnisse zu befürchten sind.

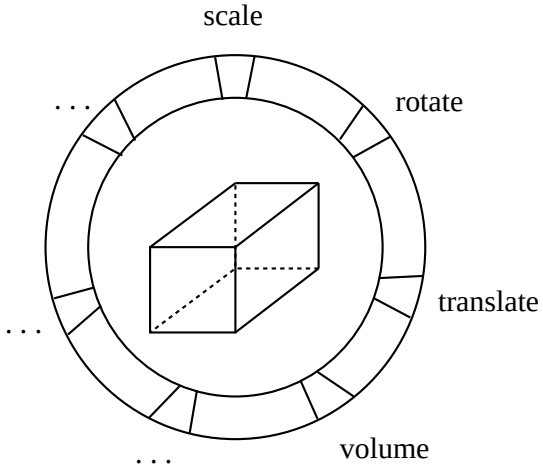

Abbildung 2: Schematische Darstellung der Objektkapselung

die die folgende Bedeutung haben:

public: In der **public**-Klausel werden die Namen der Operationen aufgeführt, die man „von außen“ auf dem Typ anwenden kann. Insofern verfolgt GOM das Konzept der Objektkapselung, das darin besteht, die interne Struktur eines Objekts hinter einer Kollektion wohldefinierter Operationen zu verstecken (information hiding). Für ein Beispielobjekt vom Typ *Cuboid* ist dies anschaulich in Abbildung 2 dargestellt. Die interne Repräsentation eines *Cuboid*-Objekts bestehe beispielsweise aus den Koordinaten der acht Begrenzungspunkte. Die Objektkapselung verbietet aber den direkten Zugriff auf diese Koordinaten—was ja auch leicht zu einem inkonsistenten Zustand der *Cuboid*-Repräsentation führen könnte. Vielmehr muß jeder Zugriff und jede Objektmanipulation über die Menge der Schnittstellenoperationen durchgeführt werden. In diesem Fall stehen als Mutatoren die wohlbekannten geometrischen Transformationsoperationen *rotate*, *scale* und *translate* zur Verfügung; als Observierer gibt es weitere Zustandsabfrage-Operationen wie *volume*.

operations: In der **operations**-Klausel werden die Signaturen der Operationen aufgeführt. Allerdings können einige der hier aufgeführten Operationen auch *private* Operationen sein, die nicht in der **public**-Klausel aufgeführt werden. Die privaten Operationen werden dann lediglich verwendet, um die Implementierung der öffentlichen Operationen modular durchführen zu können.

implementation: Zu den in der **operations**-Klausel aufgeführten Operationen wird die Realisierung (Programmierung) im **implementation**-Teil der Typdefinition vorgenommen.

5 Tupelstrukturierte Typen

Tupelstrukturierte Typen sind solche, deren **body**-Klausel ein Tupel bestehend aus beliebig vielen benannten Attributen definiert. In diesem Fall ist der **body** wie folgt spezifiziert[4]:

body $[A_1 : T_1;$
$\ldots;$
$A_n : T_n;]$

[4]Die hier verwendeten Tupelklammern [ ] sind nicht mit den Syntax-Klammern des Typdefinitionsrahmens zu verwechseln, die die Optionalität eines syntaktischen Konstrukts kennzeichnen.

Hierbei müssen die A_i $(1 \leq i \leq n)$ paarweise verschiedene Attributnamen sein, die T_i sind nicht notwendigerweise verschiedene Typnamen. Es können entweder Sorten oder komplexe Objekttypen sein. Es gibt auch keinerlei Restriktionen hinsichtlich rekursiver Objektstrukturen. Ein anschauliches Beispiel für eine rekursive Typdefinition wäre der Typ *Person*, der selbst wiederum ein Attribut *Ehepartner* vom Typ *Person* hätte.

Der nachfolgend definierte Typ stellt ein einfaches Beispiel für einen tupelstrukturierten Typ dar, nämlich einen *Vertex*, dessen Struktur einfach aus den drei Koordinaten X, Y und Z (jeweils vom Typ *float*) besteht.

```
type Vertex supertype ANY is
   public ...
   body [X: float;
         Y: float;
         Z: float;]
         ...
end type Vertex;
```

Eine Kurzschreibweise für den obigen Typ wäre auch folgende:

```
type Vertex is
   public ...
   body [X,Y,Z: float;]
   ...
end type Vertex;
```

Attribute gleichen Typs können also zusammengefaßt werden. Ebenso ist auch die Angabe von *ANY* als Obertyp nicht unbedingt erforderlich. Der vordefinierte Typ *ANY* ist standardmäßig Obertyp aller Typen. Falls die **supertype**-Klausel weggelassen wird, wird implizit *ANY* als direkter Obertyp angenommen.

6 Operationen

Den GOM Objekttypen sind Operationen zugeordnet, die das *Verhaltensmuster* der Instanzen des Typs festlegen. Wir unterscheiden zwischen *vordefinierten Operationen*, die implizit vom System zur Verfügung gestellt werden, und *benutzerdefinierten Operationen*, die vom Datentyp-Designer zu definieren sind.

6.1 Vordefinierte Operationen

6.1.1 Operationen zum Attributzugriff und zur Attributänderung

Für tupelstrukturierte Typen gibt es in GOM vordefinierte Operationen, die den Zugriff auf und die Zuweisung zu den Attributen bewerkstelligen. Wenn A_i ein Attribut vom Typ T_i innerhalb des tupel-strukturierten Typs T ist, dann werden die beiden nachfolgend deklarierten Operationen implizit für den Typ T zur Verfügung gestellt:

declare A_i: T|| $\rightarrow$ T_i;
declare A_i: T|| $\leftarrow$ T_i;

Bei dieser Deklaration grenzt das Sonderzeichen "||" den Typ des Empfängerobjekts—also den Typ, dem die Operation zugeordnet ist, von eventuell existierenden weiteren Argumenttypen ab. Die erste Operation ist eine sogenannte *Value reTurning Operation* (*VTO*), die den aktuellen Wert (genauer: die aktuelle Belegung) von A_i zurückliefert. Die zweite Operation ist eine *Value reCeiving Operation* (*VCO*), d.h. diese Operation weist dem Attribut A_i einen neuen Wert zu.

Sowohl die Deklaration der Signaturen als auch die Implementierung dieser beiden Operationen ist implizit. Obwohl diese Operationen für jeden tupelstrukturierten Typ und jedes

Attribut implizit vordefiniert sind, bleibt es dennoch dem Typ-Implementator überlassen, ob er diese Operationen durch Eintrag in die **public**-Klausel nach außen *sichtbar*, d.h. anwendbar machen will. Dies erfolgt durch Angabe von A→ bzw. A← wenn der direkte Lese- bzw. Schreibzugriff auf das Attribut A erlaubt sein soll. Sollen beide Arten von Zugriffen von außen möglich sein, kann anstelle der expliziten Angabe von A→ bzw. von A← verkürzend der Attributname A in der **public**-Klausel angegeben werden. Das in Abschnitt 6.2 angeführte Beispiel verdeutlicht dies noch einmal.

Wir wollen den Aufruf und die Benutzung dieser Operationen an unserem Beispiel-Typ *Vertex* illustrieren.

```
var einPunkt: Vertex;
    f: float;
    ...
f := einPunkt.X;
einPunkt.Y := f;
einPunkt.Z := einPunkt.Y;
```

An diesem Programmfragment fallen verschiedene Besonderheiten auf:

- *Empfängerobjekt*
  Eine typassoziierte Operation[5] hat immer ein sogenanntes *Empfängerobjekt*, auf dem sie angewendet wird. Empfängerobjekte müssen—natürlich—immer dem Typ angehören, innerhalb dessen die Operation definiert wurde. In unserem Beispiel ist das Objekt, auf das die Variable *einPunkt* verweist, das Empfängerobjekt der Invokationen X, Y und Z.

- *„Dot"-Notation*
  In GOM werden alle typassoziierten Operationen durch die sogenannte „*Dot*"-Notation aufgerufen, wobei die Aufrufe durchaus auch aneinandergereiht werden können. Zum Beispiel ist folgende Sequenz—unter der Annahme, daß entsprechende Typen und Operationen definiert sind—möglich:

  ```
  var einePerson: Person;
  ...
  einePerson.EhePartner.Vater.Mutter.Alter
  ```

  In diesem Programmfragment werden nacheinander die VTOs *Ehepartner*→, *Vater*→, *Mutter*→ und *Alter*→ aufgerufen. Dabei wird das *Empfängerobjekt* der Invokation jeweils durch das Resultat der vorhergehenden VTO-Invokation bestimmt.

- *VCO-Invokation*
  Die Invokation einer VCO-Operation sieht syntaktisch genauso aus wie eine Zuweisung in Programmiersprachen wie Pascal oder Algol. Zum Beispiel enthält der Aufruf

  ```
  einPunkt.Z := einPunkt.Y;
  ```

  die Invokation einer VCO, nämlich *Z*← und einer VTO, nämlich *Y*→. Da beide Operationen gleich benannt sind, muß der Compiler aus der Stellung des Aufrufs relativ zum „:="-Zeichen ableiten, ob es sich um die VTO oder die VCO-Operation gleichen Namens handelt.

[5] Es gibt in GOM auch sogenannte *freie* Operationen, die keinem Typ zugeordnet sind.

6.1.2 Der Konstruktor *create*

Auf jedem (nicht virtuellen) Typ ist implizit eine Operation zur Erzeugung von Objekten dieses Typs definiert. Diese sogenannten Konstruktoren können, wie die folgenden Beispiele zeigen, auf verschiedene Weise aufgerufen werden.

```
        var v: Vertex;
            ...
(1)     v := Vertex$create;
(2)     v.create;
```

Während bei dem ersten Aufruf der Typ des zu erzeugenden Objekts noch einmal explizit angegeben wird, erfolgt seine Bestimmung im zweiten Fall implizit über den Typ der Variablen. Beim Aufruf (2) erfolgt auch die Zuweisung der neu erzeugten Instanz an die Variable v implizit.

Variablen kann man mit Hilfe des Konstruktors auch schon bei ihrer Deklaration eine neue Instanz des betreffenden Typs zuordnen.

```
        var v: Vertex := Vertex$create;
```

Bei dieser Verwendungsart des Konstruktors zeigt die Variable v bei ihrer Deklaration auf ein neu erzeugtes Objekt vom Typ *Vertex*. Natürlich kann der Konstruktor auch im Rumpf von Operationen aufgerufen werden.

Wie im nächsten Abschnitt erläutert wird, können Konstruktoren parametrisiert sein, um das zu erzeugende Objekt auf benutzerdefinierte Art initialisieren zu können. Der eingebaute Konstruktor *create* ist immer sichtbar und muß deshalb nicht in der **public**-Klausel angegeben werden.

6.2 Benutzerdefinierte Operationen

Neben diesen vordefinierten Operationen kann man den GOM-Objekttypen semantisch reichere, anwendungsspezifische Operationen zuordnen. Wir wollen dies an unserem Beispiel-Typ *Vertex* demonstrieren, der folgendermaßen vervollständigt wird:

```
type Vertex supertype ANY is
    public X→, Y→, Z→, translate, scale, rotate, distance
    body [X: float;
          Y: float;
          Z: float;]
    operations
        declare Vertex: float, float, float → void;
        overload translate: float, float, float → void
            code translateFloatCode;
        overload translate: Vertex → void
            code translateVertexCode;
        declare scale: Vertex → void;
        declare rotate: float, char → void;     !! Rotations-Winkel und -Achse
        declare distance: Vertex → float;
    implementation
        define Vertex(x,y,z) is
            begin
                self.X:= x;
                self.Y:= y;
                self.Z:= z;
            end define Vertex;
        define scale(s) is
            begin
                self.X := self.X * s.X;
```

```
            self.Y := self.Y * s.Y;
            self.Z := self.Z * s.Z;
        end define scale;
    define translateFloatCode(t_X, t_Y, t_Z) is
        begin
            self.X := self.X + t_X;
            self.Y := self.Y + t_Y;
            self.Z := self.Z + t_Z;
        end define translateFloatCode;
    define translateVertexCode(t) is
        begin
            self.X := self.X + t.X;
            self.Y := self.Y + t.Y;
            self.Z := self.Z + t.Z;
        end define translateVertexCode;
    define rotateCode(Angle, Axis)
        ...
    define distance(OtherVertex) is
        var dx, dy, dz: float;
        begin
            dx := self.X-OtherVertex.X;
            dy := self.Y-OtherVertex.Y;
            dz := self.Z-OtherVertex.Z;
            return sqrt(dx*dx+dy*dy+dz*dz);
        end define distance;
end type Vertex;
```

Bei allen hier gezeigten Operationsdeklarationen handelt es sich um sogenannte *innere* Deklarationen, d.h. die Deklarationen stehen innerhalb des Typdefinitionsrahmens des Typs, dem sie zugeordnet sind. Man kann deshalb das "||" Zeichen und den Typ des Empfängerobjekts weglassen, da dies implizit aus dem Kontext hervorgeht. Die (komplettierte) Typdefinition *Vertex* ist größtenteils selbsterklärend. Die Operationen von *Vertex* lassen sich in die drei folgenden Klassen einteilen:

- *Konstruktoren* und *Initialisierer*: In Vertex ist eine Initialisierungsoperation *Vertex* deklariert. Solche Initialisierungsoperationen müssen immer den Namen des Typs, dessen Objekte damit initialisiert werden sollen, besitzen. Durch die Initialisierungsoperation können *Vertex*-Objekte in einem beliebigen Punkt des Koordinatensystems erzeugt werden. Auf einem Typ können mehrere Initialisierungsoperationen mit unterschiedlichen Parametern definiert werden. Um zu erläutern, wie solche Operationen verwendet werden können, betrachten wir wieder unsere *Vertex*-Variable v von oben:

  ```
  (1) v.create(0.5, 9.0, 7.5);
  (2) v.Vertex(3.5, 4.0, 9.9);
  ```

  Die erste Anweisung erzeugt ein *Vertex*-Objekt an den angegebenen Koordinaten, während die zweite eine Reinitialisierung eines existierenden Objekts vornimmt. Hierbei wird kein neues Objekt erzeugt. Um jedoch eine explizite Reinitialisierung vornehmen zu können, muß der entsprechende Initialisierer, hier *Vertex* nach außen sichtbar gemacht werden, was in unserem Beispiel nicht der Fall ist. Der Aufruf (2) wäre also illegal, weil die Operation *Vertex* nicht in der **public**-Klausel der Typdefinition enthalten ist.

- *Observierer*: in *Vertex* werden vier Leseoperationen, die keinerlei Zustandsänderungen verursachen, angeboten:

  - die VTO-Operationen $X\rightarrow$, $Y\rightarrow$ bzw. $Z\rightarrow$ lesen den aktuellen Wert des jeweiligen Attributs X, Y bzw. Z.

– *distance* hat—neben dem Empfängerobjekt noch—einen Parameter vom Typ *Vertex* und ermittelt den Abstand zwischen dem Empfänger-*Vertex* und dem Argument-*Vertex*.

- *Mutatoren*: in *Vertex* gibt es die Standardoperationen *translate*, *rotate* und *scale* zur geometrischen Transformation. Diese Operationen bezeichnet man als *Mutatoren*, weil sie den internen Zustand des Objekts, auf das sie angewendet werden, verändern.

Man kann sich Objekte vom Typ *Vertex* nun so vorstellen, wie wir dies in Abbildung 3 skizziert haben. Objekte (Instanzen) des Typs *Vertex* sind verkapselt gegen „direkte" Zugriffe von außen. Für Klienten des Typs steht eine wohldefinierte Schnittstelle bestehend aus den in der **public**-Klausel spezifizierten Operationen zur Verfügung. Zum Beispiel ist das direkte Setzen von Koordinaten nicht möglich, da die—vordefinierten—VCO-Operationen $X\leftarrow$, $Y\leftarrow$ und $Z\leftarrow$ nicht sichtbar gemacht wurden. Eine Zustandsänderung einer *Vertex*-Instanz ist nur über die Operationen *rotate*, *scale* und *translate* möglich.

Will man sowohl lesenden als auch schreibenden Zugriff auf Attribute von außen erlauben, so kann man die entsprechenden VTO und VCO in der **public**-Klausel zusammenfassen: $X\rightarrow$ *und* $X\leftarrow$ in der **public**-Klausel ist äquivalent zu X.

Will man alle Operationen, die implizit oder explizit innerhalb des Typs definiert wurden, sichtbar machen, kann dies durch vollständiges Weglassen der **public**-Klausel geschehen. Alle Operationen sind dann default-mäßig sichtbar. Wenn man bei tupelstrukturierten Typen nur die strukturelle Beschreibung angibt, zum Beispiel:

```
type Material is
    [Name: string;
     SpecWeight: float;]
```

kann man auf Objekten des Typs *Material* gerade die vordefinierten Operationen zum Lesen (VTO) und Schreiben (VCO) der Attribute anwenden. Unter der Annahme, daß eine Variable m auf eine *Material*-Instanz verweist, könnte man also folgende Operationen ausführen:

```
m.SpecWeight := 7.55;      !! VCO SpecWeight←
m.Name := "Iron";          !! VCO Name←
...
m.SpecWeight;              !! VTO SpecWeight→
m.Name;                    !! VTO Name→
```

In dieser Hinsicht bietet ein solcher „nackter" Objekttyp gerade die aus herkömmlichen Programmiersprachen bekannte Funktionalität von *record*-Typen.

6.3 Overloading

In der obigen Typdefinition *Vertex* fallen noch zwei durch **overload** gekennzeichnete Operationen auf. Grundsätzlich gilt in GOM, daß jeder Objekttyp einen eigenen Namensraum für die Definition von Operationen zur Verfügung stellt. Innerhalb dieses Namensraums müssen dann aber in der Regel die Operationen eindeutig benannt sein. Eine Ausnahme bilden die Operationen, die bei ihrer Deklaration durch Voranstellung des Schlüsselworts **overload** als überladen gekennzeichnet werden, wie in unserem Beispiel die Operation *translate*, für die es zwei Varianten gibt:

1. **overload** translate: float, float, float → **void code** translateFloatCode;
   In dieser Variante verlangt *translate* drei Fließkommazahlen als Parameter.

2. **overload** translate: Vertex → **void code** translateVertexCode;
   In dieser Version erwartet die Operation *translate* eine *Vertex*-Instanz als Argument.

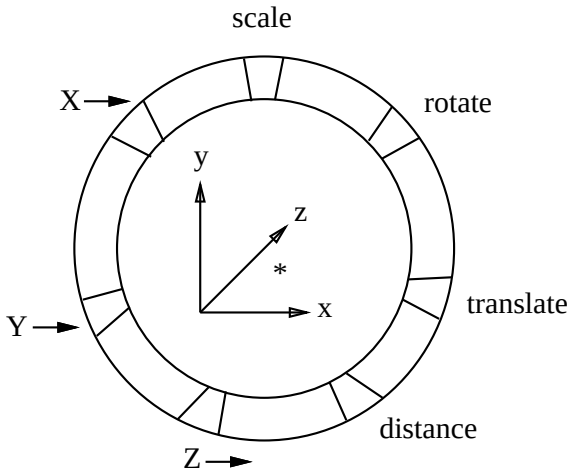

Abbildung 3: Grafische Darstellung der Verkapselung der *Vertex*-Objekte

Welche dieser beiden Versionen „gemeint“ ist, entscheidet der Compiler anhand des Kontextes, in dem die Invokation stattfindet. In unserem Beispiel läßt sich dies leicht aus der Anzahl der Parameter im Operationsaufruf ableiten.

Grundsätzlich müssen sich je zwei überladene Operationen mindestens in einem der beiden nachfolgend aufgeführten Kriterien unterscheiden:

- in der Anzahl der Parameter
- im Typ der Parameter, wobei bei Unter-/Ober-Typen besondere Bedingungen gelten.

Wie der aufmerksame Leser bei Betrachtung der bisher angeführten Beispiele bemerkt haben wird, haben Deklaration und Definition von typassoziierten Operationen üblicherweise den gleichen Namen, beispielsweise „**declare** scale“ und „**define** scale“. Der Implementierer kann jedoch für die Implementierung einer Funktion einen von der Deklaration abweichenden Namen spezifizieren. Im Falle von überladenen Operationen wird aus dieser Option ein Zwang. Die Zuordnung von Deklaration zu Definition einer Operation erfolgt dann über die **code**-Klausel, im Beispiel an den beiden *translate*-Varianten illustriert. Ist keine **code**-Klausel angegeben, erfolgt die Zuordnung über die Namensgleichheit von Deklaration und Definition der Operation.[6]

6.4 Äußere Operationen

Es ist oftmals nützlich, einem existierenden Typ nachträglich noch Operationen zuzuordnen. Dabei wäre es umständlich, den Typdefinitionsrahmen zu verwenden, den man dazu erst aus dem Schema holen müßte, um ihn dann zu erweitern. Deshalb ist es in GOM möglich, Operationen außerhalb des Typdefinitionsrahmens zu spezifizieren. Die so definierten Operationen werden *äußere Operationen* genannt.

Wir wollen jetzt äußere Operationen an unserem oben eingeführten Beispieltyp *Vertex* illustrieren:

```
declare inOrigin: Vertex || → bool
   code VertexInOriginCode;

define VertexInOriginCode is
```

[6] Namen von Operationsdefinitionen müssen aus implementierungstechnischen Gründen in der vorliegenden Version programmweit eindeutig sein.

```
return ( self.X >= -0.0005 and self.X <= 0.0005 and
         self.Y >= -0.0005 and self.Y <= 0.0005 and
         self.Z >= -0.0005 and self.Z <= 0.0005 );
```

Die oben eingeführte Operation *inOrigin* ist dem (existierenden) Objekttyp *Vertex* zugeordnet, was durch die Klausel „*Vertex* ||“ ausgedrückt wird, d.h. der Typ des Empfängerobjekts einer Invokation wird durch „||“ von den übrigen Argumenttypen (falls es noch weitere gibt) abgegrenzt.

Bei der Implementierung der äußeren Operationen darf nur auf die in der **public**-Klausel des Typs nach außen sichtbaren Operationen zurückgegriffen werden. Dadurch wird sichergestellt, daß nicht durch nachträglich hinzugefügte Operationen der Objekttyp ein inkonsistentes Verhaltensmuster bekommt. Diese muß in den Typdefinitionsrahmen von *Vertex* aufgenommen werden.

7 Kollektions-Typen

GOM unterstützt derzeit zwei Arten von Kollektionen:

1. *Mengen*, deren strukturelle Repräsentation in der **body**-Klausel als $\{T\}$ notiert wird
2. *Listen*, die als $< T >$ in der **body**-Klausel des Typdefinitionsrahmens definiert werden.

Mengenwertige Typen sind in GOM so definiert, daß die Elemente der Menge dem spezifizierten Elementtyp T oder einem Untertyp von T angehören müssen. Falls T eine Sorte ist, enthält die Menge die Werte der atomaren Sorte T; für einen komplexen Objekttyp T enthält die Menge Referenzen auf Objekte des Typs T. In jedem Fall handelt es sich um eine „echte“ Menge, so daß keine Mehrfachelemente erlaubt sind.

Listen hingegen erlauben dieses mehrfache Vorkommen des gleichen Elements an unterschiedlichen Positionen. Elemente einer Liste können—anders als in Mengen—über ihre Position referenziert werden, wobei die bei Reihungen übliche [ ]-Notation† verwendet werden kann.[7]

Ein Beispiel für einen mengenwertigen Objekttyp ist nachfolgend definiert. Dieser Typ *VertexSet* kann benutzt werden, um Mengen von *Vertex*-Instanzen zu unterhalten.

```
type VertexSet is
    public *
    body { Vertex }
    operations
        declare cardinality: → int;
        ...
    implementation
        define cardinality is
            var v: Vertex;
                number: int := 0;
            begin
                foreach (v in self)
                    number := number + 1;
                return number
            end; !! cardinality !!
        ...
end type VertexSet;
```

Objekte des Typs *VertexSet* enthalten (Referenzen auf) Instanzen des Typs *Vertex* oder Instanzen eines Untertyps von *Vertex*, sofern ein solcher definiert ist.

[7] Die [ ]-Notation ist noch nicht implementiert. Stattdessen steht die Zugriffsfunktion *nth()* für jeden Listentyp zur Verfügung, die wir auch in den folgenden Beispielen verwenden werden.

8 Objekte, Variablen und Werte

8.1 Unterscheidung zwischen Werten und Objekten

Es ist wichtig, daß man eine klare Unterscheidung zwischen *Werten* und *Objekten* trifft. Ein Wert, wie z.B. der *int*-Wert 59, ist ein *nicht-mutierbares*, d.h. nicht-änderbares Datum. In diesem Sinne sind die Elemente der Sorten, wie *int*, *bool*, *float*, *char*, etc. fest vorgegebene Einheiten in unserem System, die nicht verändert oder vernichtet werden können. Im Gegensatz dazu sind Objekte zwar *persistent*, aber potentiell veränderbar.

Betrachten wir als Beispiel das Objekt vom Typ *Person*, das die Person namens „Mickey Mouse“ repräsentiert. Dieses Objekt hat neben anderen Attributen auch ein Attribut *Alter*, das auf *int*-Werte eingeschränkt ist und den aktuellen Wert 59 hat. An Mickey Mouse's Geburtstag wird dieses Attribut auf 60 gesetzt. Man beachte aber, daß der *int*-Wert 59 sich nicht geändert hat; vielmehr wurde dieser feste Wert 59 durch einen anderen festen Wert 60 ersetzt. Andererseits hat sich das Objekt, das die Person „Mickey Mouse“ repräsentiert, tatsächlich geändert. Es ist eines der wesentlichen Grundkonzepte des objektorientierten Paradigmas, daß sich durch diese „Mutation“ zwar der Objektzustand geändert hat; die Identität des Objekts aber erhalten geblieben ist.

8.2 Instantiierung von Objekten

Es wurde bereits angedeutet, daß man Objekte durch Instantiierung von Typen erzeugt. Für die Instantiierung verwendet man die implizit jedem Typ zugeordnete Operation *create*. Wir werden jetzt an einem Beispiel die Erzeugung einer sehr kleinen (und wenig sinnvollen) Datenbank illustrieren:

```
     persistent var meinePunkte: VertexSet;
                    einPunkt: Vertex;
                    ...
(1)  meinePunkte.create;     !! dieses Objekt wird zur leeren Menge initialisiert
(2)  einPunkt.create(0.0, 0.0, 0.0);
(3)  meinePunkte.insert(einPunkt);
(4)  einPunkt.create(9.0, 5.0, 13.0);
(5)  meinePunkte.insert(einPunkt);
(6)  einPunkt.create(9.0, 0.0, 0.0);
(7)  meinePunkte.insert(einPunkt);
```

In diesem Programmfragment wurden zwei Variablen deklariert:

- *meinePunkte* vom Typ *VertexSet* und
- *einPunkt* vom Typ *Vertex*

Insgesamt wurden in dem kurzen Programm vier Objekte durch Instantiierung erzeugt: ein Objekt vom Typ *VertexSet*, auf das *meinePunkte* verweist, und drei Objekte vom Typ *Vertex*. Die aus diesem Programm resultierende Objektbank-Ausprägung ist in Abbildung 4 gezeigt.

8.3 Objektidentität

Jedes Objekt erhält bei seiner Instantiierung eine zugeordnete *Objektidentität*, genannt OID, die sich während seiner Lebenszeit nicht ändert. Die Objektidentität ist vom Speicherort des Objekts und vom internen Zustand der Instanz unabhängig. Die Objektidentität der Objekte wird in unserer Darstellung (abstrakt) mit $id_1, id_2, id_3, \ldots$ bezeichnet. Der OID eines Objekts wird systemintern benutzt, um das Objekt eindeutig zu referenzieren. Zum Beispiel werden in

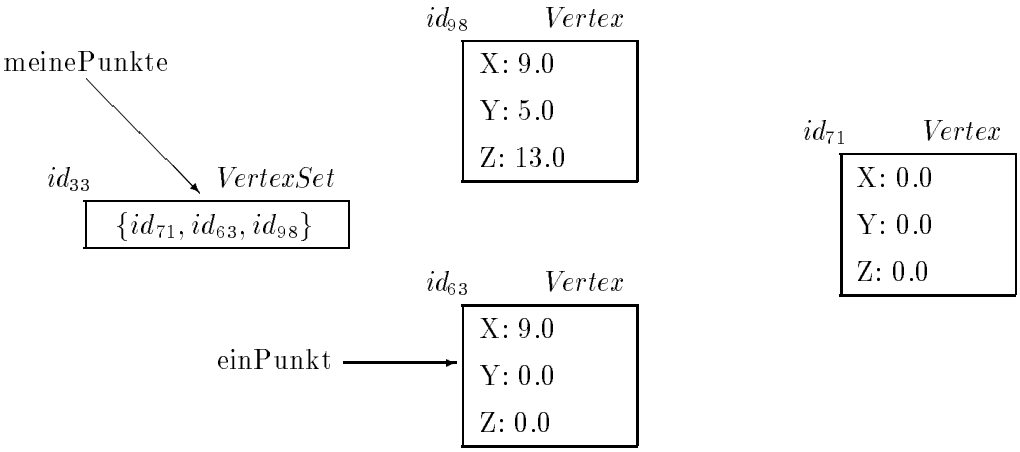

Abbildung 4: Beispielausprägung einer *VertexSet*-Instanz

der in Abbildung 4 gezeigten Objektbank die *Vertex*-Objekte mit den OIDs id_{98}, id_{71} und id_{63} über diese Objektidentitäten von dem *VertexSet*-Objekt id_{33} referenziert.

So erstaunlich das für den unerfahrenen GOM-Programmierer auch klingen mag: die Objektidentität ist dem Benutzer nicht sichtbar. Als Benutzer kann man also nur über entsprechend deklarierte Variablen auf die Objekte zugreifen. In unserer Beispiel-Objektbank in Abbildung 4 kann man über die Variable *meinePunkte* auf das Objekt (mit dem OID) id_{33} zugreifen, und mit der Variablen *einPunkt* kann man auf das *Vertex*-Objekt id_{63} zugreifen. Andererseits sind die *Vertex*-Instanzen id_{98} und id_{71} nur über die Anwendung entsprechender Lese-Operationen auf das Mengen-Objekt id_{33} zugreifbar.

8.4 Variablen

Man kann in GOM beliebig viele Variablen benutzen. Die Bedingungen für die Benutzung einer Variablen sind:

1. Die neu zu benutzende Variable muß entsprechend deklariert worden sein.
2. Die Variable muß bei der Deklaration auf einen Typ eingeschränkt werden. Im Laufe ihrer Lebenszeit darf die Variable nur Werte oder Objekte des Typs oder eines Untertyps annehmen, auf den sie eingeschränkt wurde.
3. Es darf keine Namenskonflikte (Namensgleichheit) mit anderen, schon existierenden Variablen geben.

Variablen können bei ihrer Definition initialisiert werden. Bei Variablen, die auf Sorten eingeschränkt sind, geschieht dies durch Zuweisung eines Werts; Variablen eines Objekttyps können durch Erzeugung oder Zuweisung eines Objekts initialisiert werden. Beispiel:

```
var f: float := 0.0;
    v: Vertex := Vertex$create(9.0, 3.5, 0.5);
    v1: Vertex := v;
```

9 Persistenz

Unter Persistenz versteht man das Überleben von Programmkomponenten über die Ausführungszeit des Programmes hinweg. Dazu müssen diese Komponenten natürlich dauerhaft auf dem Hintergrundspeicher abgespeichert werden.

In GOM unterscheidet man zwei unterschiedliche Komponenten, die potentiell persistent sein können:

- Objekte
- Variablen

Wir wollen nacheinander die Initiierung und Konsequenzen der Persistenz untersuchen.

9.1 Persistenz von Objekten

Jedes GOM-Objekt „versteht“ den Operationsaufruf *persistent*. Erst nach Aufruf der *persistent*-Operation auf einem Objekt wird dieses dauerhaft in der Datenbank gespeichert. Beispiele sind:

```
einPunkt.persistent;
meinePunkte.persistent;
```

Um die Persistenz aller Instanzen eines Typs zu garantieren, sollte man diese Invokation der Operation *persistent* schon in der Initialisierung durchführen.

Nich alle Objekte, die von einem persistenten Objekt aus referenziert werden, müssen persistent sein. Beispielsweise können in die oben angeführte persistente Punktmenge *meinePunkte* auch transiente *Vertex*-Instanzen eingefügt werden. Die Referenzierung transienter Objekte von persistenten Objekten aus führt bei Beendigung des Programms, in dessen Verlauf die Referenz etabliert wurde, zu sogenannten *dangling references*. Das bedeutet, daß in der Datenbasis Objekte referenziert werden, die gar nicht mehr existieren, da die Lebensdauer transienter Objekte auf einen Programmlauf begrenzt ist. Gute GOM-Programmierer zeichnen sich unter anderem dadurch aus, daß sie Verweise persistenter Objekte auf transiente Objekte vermeiden.

9.2 Persistenz von Variablen

Man kann in einem GOM-Programm Variablen „persistent machen“, indem man bei der Deklaration das Schlüsselwort **persistent** voranstellt.

Wird ein Programm, in dem persistente Variablen definiert sind, beendet und zu einem späteren Zeitpunkt neu gestartet, so besitzt die Variable den letzten, im vorherigen Programmlauf zugewiesenen Wert. Persistente Variablen können auch von unterschiedlichen Programmen aus benutzt werden.

Beispiele für persistente Variablen sind:

```
persistent var meinePunkte: Vertex;
               einPunkt: Vertex;
```

Die Deklaration einer Variablen als persistent erzwingt nicht, daß das jeweilige referenzierte Objekt ebenfalls persistent sein muß. Verweist eine persistente Variable am Ende eines Programmlaufs auf ein transientes Objekt, so entsteht eine sogenannte *dangling reference*,da das transiente Objekt am Ende des Programms gelöscht wird.

10 Vererbung

Derzeit unterstützt GOM *singuläre (einfache) Vererbung* entlang der Ober-/Untertyp-Hierarchie. Das heißt, daß der Untertyp alle Eigenschaften—Attribute und Operationen—des Obertyps erbt.

Wir wollen Vererbung anhand der in Abbildung 5 dargestellten Typhierarchie erläutern. Der Typ *GeometricPrimitive* ist dabei als direkter Untertyp der gemeinsamen Wurzel *ANY* definiert. *ANY* ist ein vordefinierter Typ, der automatisch Obertyp aller Typen ist. Wird bei der Definition eines Typs kein Obertyp explizit in der optionalen **supertype**-Klausel angegeben, so

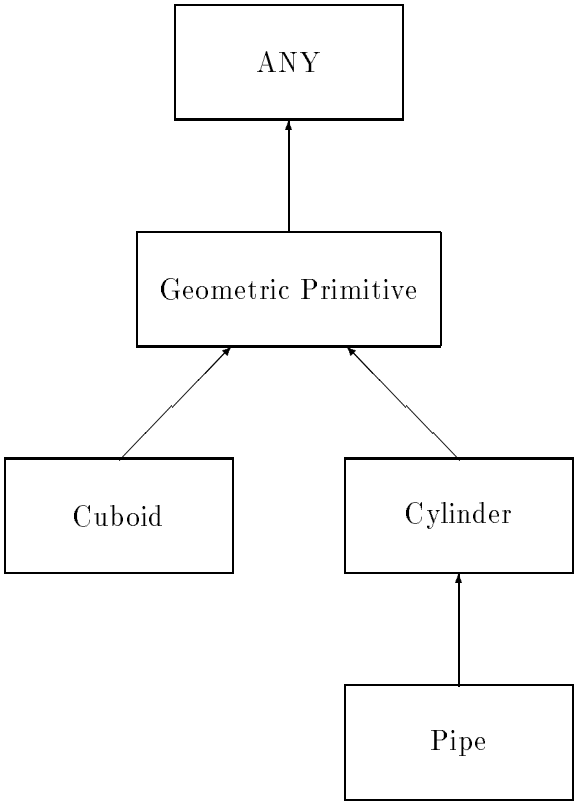

Abbildung 5: Eine exemplarische Typhierarchie geometrischer Objektklassen

wird implizit *ANY* als direkter Obertyp dieses Typs angenommen. *GeometricPrimitive* ist weiter verfeinert zu den Typen *Cuboid* und *Cylinder*, wobei *Cylinder* selbst wieder einen Untertyp *Pipe* besitzt. Der Typdefinitionsrahmen für *GeometricPrimitive* ist wie folgt definiert:

```
type GeometricPrimitive supertype ANY is
    public paint, SpecWeight, GeoID, Color→
    body [GeoID: string;
          Color: string;
          Mat: Material;]
    operations
        declare SpecWeight: → float;
        declare paint: string → void;
    implementation
        define paint(c)
            self.Color := c;
        define SpecWeight
            return self.Mat.SpecWeight;
end type GeometricPrimitive;
```

Der Objekttyp *GeometricPrimitive* ist—als direkter Untertyp des vordefinierten Typs *ANY*—als tupelstrukturierter Typ definiert. Er besitzt drei Attribute

- *GeoID* vom Typ *string*[8], ein vom Benutzer gewählter Identifikator des geometrischen Objekts. Dieses Attribut ist nicht mit dem Objektidentifikator, der jedem komplexen GOM-Objekt zugeordnet ist, zu verwechseln.

[8] string ist ein vordefinierter Objekttyp, dessen Struktur eine Liste von Zeichen ist und auf dem etliche Operationen vordefiniert sind.

```
type Cylinder supertype GeometricPrimitive is
    public Radius→, weight, volume, translate, rotate, shrink
    body [Radius: float;
          Center1: Vertex;
          Center2: Vertex;]
    operations
        declare volume: → float code CylinderVolumeCode;
        declare weight: → float;
        declare translate: Vertex → void;
        declare Cylinder: string, string, Material, float, Vertex, Vertex → void;
        declare length: → float;
        ...
    implementation
        define Cylinder(g, c, m, r, c1, c2)
            begin     !! Generiere den „Einheits“-Zylinder !!
                self.GeoID := g;
                self.Color := c;
                self.Material := m;
                self.Radius := r;
                self.Center1 := c1;
                self.Center2 := c2;
                return self;
            end define Cylinder;
        define length is
            return self.Center1.distance(self.Center2);
        define CylinderVolumeCode is
            return (self.Radius * self.Radius * 3.14 * self.length);
        define weight is
            return self.volume * self.SpecWeight;
        define translate(t)
            begin
                Center1.translate(t);   !! translate wird an die Vertex-Instanz Center1 delegiert!!
                Center2.translate(t);
            end define translate;
end type Cylinder;
```

Abbildung 6: Die Definition des Objekttyps *Cylinder*

- *Color* ist ebenfalls vom Typ *string*.
- *Material* ist ein objektwertiges Attribut, dem ein Objekt vom Typ *Material* zugeordnet werden kann.

Auf der Basis dieser Typdefinition können wir jetzt den Untertyp *Cylinder* definieren. Der Typdefinitionsrahmen ist in Abbildung 6 gezeigt.

Der Objekttyp *Cylinder* ist als direkter Untertyp von *GeometricPrimitive* definiert. Zusätzlich zu den im Typ *GeometricPrimitive* „öffentlich“ gemachten Operationen bietet der *Cylinder*-Typ noch die Operationen *Radius→, length, weigth, volume*, sowie die geometrischen Transformationen *rotate, translate* und *shrink*. Die letztgenannte Operation dient dazu, den *Radius* einer *Cylinder* Instanz zu verändern, d.h. zu schrumpfen bzw. auszudehnen.

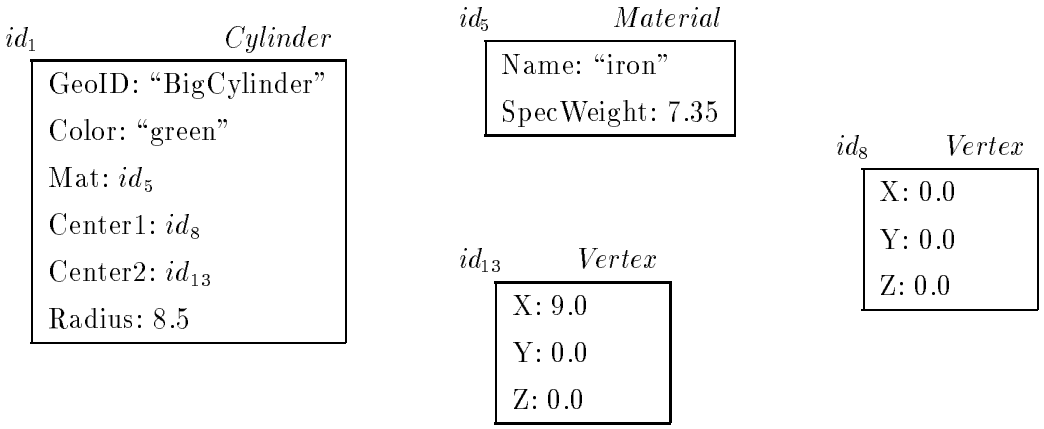

Abbildung 7: Beispielausprägung einer *Cylinder*-Instanz

11 Verfeinerung von Operationen und dynamisches Binden

In unserer Typhierarchie in Abbildung 5 ist *Pipe* zur Modellierung von Röhren als weiterer Objekttyp enthalten. Dieser Objekttyp unterscheidet sich von *Cylinder* dadurch, daß er über ein weiteres Attribut *InnerRadius* vom Typ *float* verfügt. Die Struktur einer *Pipe*-Instanz ist also beschrieben durch die acht Attribute

- *GeoID* und *Color* vom Typ *string*
- das Attribut *Mat* vom Typ *Material*
- *Center1* und *Center2*, beide vom Typ *Vertex*
- *Radius* und *Length*, jeweils von der Sorte *float*, und
- *InnerRadius*, auch vom Typ *float*

Die ersten drei Attribute wurden vom indirekten Obertyp *GeometricPrimitive* geerbt, die nächsten vier vom direkten Obertyp *Cylinder*, und nur das letzte Attribut *InnerRadius* ist direkt in *Pipe* definiert worden. Die Typdefinition sieht dann folgendermaßen aus:

```
type Pipe supertype Cylinder is
    public InnerRadius, connect
    body
        [InnerRadius: float;]
    operations
        declare connect: Pipe → Pipe;
        refine volume: → float code PipeVolumeCode;
        refine weight: → float code PipeWeightCode;
    implementation
        define PipeVolumeCode is
            return (super.volume −
                    self.InnerRadius ∗ self.InnerRadius ∗ 3.14 ∗ self.Length);
        define PipeWeightCode is
            return self.volume ∗ self.SpecWeight;
        define connect(otherPipe) is
            ...
end type Pipe;
```

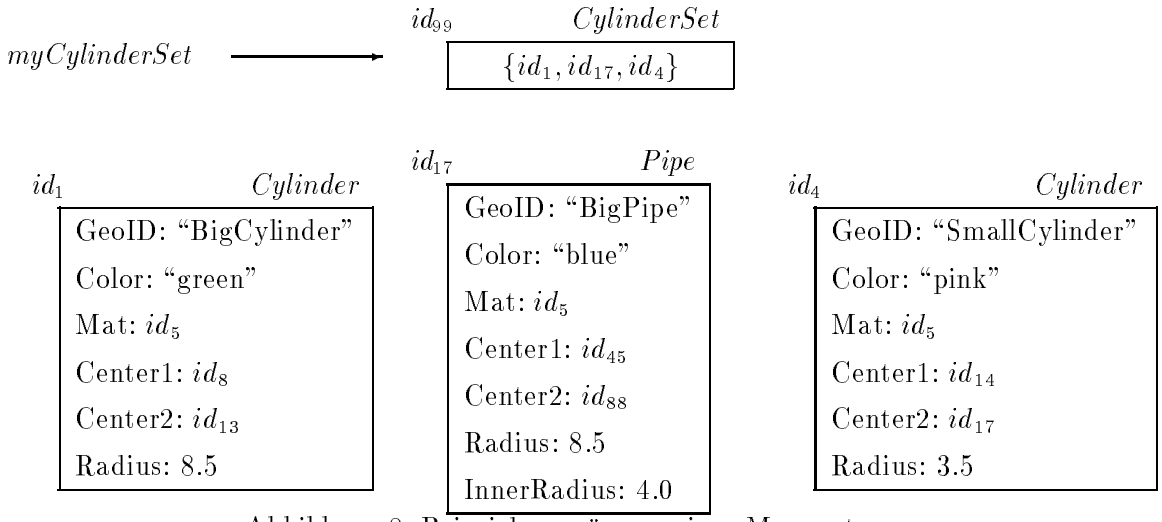

Abbildung 8: Beispielausprägung eines Mengentyps

Das bemerkenswerteste an der obigen Typdefinition *Pipe* sind die Verfeinerungen der Operationen *volume* und *weight*. Die Operation *volume* ist jetzt so implementiert, daß das Hohlraumvolumen einer Röhre von dem Gesamtvolumen abgezogen wird. Interessanterweise wurde bei der Neu-Implementierung die von *Cylinder* geerbte Operation *volume* verwendet, die in dem Ausdruck **super**.*volume* invokiert wird. Das Schlüsselwort **super** besagt, daß die im Obertyp existierende Operation gebunden wird. Somit ist die obige Implementierung äquivalent zu

```
define PipeVolumeCode is
    return (self.Radius * self.Radius * 3.14 * self.length −
                self.InnerRadius * self.InnerRadius * 3.14 * self.length);
```

Alle verfeinerten Operationen werden in GOM dynamisch—also zur Laufzeit—gebunden. Die Notwendigkeit, verfeinerte Operationen dynamisch zu binden, soll an der Beispielausprägung einer Objektbank in Abbildung 8 erläutert werden. In diesem Beispiel gibt es eine Variable *MyCylinderSet*, die auf die *CylinderSet* Instanz id_{99} verweist. Diese Menge enthält (Verweise auf) drei Objekte:

- 2 direkte *Cylinder*-Instanzen mit den OIDs id_1 und id_4
- eine direkte *Pipe*-Instanz mit dem OID id_{17}

Da *Pipe* ein Untertyp von *Cylinder* ist, ist es völlig legitim, daß ein *CylinderSet* auch eine (oder mehrere) *Pipe*-Instanz(en) enthält. Man bezeichnet dies im Zusammenhang objektorientierter Modelle als *Substituierbarkeit*: eine Untertyp-Instanz ist überall dort legal einsetzbar, wo eine Obertyp-Instanz gefordert ist.

Jetzt betrachten wir folgendes Programmfragment:

```
var c: Cylinder;
    TotalVolume: float := 0.0;
    ...
foreach (c in myCylinderSet)
    TotalVolume := TotalVolume + c.volume;
```

Würde man in diesem Programmfragment die Operation *volume* statisch (zur Übersetzungszeit) binden, so würde bei jedem Schleifendurchlauf die gleiche „Version“ von *volume* ausgeführt, und zwar die *CylinderVolumeCode*-Version. In dem Durchlauf der **foreach**-Schleife, in dem *c* an die *Pipe*-Instanz id_{17} gebunden wird, muß aber die *PipeVolumeCode*-Version gebunden werden. Genau dies wird in GOM durch *dynamisches Binden* sichergestellt. Verfeinerte Operationen werden

zur Laufzeit entsprechend dem *direkten* Typ des Empfängerobjekts gebunden. Einmal verfeinerte Operationen können durchaus nochmals verfeinert werden, so daß man im allgemeinen eine Verfeinerungshierarchie erhält. Durch dynamisches Binden wird garantiert, daß ausgehend vom Typ des Empfängerobjekts immer die „spezifischste“ Version einer verfeinerten Operation zur Ausführung kommt. Dies geschieht ganz einfach dadurch, daß zur Laufzeit der direkte Typ des Empfängerobjekts bestimmt wird und dann innerhalb der Typhierarchie—beginnend beim Typ des Empfängerobjekts—in Richtung der Wurzel ANY nach einer Operation des gegebenen Namens gesucht wird. Dies könnte entweder eine Verfeinerung der Operation sein oder aber auch die Originalversion.

12 Polymorphe Operationen

Polymorphe Operationen sind Operationen, die nicht nur auf einem Typ, sondern auf einer Menge von Typen mit bestimmten Eigenschaften definiert sind. Eine Form des Polymorphismus ist die Vererbung von Operationen an alle Untertypen eines Typs: typ-assoziierte Operationen sind auf dem Typ selbst und auf allen seinen Untertypen definiert. In GOM ist noch eine weitere Form des Polymorphismus realisiert. Diese werden wir im folgenden kurz vorstellen.

Betrachten wir folgende Typdefinitionen von *Personen* und *Schwänen*:

```
type Person
   supertype Lebewesen is
   public heiraten
   body [EhePartner: Person;
         ...; ]
   operations
      declare heiraten: Person → void
         code PersonenHochzeit;
      ...
   implementation
      define PersonenHochzeit(Opfer) is
         self.EhePartner := Opfer;
      ...
end type Person;
```

```
type Schwan
   supertype Lebewesen is
   public heiraten
   body [EhePartner: Schwan;
         ...; ]
   operations
      declare heiraten: Schwan → void
         code SchwanenHochzeit;
      ...
   implementation
      define SchwanenHochzeit(Opfer) is
         self.EhePartner := Opfer;
      ...
end type Schwan;
```

Bei dieser Definition fällt auf, daß wir zwei Operatoren definieren müssen (*PersonenHochzeit*, *SchwanenHochzeit*), die fast identisch sind. Die Definition nur eines Operators *Hochzeit* ist mit den bisher vorgestellten Mitteln nicht möglich. Dies liegt daran, daß der Operator *Hochzeit* nur auf solche Instanzen angewandt werden darf, die einen Tupeltyp repräsentieren, der ein Attribut *EhePartner* besitzt, das wiederum vom gleichen Typ sein muß, wie die “zu verheiratende” Instanz selbst. Genau um diese Art von Bedingungen zu fassen, werden polymorphe Operationen eingeführt. Die obige Bedingung an einen Typ, dessen Instanzen als Argumente für den *Hochzeit*-Operator zugelassen sind, läßt sich wie folgt ausdrücken:

$$\backslash HoTy \leq [\text{EhePartner}{:}\backslash HoTy]$$

$\backslash HoTy$ stellt hier eine Typvariable dar. Die Typen, die diese Variable annehmen kann, muß bestimmten Eigenschaften genügen. Die geltende Einschränkung steht rechts des $\leq$-Zeichens. Diese besagt zunächst einmal, daß $\backslash HoTy$ einen Tupeltyp darstellen muß. Dies erkennt man an den umschließenden eckigen Klammern. Dieser Tupeltyp muß weiterhin ein Attribut *Ehepartner* besitzen, das wiederum vom Typ $\backslash HoTy$ ist. Die Typüberprüfung, die zum Zeitpunkt der Übersetzung eines GOM-Programmes aufgerufen wird, kann nun für die Typvariable $\backslash HoTy$ nur solche Typnamen einsetzen, die die spezifizierten Eigenschaften erfüllen. Dies heißt für unser Beispiel, daß für $\backslash HoTy$ der Typ *Person* oder *Schwan* eingesetzt werden kann, da beide resultierenden Ausdrücke

$Person \leq$ [EhePartner: $Person$]
$Schwan \leq$ [EhePartner: $Schwan$]

gültig sind. Der Typ *Zylinder* beispielsweise kann an dieser Stelle nicht für $\backslash HoTy$ eingesetzt werden, da er nicht über ein entsprechendes Attribut *EhePartner* verfügt.

Innerhalb einer Operationsdeklaration werden die den Typvariablen auferlegten Typgrenzen in runden Klammern eingeschlossen und durch ',' voneinander getrennt. Die Typgrenzen werden direkt nach dem Operationsnamen angegeben. Zusätzlich wird verlangt, daß das Schlüsselwort **polymorph** dem Operionsnamen vorangeht. Die polymorphe Version des *Hochzeit*-Operators lautet

polymorph declare Hochzeit($\backslash HoTy \leq$ [EhePartner : $\backslash HoTy$]) : $\backslash HoTy \parallel \backslash HoTy \rightarrow$ **void**
code PolymorpheHochzeit;

Wir wiederholen an dieser Stelle einige Anwendungen der polymorphen *Hochzeit*-Operation. Hinter den Anwendungen ist verzeichnet, welche Ausdrücke legal sind und welche nicht.

```
var Helmuth: Angestellter;
    Lothar, Sabine: Person;
    Swen, Swenja: Schwan;
...
(1)  Lothar.heiraten(Sabine);     !! okay
(2)  Helmuth.heiraten(Sabine);    !! okay
(3)  Sabine.heiraten(Helmuth);    !! okay
(4)  Sabine.heiraten(Lothar);     !! okay
(5)  Swen.heiraten(Swenja);       !! okay
(6)  Sabine.heiraten(Swen);       !! ILLEGAL
(7)  Swen.heiraten(Sabine);       !! ILLEGAL
(8)  Lothar.heiraten(Helmuth);    !! okay
```

Das Symbol $\leq$, das wir oben benutzt haben, um für Typvariablen eine Typgrenze anzugeben, mit Hilfe derer wir die Mindestanforderungen an die Struktur des einzusetzenden Typen formulierten, repräsentiert eine Ordnung auf der Menge der Typausdrücke.

13 Selektion von Objekten

Das Konzept der polymorphen Operationen ist so mächtig, daß man damit (fast) mühelos Sprachkonstrukte realisieren kann, für die man in anderen objektorientierten Sprachen eigenständige Operationen benötigt. Hierzu zählen insbesondere auch ausdrucksstarke Selektionsoperationen, die aus einer Menge von Objekten genau diejenigen ermitteln, die ein bestimmtes *Selektionsprädikat* erfüllen. In GOM kann man einen derartigen *select*-Operator als polymorphe Operation definieren, die als Parameter das *Selektionsprädikat*, modelliert als boole'sche Funktion, bekommt. Wir müssen allerdings nach der Anzahl der Parameter unterscheiden, die dieses Selektionsprädikat benötigt.

13.1 Selektionsprädikat ohne Parameter

Dies ist der einfachste Fall, da wir den *select*-Operation ganz einfach wie folgt formulieren können:

polymorph overload select ($\backslash t_1 \leq \{\backslash t_2\}$) : $\backslash t_1 \parallel (\backslash t_2 \parallel \rightarrow$ bool) $\rightarrow \backslash t_1$
code selectNoParam;

Diese Signatur besagt, daß *select* eine polymorphe überladene Operation—es werden nachfolgend noch weitere *select*-Operationen eingeführt—ist, die zwei Parameter hat:

- das Empfängerobjekt vom mengen-wertigen Typ $\backslash t_1$, wobei die Elemente der Menge vom Typ $\backslash t_2$ sind
- eine boole'sche Operation, die auf dem Typ $\backslash t_2$ definiert ist und die Signatur ($\backslash t_2 \parallel \rightarrow$ *bool*) haben muß.

Die Implementierung dieser *select*-Operation sieht wie folgt aus:

```
define selectNoParam(SelPred) is
    var result: \t1;
        candidate: \t2;
    begin
        result.create;     !! es wird die leere Ergebnismenge erzeugt
        foreach (candidate in self)
            if candidate.SelPred then result.insert(candidate);
        return result;
    end define selectNoParam;
```

In dieser Implementierung wird zunächst die Ergebnismenge, auf die *result* verweist, als leere Menge initialisiert. Danach „läuft" die Variable *candidate* durch die Menge, auf der die *select*-Operation angewendet wurde—also **self**—und ermittelt, ob das Selektionsprädikat *SelPred* erfüllt ist. Falls dies der Fall ist, wird *candidate* in *result* eingefügt. Die Menge *result* wird letztendlich übergeben.

Wir wollen jetzt ein kleines Anwendungsbeispiel für die *select*-Operation auf *Cuboid*-Mengen zeigen. Dazu definieren wir zunächst eine neue boole'sche Operation namens *inOrigin* auf dem Objekttyp *Cuboid*, die feststellt, ob einer der Eckpunkte des *Cuboids*, auf den die Operation angewendet wird, im Nullpunkt ($\pm\,\varepsilon$) des Koordinatensystems liegt:

```
declare inOrigin: Cuboid || → bool;

define inOrigin is     !! liegt einer der Eckpunkte im Ursprung?
    return ((self.V1.X = 0.0 ± ε and
             self.V1.Y = 0.0 ± ε and
             self.V1.Z = 0.0 ± ε) or
             ...
            (self.V8.X = 0.0 ± ε and
             self.V8.Y = 0.0 ± ε and
             self.V8.Z = 0.0 ± ε);
```

Aufbauend auf diesem Selektionsprädikat können wir nun sehr einfach die *select*-Operation wie folgt anwenden:

```
var meineCuboide, CuboideImUrsprung: CuboidSet;
    ...
CuboideImUrsprung := meineCuboide.select(inOrigin);
```

Nach Ausführung dieses Programmfragments enthält die Menge *CuboideImUrsprung* (Verweise auf) die *Cuboide*, die in der Menge *meineCuboide* enthalten sind und einen Eckpunkt im Ursprung haben.

Übrigens sollten wir an dieser Stelle anmerken, daß das Selektionsprädikat *inOrigin* nur auf *Cuboid*-Instanzen anwendbar ist, da es nicht-polymorph definiert wurde. Andererseits ist die *select*-Operation, wenn ihr ein geeignetes Selektionsprädikat übergeben wird, auf Mengen beliebigen Elementtyps anwendbar.

13.2 Selektionsprädikat mit Parametern

Wir wollen jetzt das Prinzip der parametrisierten Selektionsprädikate aufzeigen. In diesem Fall muß der Parameter des Selektionsprädikats mit an die *select*-Operation übergeben werden. Die Signatur und die Implementierung dieser polymorphen *select*-Operation sieht jetzt wie folgt aus:

```
polymorph overload select (\t1 ≤ {\t2}) : \t1 || (\t2 || \t3 → bool), \t3 → \t1
    code selectOneParam;

define selectOneParam(SelPred, p1)
    var result: \t1;
    var candidate: \t2;
    begin
        result.create;
        foreach (candidate in self)
            if candidate.SelPred(p1) then result.insert(candidate);
        return result;
    end define selectOneParam;
```

Es fällt auf, daß $\backslash t_3$ nicht in der Qualifikation der Typvariablen angegeben wurde. Dies bedeutet, daß über $\backslash t_3$ keine Annahmen vorausgesetzt werden, d.h. der Typ des Parameters des Selektionsprädikats kann beliebig spezifiziert werden. Implizit muß nur gelten: $\backslash t_3 \leq \mathrm{ANY}$.

Analog können nun Selektionsoperationen mit Selektionsprädikaten mit beliebig vielen Parametern definiert werden.

14 Iteratoren

Ein weiteres Sprachkonstrukt, das bei vielen Anwendungen, die große Datenmengen verarbeiten, sehr hilfreich ist, stellen die *Iteratoren* dar. Iteratoren werden auf Mengen von Objekten definiert und erlauben den sequentiellen Durchlauf durch die Menge. Hierzu gibt es in GOM die **foreach**-Schleife, bei der jedes Objekt der Menge genau einmal „besucht“ wird. Oft ist es aber sinnvoll, dem Iterator einen sogenannten „*Filter*“ vorzuschalten, so daß nur die Objekte bearbeitet werden, die diese Filterbedingung erfüllen. In GOM kann man dieses Konzept sehr einfach durch Kombination der **foreach**-Schleife mit der polymorphen *select*-Operation erzielen. Die *select*-Operation erfüllt hierbei die Filterfunktion. Anstatt dieses Konzept hier zu formalisieren, wollen wir uns auf die Illustration an einem konkreten Beispiel beschränken.

```
declare bigCyl: Cylinder || float → bool
    code bigCylCode;

define bigCylCode(Threshold)
    return self.volume ≥ Threshold;

...
var c: Cylinder;
    myCylinders: CylinderSet;
    BigCylindersTotalWeight: float := 0.0;
    ...
foreach (c in myCylinders.select([bigCyl: Cylinder || float → bool], 20.0))
    BigCylindersTotalWeight := BigCylindersTotalWeight + c.weight;
```

Im obigen Programmfragment wurde zunächst ein weiteres Selektionsprädikat namens *bigCyl* auf *Cuboid*-Objekten definiert. In diesem Prädikat wird überprüft, ob der *Cylinder*, auf den die boole'sche Operation angewendet wird, ein Mindest-Volumen der als Parameter übergebenen Größe *Threshold* hat. In der **foreach**-Schleife wird das Gewicht der *Cylinder* summiert, die mindestens ein (Individual-)Volumen von 20.0 aufweisen. Die *select*-Operation filtert also kleinere *Cylinder*-Instanzen der Menge *myCylinders* aus.

15 Generische Typen

Bei der Erstellung größerer Softwaresysteme kommt es oft vor, daß man ähnliche Datenstrukturen für unterschiedliche Datentypen erstellen muß. Es hat sich gezeigt, daß Vererbung und Subtypisierung für diesen Zweck wenig geeignet sind, da hierdurch nur Verfeinerungen von Strukturen und/oder Verhalten unterstützt wird. Zur Unterstützung ähnlicher, aber unterschiedlich typisierter Datenstrukturen bieten wir in GOM das Konzept der generischen Typen an. Wir wollen dies an einem Beispiel, das allgemein aus dem Bereich der abstrakten Datentypen bekannt sein dürfte, illustrieren: den (abstrakten) Datentyp *Stack*, den wir in unserer nachfolgenden Definition so parametrisieren, daß man aus dem generischen Typ *Stack* konkrete Typen für beliebige *Stack*-Elemente generieren kann.

```
generic type Stack (\elemType ≤ ANY) is
    public push, pop, card, empty
    body [elems: < \elemType >;
          card: int;]
    operations
        declare push: \elemType → void;
        declare pop: → \elemType;
        declare card: → int;
        declare empty: → bool;
        declare Stack: → void;
    implementation
        define push(elem) is
            begin
                elems.nth(card + 1) := elem;
                card := card + 1;
        end define push;
        define pop is
            begin
                card := card − 1;
                return elems.nth(card+1);
        end define pop;
        define empty is
            return (self.card = 0);
        define Stack is
            begin
                self.card := 0;
                self.elems.create;     !! initialisiere mit leerer Liste
        end define Stack;
end generic type Stack;
```

Einen „speziellen“ *Stack* für *Cuboid* Instanzen kann man nun wie folgt generieren:

```
type CuboidStack is Stack(Cuboid);
```

Auf analoge Weise kann man einen *CylinderStack* erzeugen:

```
type CylinderStack is Stack(Cylinder);
```

Um jetzt *Cuboide* in einen Stack abzuspeichern, muß man sich dann noch eine Variable vom Typ *CuboidStack* deklarieren:

```
var ManyCuboids: CuboidStack;
    ...
```

```
ManyCuboids.create;
   ...
ManyCuboids.push(...);
```

Erwähnenswert ist hierbei noch, daß bei der Instantiierung der Typen *CylinderStack* und *CuboidStack* auch die Initialisierungs-Operation *Stack* instantiiert wird. Somit steht z.B. in *CylinderStack* anstelle der Operation *Stack* eine entsprechende Operation *CylinderStack* als Initialisierer zur Verfügung.

16 Nebenläufigkeit von GOM-Applikationen†

Um mehrere GOM-Applikationen gleichzeitig auf einem gemeinsamen Objektbestand arbeiten lassen zu können, ohne sich bei der Implementierung der einzelnen Anwendung um die hierbei möglicherweise auftretenden Effekte kümmern zu müssen, werden sogenannte *atomare Operationen* angeboten, die die Eigenschaften klassischer Transaktionen [BHG87]

1. Atomarität
2. Konsistenzerhaltung
3. Isolation
4. Dauerhaftigkeit

erfüllen.

Atomare Operationen werden folgendermaßen deklariert:[9]

atomic [virtual] [polymorphic] declare op: $[t\|\|]t_1, \ldots, t_n(\rightarrow \mid \leftarrow)t_{n+1}$
define op(...) **is**
  begin ⟨statements⟩ **end**;

begin entspricht in diesem Fall **BeginOfTransaction**, **end CommitTransaction**. Es können beliebige Arten von Operationen (also auch virtuelle, polymorphe) als atomar deklariert werden. Ebenso können sowohl freie als auch typ-assoziierte Operationen als atomar deklariert werden.

Eine atomare Operation kann mittels **abort** benutzer- und system-initiiert beendet werden. Nach einem **abort** wird der Zustand der Objektbank auf den am Anfang der Operation gültigen zurückgesetzt. Bei einem system-initiierten **abort** erfolgt darüberhinaus ein automatischer Neustart der abgebrochenen Operation, so daß hiervon ein Benutzer, bis auf die auftretende Verzögerung, nichts mitbekommt.

Ein benutzer-initiierter Abbruch einer Operation kann mittels einer vordefinierten freien Operation

declare aborted: → Bool;

abgefragt werden. *aborted* bezieht sich immer auf die *letzte* ausgeführte atomare Operation des aktuellen Anwendungsprozesses.

Variablen, an die ein Resultatwert zugewiesen werden sollte, enthalten nach Abbruch einer atomaren Operation ihren ursprünglichen Wert. Nicht-atomare Operationen können nicht mittels **abort** abgebrochen werden.

[9] Wie das † an der Überschrift dieses Abschnitts andeutet, sind atomare Operationen noch nicht in die Implementierung von GOMpl aufgenommen. Stattdessen stellt jeder mit dem Schlüsselwort **process** (siehe Abschnitt 2) definierte Anwenderprozeß eine eigene Transaktion dar. Mehrere Prozesse können also gleichzeitig auf einer Datenbasis arbeiten, wobei die korrekte Synchronisierung der Prozesse gewährleistet ist.

Alle vordefinierten Operationen, wie z.B. das Lesen und Schreiben von Attributen, sind atomar. Atomare Operationen können als nicht-atomare verfeinert werden und umgekehrt. Gleiches gilt für das Überladen von Operationen.

Wird ein ganzer Typ als atomar definiert,

atomic [virtual] [generic] type t ...

so sind alle Operationen dieses Typs atomar. Es können beliebige Arten von Typen (also auch virtuelle und generische) als atomar deklariert werden.

Nicht jede Operation muß (und sollte) als atomar deklariert werden. Wie die folgenden Beispiele zeigen, kann es durchaus sinnvoll sein, auch nicht-atomare Operationen zu deklarieren. Hierbei muß sich der Implementierer jedoch gewahr sein, daß möglicherweise gleichzeitig aktive Anwendungen, Effekte in der Objektbank erzeugen können, die seine Anwendung beeinflussen.

Beispiele

Wird das „Hauptprogramm“ einer Anwendung als atomar deklariert,

atomic process myProg ...

so läßt sich dieses Programm als Transaktion im herkömmlichen Sinne auffassen. Oftmals soll jedoch gerade vermieden werden, daß die gesamte Anwendung als eine Transaktion aufgefaßt wird, z.B. wenn es sich um eine interaktive Benutzerschnittstelle handelt. Diese würde sinnvollerweise folgendermaßen implementiert werden:

```
process menu: → void;
define menu is
   begin
      ⟨get input from terminal⟩
      while ⟨input⟩ ≠ "QUIT"
         begin
            switch (⟨input⟩)
               case ⟨option_1⟩: ⟨atomic operation_1⟩;
               ...
               case ⟨option_n⟩: ⟨atomic operation_n⟩;
            end
            ⟨get input from terminal⟩
         end
   end menu
```

In vielen Fällen kann es durchaus sinnvoll sein, *observierende* Operationen auf Objekten als nicht-atomar zu deklarieren, wenn gewisse „Unschärfen“ beim Ergebnis in Kauf genommen werden können (z.B. eine informelle Bestandsaufnahme in einem Lager):

```
type stock is
   public request, supply, qoh→
   body [contents: {Item};
         qoh: int;]
   operations
      atomic declare request: int → {Item};
      atomic declare supply: {Item} → void;
      declare showContents: → {Item};
   implementation
      define request (q) is
         begin
            if (self.qoh ≥ q)
```

```
            begin
                self.qoh:= self.qoh − q;
                return self.contents.retrieve(q);
            end;
        end request;
    define supply (is) is
        begin
            self.contents.union(is);
            self.qoh:= self.qoh + is.cardinality;
        end supply;
    define showContents is
        return self.contents.select(self.qoh);
end type stock;
```

Durch die Atomarität der Operationen *request* und *supply* (und durch geeignete Wahl der öffentlich zugänglichen Operationen) ist immer gewährleistet, daß die Attribute *qoh* und *contents* konsistent sind, d.h. daß immer gilt *card(contents) = qoh*. Bei der Operation *showContents* kann dagegen der Fall eintreten, daß der durch *self.qoh* ermittelte Lagerbestand nicht mehr mit der Kardinalität der Menge *contents* übereinstimmt, wenn z.B. zwischen der Ausführung von *self.qoh* und *self.contents.select* durch eine gleichzeitig aktive Anwendung ein *retrieve* oder *supply* ausgeführt wurde[10].

17 Schnittstelle zwischen C und GOM

Dieser Abschnitt ist in zwei Teile gegliedert: zunächst wird die *C-Schnittstelle* beschrieben, die es ermöglicht, in C-Programmen auf GOM-Objekte zuzugreifen und von C-Funktionen aus GOM-Operationen aufzurufen. Danach folgt die Beschreibung der *GOM-Schnittstelle*, die die Einbettung von C-Konstrukten in GOM-Typdefinitionen ermöglicht.

17.1 C-Schnittstelle

Mittels GOM können komplette Datenbankanwendungen in einem objektorientierten Datenbankschema realisiert werden. Theoretisch ist es z.B. möglich, ein komplettes CAD-System in GOM zu realisieren. Dabei ergeben sich jedoch folgende Nachteile:

- Der Zugriff auf Daten über Objektidentifikatoren ist aufwendiger als der direkte Zugriff mittels Hauptspeicherzeigern. Dies bedeutet, daß sehr rechenintensive Anwendungen wie Simulationen oder CAD-Berechnungen in GOM ineffizienter ablaufen als in herkömmlichen Programmiersprachen, wenn keine besonderen Maßnahmen zur Zugriffsoptimierung eingesetzt werden.[11]

- Zu vielen Problemen oder Anwendungsgebieten existiert schon eine große Anzahl von Software-Paketen. Um beim Einsatz von GOM in solchen Gebieten die vorhandene Software wiederverwenden zu können, muß ein Schnittstellen-Modul zur Kopplung von GOM mit der vorhandenen Software entwickelt werden. Dieses Modul wird von dem Benutzer in C geschrieben und kann über die C-Schnittstelle von GOM auf die GOM-Objektbank zugreifen. Die aus der Objektbank gelesenen Daten können von dem Schnittstellenmodul in das Format der vorhandenen Software transformiert werden. Weiterhin kann das Schnittstellenmodul über die C-Schnittstelle auch Daten in der Objektbank persistent speichern.

[10] Es muß dabei von einer Implementierung von *select* ausgegangen werden, bei der die Anzahl zu entnehmender Elemente die tatsächlich vorhandene übersteigen kann.

[11] Für GOM werden derzeit solche Optimierungsmaßnahmen zur Beschleunigung des Zugriffs auf Objekte entwickelt.

Beide Probleme können also mit der C-Schnittstelle von GOM gelöst werden. Insgesamt bietet die C-Schnittstelle die im folgenden beschriebene Funktionalität:

- GOM-Objekte können von C aus gelesen und verändert werden.
- Modifizierte Objekte können persistent gespeichert werden.
- Objekte in der Objektbank können gelöscht werden.
- In GOM definierte Operationen können aufgerufen werden.

Im allgemeinen wird es nicht notwendig sein, für alle Typen und Operationen eines GOM-Schemas C-Konstrukte zu erzeugen. Stattdessen werden beim Übersetzens eines GOM-Schemas die Teile spezifiziert, für die eine C-Schnittstelle erzeugt werden soll. Dazu steht die Schnittstellendefinitionssprache GOMil zur Verfügung. Die Schnittstellendefinitionssprache GOMil ist beschrieben in [Zac92].

17.2 GOM-Schnittstelle†

Die GOM-Schnittstelle ermöglicht es, C-Konstrukte in einem GOM-Schema zu verwenden. Dabei können C-Konstrukte zum einen zur Definition der Struktur und der Basisfunktionalität von GOM-Typen verwendet werden, und zum anderen zur Implementierung von GOM-Operationen.

Für die Definition von Typen stehen in GOM die Typkonstruktoren

[ ], { }, < >

für tupelstrukturierte, mengenstrukturierte und listenstrukturierte Typen, das Konzept der generischen Typen zur Verfügung sowie eine bestimmte Anzahl atomarer Typen (Sorten) zur Verfügung. In gewissen Fällen kann dieses Anbegot nicht ausreichen. Stellen wir uns beispielsweise vor, daß wir einen Typ *Hashtabelle* definieren wollen, und wir in C bereits ein Hashtabellen-Modul zur Verfügung haben. Es liegt nun nahe, genau dieses Modul zur Definition eines neuen GOM-Typs zu verwenden. Dies wird durch die GOM-Schnittstelle ermöglicht.

Die zweite Aufgabe der GOM-Schnittstelle ist es, die Verwendung von C-Konstrukten in der Implementierung von GOM-Operationen zu verwenden. Dabei gibt es zwei Möglichkeiten:

- Die gesamte Implementierung einer GOM-Operation wird in C durchgeführt. In der **implementation**-Klausel steht nur vermerkt, wie die entsprechende C-Operation heißt und in welcher Datei sie zu finden ist.
- In der Implementierung einer Operation in GOMpl kann C-Code, sogenannter *inline*-Code, verwendet werden. Dieser wird von dem Compiler an der entsprechenden Stelle in den erzeugten C-Code eingebaut und beim Aufruf der Operation ausgeführt.

18 Ausblick

Alle in diesem Bericht beschriebenen Konzepte von GOMpl (außer den mit † gekennzeichneten) sind im Rahmen des GOM-Prototypen implementiert. Zukkünftige Versionen des GOM-Prototypen werden um weitere der hier mit † gekennzeichneten Konzepte erweitert werden. Über die in diesem Bericht beschriebenen Konzepte hinaus wurden innerhalb des GOM-Projektes weitere Sprachkonzepte für objektorientierte Datenbankprogrammiersprachen entwickelt:

- *Fashion*: Das **fashion**-Konstrukt ermöglicht die Erweiterung der Substiuierbarkeits-Beziehung zwischen Typen über die Subtyp-Beziehung hinaus [MZ92].

- *GOM-Anfragesprache GOMql*: Zur Formulierung von deklarativen Anfragen wurde die Anfragesprache GOMpl entwickelt.

- *Schemaverwaltung*: Konzepte für eine flexible Schemaverwaltung für objektorientierte Datenbanksysteme sind beschrieben in [MZ93].

Diese werden ebenfalls in Zukunft in den GOM-Prototypen aufgenommen werden.

19 Bibliographie

Beim Entwurf des persistenten Objektmodells GOM wurden viele Konzepte aus anderen Datenmodellen und Programmiersprachen entliehen. Das Konzept der Objektidentität wurde in ähnlich strikter Form in FAD [BBKV87] eingeführt. Eine detaillierte Diskussion der verschiedenen Objektidentifikationsmechanismen findet sich in [KC86]. Anders als in FAD unterscheiden wir jedoch zwischen (atomaren) Werten, die keine Identität besitzen, und komplexen Objekten, die eine während ihrer Lebensdauer invariant bleibende Identität haben. In diesem Ansatz stimmen wir mit dem Vorgehen in FOOPS überein [GW90]. Die in GOM eingebauten Typkonstruktoren, also *Tupel-*, *Listen-* und *Mengen*-Konstruktor stimmen mit denen im erweiterten NF^2-Modell überein [D+86]. Den gleichen Satz eingebauter Konstruktoren bieten die Objektmodelle EXTRA [CDV88] und Orion [KCB88].

Das Objektreferenzierungs-Konzept, das wir in GOM verwenden, ist ähnlich dem im Datenmodell MAD eingebauten [HMWMS87]. Es zeigt auch große Ähnlichkeiten zu Ansätzen aus der Künstlichen Intelligenz. Die *Frame*-Konstrukte aus den Wissensrepräsentationssprachen, wie z.B. KRL [BW77, BW79], ähnelt den grundlegenden Konzepten der Objektreferenzierung und den damit einhergehenden "gemeinsamen Unterobjekten" in GOM. Ein *slot* in einem *Frame* enthält entweder einen atomaren Wert oder eine Referenz auf ein anderes *Frame*-Objekt.

Wir unterscheiden uns von dem Objektmodell EXTRA was die Derefenzierung von Objekten anbelangt: während EXTRA die explizite Dereferenzierung verlangt werden GOM-Verweise implizit dereferenziert. Insofern gibt es keinen Unterschied zwischen dem Zugriff auf ein atomares Datum oder ein komplexes Objekt—wie dies auch von Beeri [Bee89] gefordert wird.

Natürlich haben wir beim Entwurf von GOM auch ganz wesentliche Anleihen aus dem Bereich der objektorientierten Programmiersprachen genommen. Das Unter-/Obertyp-Konzept hat seinen Ursprung in dem Vorläufer aller objektorientierten Programmiersprachen: Simula-67 [DMN70, ND81]. Simula-67 war die erste Programmiersprache, die eine Typhierarchie mit einhergehender Substituierbarkeit der Untertyp-Instanzen für Obertyp-Instanzen einführte. Direkte Nachfolger von Simula-67 sind die Sprachen Smalltalk-80 [GR83], Eiffel [Mey88], ObjectiveC [Cox86] und C++—um nur einige wenige zu nennen. Es gibt mittlerweile auch etliche Lisp-basierte objektorientierte Programmiersprachen. Die wichtigsten sind: Loops [BS82] und Flavors [Moo86].

Obwohl GOM sehr viel früher entworfen wurde, entsprechen die wesentlichsten Konzepte unseres Objektmodells doch denen, die im "Manifesto über Objektmodelle" [ABD+89] als Minimalfunktionalität für ein objektorientiertes Datenmodell gefordert sind. Weitere Ähnlichkeit weist GOM—obwohl es auch vor dieser Veröffentlichung entstand—mit dem von Zdonik und Maier definierten Referenzmodell auf [ZM89].

Weitere objektorientierte Datenmodelle sind: O_2 [LR89, Deu90], GemStone [MS87] und Iris [WLH90].

Die folgenden Bücher über objektorientierte Datenbanksysteme sind als weiterführende Literatur zu nennen: [KL89, ZM89]

Danksagung

Diese Arbeit wurde gefördert von der Deutschen Forschungsgemeinschaft (DFG) im Rahmen des Sonderforschungsbereichs 346 "Rechnerintegrierte Konstruktion und Fertigung von Bauteilen".

An der Realisierung des GOM-Prototypen sowie an weiteren Arbeiten im GOM-Projekt waren viele Studenten beteiligt, ohne deren Einsatz die Ergebnisse des GOM-Projektes nicht erreicht worden wären. Unser Dank gilt Eckard Appel, Monika Altmann, Alex Armbruster, Kai Bruns, Uwe Degel, Thomas Demmler, Siegfried Diesch, Carsten Gerlhof, Fr'ed'eric Gonot, Petra Geutner, Johannes Gehrke, Wolfgang Häfelinger, Eric Hamann, Sven Helmer, Ingo Heuten, Andreas Horder, Birgitta König-Ries, Donald Kossmann, Kai Leberer, Angela Lopes de Lima, Thorsten Mehnert, Kurt Moos, Christoph Müller, Ralf Müller, Thomas Müller, Kioumars Namiri, Achim Neumann, Uwe Oetken, Helmut Ott, Apostolos Papapostolou, Klaus Peithner, Dorothee Ruehl, Christian Salzmann, Georg Schifferdecker, Ulrich Schreiber, Bertil Sobottke, Heiner Spies, Michael Steinbrunn, Stefan Strugies, Axel Tetzner, Stefan Thoma, Rolf Veith, Karl-Heinz Vetter, Stefan Voss, Edmund Wanner, Rüdiger Waurig, Martin Wehr, Michaela Werner, Ute Wüst und Jürgen Zimmermann.

Literatur

[ABD+89] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. Int. Conf on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, Dec 1989.

[BBKV87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a powerful and simple database language. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 97–105, Brighton, U.K., Sep 1987.

[Bee89] C. Beeri. Formal models for object-oriented databases. In *Proc. of the DOOD Conference*, pages 370–395, Kyoto, Japan, Dec 1989.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[BS82] D. G. Bobrow and M. J. Stefik. LOOPS: An object-oriented programming system for interlisp. Technical report, XEROX PARC, 1982.

[BW77] D. G. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1):3–46, 1977.

[BW79] D. G. Bobrow and T. Winograd. KRL, another perspective. *Cognitive Science*, 3(1), 1979.

[CDV88] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 413–423, Chicago, Il., Jun 1988.

[Cox86] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison Wesley, Reading, MA, 1986.

[D+86] P. Dadam et al. A DBMS prototype to support extended nf^2 relations: An integrated view on flat tables and hierarchies. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 376–387, Washington, DC, 1986.

[Deu90] O. Deux et al. The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, Mar 1990.

[DMN70] O. J. Dahl, B. Myrhaug, and K. Nygaard. Simula 67: Common base language. Publication NS 22, Norsk Regnesentral (Norwegian Computing Center), Oslo, Norway, Oct 1970.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[GW90] J. A. Goguen and D. Wolfram. On types and FOOPS. In *Proc. IFIP TC-2 Conf. on Object-Oriented Databases*, Windermere, UK, Jun 90.

[HMWMS87] T. Härder, K. Meyer-Wegener, B. Mitschang, and A. Sikeler. PRIMA – a DBMS prototype supporting engineering applications. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 433–442, Brighton, UK, Sep 1987.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. *ACM SIGPLAN Notices*, 21(11):408–416, Nov 1986. Proc. of the OOPSLA Conference.

[KCB88] W. Kim, H. T. Chou, and J. Banerjee. Operations and implementation of complex objects. *IEEE Trans. on Software Engineering*, 14(7):985–996, Jul 1988.

[KKM+90] A. Kemper, C. Kilger, G. Moerkotte, H.-D. Walter, and A. Zachmann. *Das GOM-Handbuch — Objektorientierte Programmierung und Datenmodellierung.* Inst. f. Programmstrukturen u. Datenorganisation, Univ. Karlsruhe, 7500 Karlsruhe, 1990.

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

[LR89] C. Lécluse and P. Richard. The O_2 database programming language. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 411–422, Amsterdam, NL, Sep 1989.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice Hall, 1988.

[Moo86] D. A. Moon. Object-oriented programming with flavors. In *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 1–8, 1986.

[MS87] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392, Cambridge, MA, 1987. MIT Press.

[MZ92] G. Moerkotte and A. Zachmann. Multiple substitutability without affecting the taxonomy. In *Proc. Intl. Conf. on Extending Database Technology (EDBT)*, pages 120–135, Wien, Mar 1992.

[MZ93] G. Moerkotte and A. Zachmann. Towards more flexible schema management in object bases. In *Proc. IEEE Conference on Data Engineering*, Vienna, 1993.

[ND81] K. Nygaard and O. J. Dahl. Simula 67. In R. W. Wexelblat, editor, *History of Programming Languages*, 1981.

[SZ87] K. E. Smith and S. B. Zdonik. Intermedia: A case study of the differences between relational and object-oriented database syst ems. In *Proc. of the ACM Conf on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 452–465, Oct 1987.

[WLH90] K. Wilkinson, P. Lyngbaek, and W. Hasan. The iris architecture and implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, Mar 1990.

[Zac92] A. Zachmann. Benutzung der tutorial-version des gom-systems, Oktober 1992. Release 1.4.

[ZM89] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 1–32. Morgan-Kaufman Publ. Co., 89.